MALMÖ HÖGSKOLA

Faculty of Technology and Society

Programming C# II/VB II

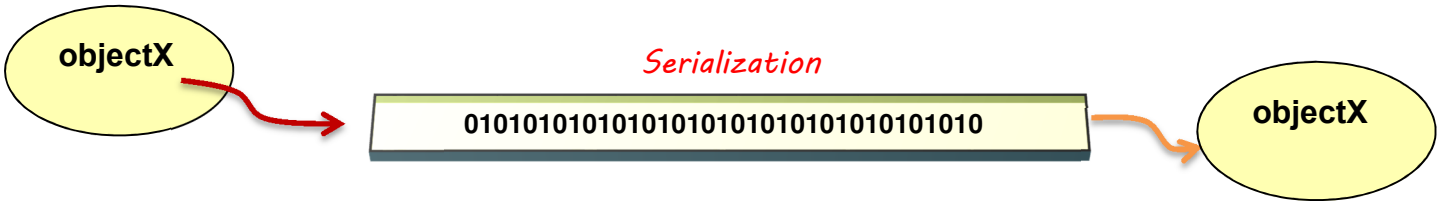
Binary and XML Object Serialization

Farid Naisan
University Lecturer
Department of Computer Science

Contents

Serialization

Serialization is a process of persisting and transferring the state of an object into a stream (e.g., file stream and memory stream). The persisted data sequence contains all the necessary information you need to reconstruct (or deserialize) the state of the object for later use. Serialization is the process of converting an object into a series bytes for storage to a disk or transferring to another location, for example to a remote computer or over a network. Serialization can be used to save an object to a permanent memory like a hard disk or to a buffer. It is like sending



Using the .NET object serialization, a whole object, together with all other objects that belong to the object through inheritance and aggregation, can be saved with a very few lines of code, although the process is very complicated behind the scenes.

Sometimes other terms are used for the processes Serialization/deserialization for example marshalling/unmarshalling or deflating/inflating of objects. There are two sorts of serialization in .NET – binary and XML Serialization.

For binary serialization, the following namespaces are to be imported:

```
System.Runtime.Serialization
System.Runtime.Serialization.Formatters.Binary
System.IO
```

The following namespace will be needed for XML serialization:

```
System.Xml.Serialization
```

For working with files and directories, you will need to import the name space:

```
System.IO
```

Serialization can be done using different formats, binary, SOAP, or XML. In this paper we skip the SOAP format and concentrate on the other two types. To serialize an object, you must stamp the object with the attribute **[Serializable]** in C# and **<Serializable()>** in Visual Basic as used in the code examples.

Note: In the code examples that are included in this document, it is assumed that the required namespaces are imported in the code files. Remember also that:

- the reason for using short variable names is only to fit the text into the page, and
- using the method call **throw** in the **catch** statement is for simplicity. You should handle the exceptions in a proper way in the **catch** blocks.

1. A few words about .NET Attributes

An attribute is a declarative tag used to convey information to runtime about the behaviours of various elements. Attributes can be applied to a type (classes, interface, structures, and enumerators), a member (methods and properties), an assembly or a module.

A declarative tag is depicted by square brackets ([ ]) in C" and the signs < > (followed by an underscore char '_') in VB, placed directly above the element it is used for. Attributes are used for adding metadata, such as compiler instruction and other information such as comments, description, methods and classes to a program. The .NEt attributes are class types that extend the abstract System.Attribute base class. The .NET Framework provides two types of attributes: the pre-defined attributes and custom built attributes. In the .NET namespaces, there are many predefined attributes that you are able to make use of in your applications. Among the predefined attributes are **Serializable**, **NonSerialized**, **Obsolete**, **DllImport**

Syntax for specifying an attribute is as follows:

C#
[attribute(positional_parameters, name_parameter = value, ...)]

Class definition or other element

VB:
<attribute(positional_parameters, name_parameter = value, ...)>_
Class definition or other element

Or writing on the same line:

<attribute(positional_parameters, name_parameter = value, ...)> Class definition or other element

Name of the attribute and its values are specified within the square brackets (or <> signs), before the element to which the attribute is applied. Positional parameters specify the essential information and the name parameters specify the optional information.

2. Binary Serialization

Binary Serialization is more effective than XML Serialization as it is fast and takes less memory space. However, because of the binary format, results of the serialization are not readable by humans.

C#	VB

`[Serializable]`
`public class MySerializableClass`
`{`
 	
`}`
	`<Serializable()> Public Class MySerializableClass`
	
	`End Class`

If a single member variable is to be excluded, it should be marked as **[NonSerialized]** and it will, therefore, not be persisted into the specified data stream.

C#
```
[NonSerialized] private int myInteger;
```

VB
```
<NonSerialized()> Private myInteger As Integer
```

The result of the above declaration is that the private member **myInteger** will not be serialized when the object gets serialized.

It is important to note that the attribute **Serializable** cannot be inherited and therefore the child classes must also be marked with the attribute **Serializable** in order for serialization to work. The same is true for objects which are associated with the object being serialized through aggregation. These must also be marked with the attribute **Serializable**.

2.1. Serialization to file

To save any serializable object to a file, the following general method can be used. The methods are general and can be used to serialize any type of object. The format that is used in this example, **BinaryFormatter**, is a compact binary format type

C#
```
public static void BinaryFileSerialize(Object obj,
                                       string filePath)
{
   FileStream fileStream = null;
   try
   {
      fileStream = new  FileStream(filePath,
                                   FileMode.Create);
      BinaryFormatter b =  new BinaryFormatter();
      b.Serialize(fileStream,obj);
   }
   catch
   {
      throw;  //Handle error instead
   }
```

VB
```
Public Shared Sub BinaryFileSerialize(ByVal obj As Object, _
                                      ByVal filePath As String)
   Dim fileStream As FileStream = Nothing

   Try
      fileStream = New FileStream(filePath, FileMode.Create)
      Dim b As BinaryFormatter = New BinaryFormatter()
      b.Serialize(fileStream, obj)
    Catch
       Throw  'Handle error instead
    Finally
       If fileStream IsNot Nothing Then
          fileStream.Close()
       End If
    End Try
End Sub
```

```
    finally
    {
       if (fileStream != null)
          fileStream. Close();
     }
}
```

To deserialize data that is serialized as above, you can make use of the method **b.Deserialize** as in the following example. As you can see from the above code, serialization can be generalized and you can use same code for all object types. Using generics, you write methods for both serialization and deserialization of any type of object. As an exercise, let us write a generic method to deserialize any arbitrary object.

C#

```
public static T BinaryFileDeSerialize<T>(string filePath)
{
    FileStream fileStream = null;
    Object obj;
    try
    {
       if (!File.Exists(filePath))
            throw new FileNotFoundException("The file" +
                        " was not found. ", filePath);
       fileStream = new FileStream(filePath, FileMode.Open);
       BinaryFormatter b = new BinaryFormatter();
       obj = b.Deserialize(fileStream);
    }
    catch
    {
        throw;
    }
    finally
    {
        if (fileStream != null)
            fileStream.Close();
    }
    return (T)obj;
}
```

VB

```
Public Shared Function BinaryFileDeSerialize(Of T)(ByVal_
                    filepath As String) As T
    Dim fileStream As FileStream = Nothing
    Dim obj As Object

    Try
        If Not File.Exists(filepath) Then
           Throw New FileNotFoundException("The file was _
                                    not found. ", filepath)
        End If

        fileStream = New FileStream(filepath, FileMode.Open)
        Dim b As BinaryFormatter = New BinaryFormatter()
        obj = b.Deserialize(fileStream)
    Catch
            Throw
    Finally
      If Not fileStream Is
         Nothing Then
         fileStream.Close()
      End If
    End Try
      Return CType(obj, T)
End Function
```

In much the same way, you can reconstruct the earlier serialize-methods as generic methods.

2.2. To binary array

In some cases you might need to serialize/deserialize an object, for instance an Image, as an array of bytes. The function below can serialize all serializable objects to a byte array:

C#
```
public static byte[] BinaryArraySerialize(Object obj)
{
    byte[] serializedObject;
    MemoryStream ms = new MemoryStream();
    BinaryFormatter b = new BinaryFormatter();
    b.Serialize(ms, obj);
    ms.Seek(0, 0);
    serializedObject = ms.ToArray();
    ms.Close();
    return serializedObject;
}
```

VB
```
Public Shared Function BinaryArraySerialize(ByVal obj As _
                    Object) As Byte()

    Dim serializedObject As Byte()
    Dim ms As MemoryStream = New MemoryStream()
    Dim b As BinaryFormatter = New BinaryFormatter()
    b.Serialize(ms, obj)
    ms.Seek(0, 0)
    serializedObject = ms.ToArray()
    ms.Close()
    Return serializedObject
End Function
```

To deserialize the array back to an object, the method below can be used. It is also written in a general form using generics.

C#
```
public static T BinaryArrayDeSerialize<T>(byte[] serializedObject)
{
    MemoryStream ms = new  MemoryStream();
    ms.Write(serializedObject,  0, serializedObject.Length);
    ms.Seek(0, 0);
    BinaryFormatter b = new BinaryFormatter();
    Object obj = b.Deserialize(ms);
```

VB
```
Public Shared Function BinaryArrayDeSerialize(Of T)(ByVal _
                        serializedObject As Byte()) As T
    Dim ms As MemoryStream = New MemoryStream
    ms.Write(serializedObject, 0, serializedObject.Length)
    MS.Seek(0, 0)
    Dim b As BinaryFormatter = New BinaryFormatter()
    Dim obj As Object = b.Deserialize(ms)
```

```
    ms.Close();
    return (T)obj;
}
```

```
MS.Close()
    Return CType(obj, T)
End Function
```

2.3. Example – Binary Serialization

Using the above **BinaryFileSerialize** and **BinaryFileDeserialize** methods, write code to serialize and desialize objects of the class **Person**.

C#

```
[Serializable]
public class Person
{
    private string firstName;
    private string lastName;

    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }

    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }

    public string FullName
    {
      get{ return FirstName + " " + LastName; }
    }
```

VB

```
<Serializable()> _
Public Class Person
   Private m_firstName As String
   Private m_lastName As String

   Public Property FirstName() As String
     Get
        Return m_firstName
     End Get
     Set(ByVal value As String)
          m_firstName = value
     End Set
   End Property

   Public Property LastName As String
     Get
        Return m_lastName
     End Get
     Set(ByVal value As String)
        m_lastName = value
     End Set
   End Property
```

```
    public Person(string firstName, string lastName)
    {
        this.FirstName = firstName;
        this.LastName = lastName;
    }
}
```

```
    Public ReadOnly Property FullName As String
        Get
          Return FirstName + " " + LastName
          End Get
    End Property

    Public Sub New(ByVal fstName As String, ByVal _
                                              lstName As String)
       Me.FirstName = fstName
       Me.LastName = lstName
    End Sub
End Class
```

To test this example, write an empty class **Serializer** and save it asSerializer.cs (C#) or Serializer.vb (VB). Copy and paste the methods, **BinaryFileSerialize**, **BinaryFileDeSerialize** as well as the corresponding methods for serialization of byte arrays.

2.3.1. Serializing an object of Person to file

You can now test the above methods with the following code.

C#
```
Person p = new Person("Kalle",  "Karlsson");
Serializer.BinaryFileSerialize(p, "kalle.dat");
```

VB
```
Dim p As Person = New Person("Kalle", "Karlsson")
Serializer.BinaryFileSerialize(p, "kalle.dat")
```

Note that the binary file, kalle.dat is saved to disk. Loading the object back from the file can be done by calling the method **BinaryFileDeserialize.** Also note how the generic methods is called specifying the return type (Person).

C#
```
Person p =
Serializer.BinaryFileDeSerialize<Person>("kalle.dat");
```

VB
```
Dim p As Person = Serializer.binaryFileDeSerialize(Of _
                                Person)("kalle.dat")
```

2.3.2. Serializing an object of Person to Array

C#
```
Person p = new Person("Kalle", "Karlsson");

byte[] pArray = Serializer.BinaryArraySerialize(p);

Person p2 =
  Serializer.BinaryArrayDeSerialize<Person>(pArray);
```

VB
```
Dim p As Person = New Person("Kalle", "Karlsson")

Dim pArray As Byte() = Serializer.binaryArraySerialize(p)

Dim p2 As Person = Serializer.binaryArrayDeSerialize(Of _
Person)(pArray)
```

2.4. More control

If you wish to have more control over the process of serialization of an object, you can implement the interface **ISerializable**.:

C#
```
public class MyClass: Serializable
{
. . .
```

VB
```
Public Class MyOwnClass
    Implements ISerializable
```

You will be then required to implement the method **GetObjectData** that is a part of the interface. Visit the MSDN site and read the documentations about how the interface **ISerializable** can be used.

3. Xml Serialization

XML Serialization is much easier to understand as it creates a readable textual result, but requires a good amount of work compared to binary serialization. The System.Xml.dll assembly provides a formatter, **System.Xml.Serialization.XmlSerializer** to persist the **public** state of a given object as pure XML. Working with XML type is a bit different from working with the BinaryFormatter type. The key difference is that the XmlSerializer requires you to specify type information that represents the class you want to serialize. Assuming that you have imported the **System.Xml.Serialization** namespace, consider the following line of code:

C#:

```
XmlSerializer xmlFormat = new XmlSerializer(typeof(Person));
```

VB:

```
Dim xmlFormat As XmlSerializer = New XmlSerializer(GetType(Person))
```

Another issue that should be kept in mind is that the XmlSerializer specifically requires the class that it serializes to have a default (parameterless) constructor.

Before we go further, it would be helpful to present a very brief overview of the XML, just in case you are not familiar with this very popular mark-up language (and a huge subject).

3.1 Brief overview of XML

Extensible Mark-up Language (XML) is a mark-up language with tags used to structure information into elements. An element is between a matching tag pair. An XML document always contains a **root** element:

```
<Root>
</Root>
```

Which contain elements:

```
<Root>
   <Element1></Element1>
   <Element2></Element2>
</Root>
```

Elements can also contain elements:

```
<Root>
   <Element1>
      <Element1. 1></Element1. 1>
   </Element1>
   <Element2></Element2>
</Root>
```

If an element is “empty” (does not contain any other element), the syntax <element/> can be used as an alternative to the tag pair <element></element> with nothing inside.

```
<Root>
   <Element1>
      <Element1. 1/>
   </Element1>
   <Element2/>
</Root>
```

The elements can contain attributes that can have values:

```
<Root>
   <Element1 Attribute1=”attributeValue” Attribute2=” attributeValue”>
      <Element1.1 />
   </Element1>
   <Element2 Attribute2=” attributeValue”/>
</Root>
```

An element can also have a value:

```
<Root>
   <Element1 Attribute1=” attributeValue” Attribute2=” attributeValue”>
      <Element1.1>elementValue</Element1.1>
   </Element1>
   <Element2 Attribut2=” attributeValue”/>
</Root>
```

As simple as above, you now have a general picture of the main structure of an XML document – not really, but enough to go on with our topic. XML is a flexible and extremely powerful especially if some rules are added.

3.2 XML Serialization

A method for XML serialization can be defined in a general form using generics as shown here:

C#

```
public static void XmlFileSerialize<T>(string filePath,
                                       T obj)
{
   XmlSerializer s = new XmlSerializer(typeof(T));
   TextWriter w = new  StreamWriter(filePath);
```

VB

```
Public Shared Sub XmlFileSerialize(Of T)(ByVal filePath _
                                  As String, ByVal obj As T)

   Dim s As XmlSerializer = New XmlSerializer(GetType(T))
   Dim w As TextWriter = New StreamWriter(filePath)
```

```
    try
    {
        s. Serialize(w, obj);
    }
    catch
    {
        throw;
    }
    finally
    {
        if(w!=null) w. Close();
    }
}
```

```
    Try
        s.Serialize(w, obj)
    Catch
        Throw
    Finally
        If Not w Is Nothing Then
            w.Close()
        End If
    End Try
End Sub
```

Now, how to use the above? Let's test with the Person class given above:

C#
```
Person p = new Person("Kalle", "Karlsson");
Serializer.XmlFileSerialize<Person>("test. xml")
```

VB
```
Dim p As Person = New Person("Kalle", "Karlsson")
Serialization.xmlFileSerialize(Of Person)("test. xml", p)
```

Well – this code will not work and hopefully you can figure out why. If you don't, remember that XML Serialization requires a default constructor (constructor without arguments) as mentioned earlier. So, we need to add a default constructor in the Person class and try again. We can of course keep the constructor with parameters and also define properties to provide access to the object's private members. By doing that and then testing the code, assuming everything goes well, the file **test.xml** should be created on your computer and should like something like this:

```
<?xml version="1. 0" encoding="utf-8" ?>
 <Person xmlns:xsi=http://www. w3. org/2001/XMLSchema-instance
     xmlns:xsd="http://www. w3. org/2001/XMLSchema">
   <FirstName>Kalle</FirstName>
   <LastName>Karlsson</LastName>
</Person>
```

The object **p** has turned into a readable textual file! The first row describes the XML document and the encoding used. Next is our **root** object **Person**. It has two automatically generated attributes that indicate the rules that are used for the object structure. Then there are the two public name properties with the instance values of the object as we had specified them.

The class also has a property called **FullName**, but since there is no set method for setting data, it is ignored in the process. To generalize the whole thing, we can write a generic method for XML serialization that should work for all types of objects.

3.3 XML Deserialization

XML Deserialization is as easy as Serialization. Let's first consider the following generic method:

C#

```
public static T XmlFileDeserialize<T>(string
filePath)
{
   XmlSerializer s = new
                        XmlSerializer(typeof(T));
   TextReader r = new StreamReader("test.xml");

try
   {
      return (T)s.Deserialize(r);
   }
   catch
   {
     throw;
   }
   finally
   {
    if (r != null) r.Close();
  }
}
```

VB

```
Public Shared Function XmlFileDeserialize(Of T)(ByVal _
                          filePath As String) As T
   Dim s As XmlSerializer = New XmlSerializer(GetType(T))
   Dim r As TextReader = New StreamReader(filePath)

   Try
      Return CType(s.Deserialize(r), T)
   Catch
      Throw
   End Try

 End Function
```

The method can then be used to deserialize the file **test.xml** back to an object:

C#	VB
`Person p = Serialization.xmlFileSerialize <Person> ("test. xml");`	`Dim p As Person = _ Serialization.xmlFileDeSerialize(Of Person)("test. xml")`

3.4 More control

By default, **XmlSerializer** serializes all public fields and properties as XML elements, rather than as XML attributes. If you want to control how the **XmlSerializer** generates the resulting XML document, you can decorate types with any number of additional .NET attributes from the **System.Xml.Serialization** namespace. The namespace contains many .NET attributes that influence how XML data can be encoded to a stream. Some of these are [XmlRoot], [XmlAttribute], [XmlElement], [XmlEnum], [XmlText] and [XmlType] (VB: change [ ] to < >). We examine a number of these.

The name of the root element

To set the name of the root element explicitly, do as shown below:

C#	VB

`[XmlRoot("RootNamn")]`
`public class MyClass`
`{`
`. . .`
	`<XmlRoot("RootNamn")> _`
	`Public Class MyClass`
	`. . .`

Element name

Explicitly set an element name:

C#	VB

`[XmlElement("ElementNamn")]`
`public string MinProperty`
`{ . . .`
	`<XmlElement("ElementNamn")> _`
	`Public Property MinProperty() As String`
	`…`

Attribute

Make the property an attribute instead of an element:

C#	VB

`[XmlAttribute("AttributNamn")]`
`public string MinProperty`
`{`
`. . .`
	`<XmlElement("AttributNamn ")> _`
	`Public Property MinProperty() As String`
	`…`

Ignore

It is sometimes not possible or not necessary to XML-serialize a variable or a property. In these cases, the serialization of the member in question can be skipped by using the attribute **XMLIgnore** as listed below:

C#	VB

`[XmlAttribute("AttributNamn")]`
`public string MinProperty`
`{`
`. . .`
	`<XmlElement("AttributNamn ")> _`
	`Public Property MinProperty() As String`
	`…`

4. Serializing collections of objects

The **Serialize()** method of the **IFormatter** interface serializes only a sing **System.Object** and does not provide a way to specify to serialize a collection of objects. Likewise, the return value of the **Deserialize()** is also a single **System.Object**. This limitation is true for XMLSerializer and XMLDesializer as well. However, this does not mean that collections cannot be serialized or deserialized.

As mentioned earlier, if you pass in an object that has been marked as **Serializable** and contains other **Serializable** objects, the entire set of objects is persisted in a single method call. Fortunately, most of the types included in the **System.Collections** and **System.Collections.Generic** namespaces have already been marked as **Serializable**. Therefore, if you would like to persist a set of objects, simply add the desired set to the container such as a normal array, an **ArrayList** or a **List<T>** in C# or **List (Of T)** in VB, and serialize the object to a stream of your choice. Here is an example. Assume that we have a collection of **Product** objects.

C#

```
[Serializable]
public class Product
{
    private string name;
    private double price;

    //XMLSerializer demans a default constructor
    public Product()
    {
        name = string.Empty;
    }

    public Product(string name, double price)
    {
        this.name = name;
        this.price = price;
    }

    public string Name
    {
        get { return name; }
        set { name = value; }
    }
```

VB

```
<Serializable> _
Public Class Product
    Private m_name As String
    Private m_price As Double

    'XMLSerializer demans a default constructor
    Public Sub New()
        m_name = String.Empty
    End Sub

    Public Sub New(name As String, price As Double)
        Me.m_name = name
        Me.m_price = price
    End Sub

    Public Property Name() As String
        Get
            Return m_name
        End Get
        Set(value As String)
            m_name = value
        End Set
    End Property

    Public Property Price() As Double
        Get
```

```
        public double Price
    {
        get { return price; }
        set { price = value; }
    }

}
```

```
            Return m_price
        End Get
        Set(value As Double)
            m_price = value
        End Set
    End Property

End Class
```

C#

```
public class ListSerializer
{
    private List<Product> productList = new List<Product>();

    public ListSerializer()
    {
        //putt some test values
        productList.Add(new Product("Product 1", 100.0));
        productList.Add(new Product("product 2", 200.0));
        productList.Add(new Product("product 3", 300.0));
        productList.Add(new Product("product 4", 400.0));
    }

    public void SerializeListOfProductsXML()
    {
        // Now persist a List<T> of Products.
        using (Stream stream =
                  new FileStream("ProductCollection.xml",
                      FileMode.Create, FileAccess.Write))
        {
            XmlSerializer xmlFormat =
                   new XmlSerializer(GetType(productList));
            xmlFormat.Serialize(stream, productList);
        }

    }
```

VB

```
Public Class ListSerializer
  Private productList As New List(Of Product)()

  Public Sub New()
    'putt some test values
    productList.Add(New Product("Product 1", 100.0))
    productList.Add(New Product("product 2", 200.0))
    productList.Add(New Product("product 3", 300.0))
    productList.Add(New Product("product 4", 400.0))
  End Sub

  Public Sub SerializeListOfProductsXML()
    ' Now persist a List<T> of Products.
    Using stream As Stream = _
             New FileStream("ProductCollection.xml",_
                   FileMode.Create, FileAccess.Write)
       Dim xmlFormat As New
XmlSerializer(GetType(productList)))
       xmlFormat.Serialize(stream, productList)
    End Using

   End Sub
```

```
    public void SerializeListOfProductsBin()
    {

        using (Stream stream =
                  File.Open("ProductCollection.dat",
                     FileMode.Create, FileAccess.Write))
        {
            BinaryFormatter bin = new BinaryFormatter();
            bin.Serialize(stream, productList);
        }

    }

    public void DerializeListOfProductsBin()
    {
        using (Stream stream =
                  File.Open("ProductCollection.dat",
                  FileMode.Open))
        {
            BinaryFormatter bin = new BinaryFormatter();

            productList.Clear();  //remove all objects
            productList =
                 (List<Product>)bin.Deserialize(stream);
        }
    }
}
```

```
Public Sub SerializeListOfProductsBin()

    Using stream As Stream =
                File.Open("ProductCollection.dat", _
                      FileMode.Create, FileAccess.Write)
      Dim bin As New BinaryFormatter()
      bin.Serialize(stream, productList)
   End Using

  End Sub

  Public Sub DerializeListOfProductsBin()
    Using stream As Stream =
                     File.Open("ProductCollection.dat",
                     FileMode.Open)
      Dim bin As New BinaryFormatter()

      productList.Clear()            'remove all objects
      productList = DirectCast(bin.Deserialize(stream), _
                         List(Of Product))
    End Using
  End Sub

End Class
```

Calling the method SerializeListOfProductsXML from the above code will create a file with the following look:

```
<?xml version="1.0"?>
<ArrayOfProduct xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Product>
    <Name>Product 1</Name>
    <Price>100</Price>
  </Product>
  <Product>
    <Name>product 2</Name>
    <Price>200</Price>
  </Product>
  <Product>
    <Name>product 3</Name>
    <Price>300</Price>
  </Product>
  <Product>
    <Name>product 4</Name>
    <Price>400</Price>
  </Product>
</ArrayOfProduct>
```

You can also generalize the serialization and deserialization of collections (for example List <T> or List (Of T)) using generics.

Last but not least, you can create a utility class, e.g. **SerializationUtility** and write all related generic methods in that class. You can then use (reuse) the class in all your projects.

5. Links

Binary serialization: http://msdn.microsoft.com/en-us/library/72hyey7b(v=vs.110).aspx

MSDN: XML serialization class: http://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlserializer.aspx'