.NET Serialization Articles

Contents

[1. Standard Serialization 1](#_Toc430778715)

[XML Serialization and Deserialization: Part-1 1](#_Toc430778716)

[Introduction 1](#_Toc430778717)

[XML Serialization and Attributes 2](#_Toc430778718)

[Object List Serialization 5](#_Toc430778719)

[**Serialization of classes containing other class objects** 6](#_Toc430778720)

[Deserialization of XML 8](#_Toc430778721)

[Conclusion 8](#_Toc430778722)

[XML Serialization and Deserialization: Part-2 9](#_Toc430778723)

[Introduction 10](#_Toc430778724)

[XML attributes while deserialization 13](#_Toc430778725)

[Conclusion: 16](#_Toc430778726)

[2. Custom Serialization 17](#_Toc430778727)

[Custom Serialization Example 17](#_Toc430778728)

[http://www.codeproject.com/Articles/22787/Custom-Serialization-Example 17](#_Toc430778729)

[Introduction 17](#_Toc430778730)

[Background 17](#_Toc430778731)

[Using the code 17](#_Toc430778732)

[Additional Food for Thought 27](#_Toc430778733)

[3. A Fast Serialization Technique 29](#_Toc430778734)

[Introduction 29](#_Toc430778735)

[Background 29](#_Toc430778736)

[Using the code 29](#_Toc430778737)

[Results 31](#_Toc430778738)

[Summary 31](#_Toc430778739)

[History 32](#_Toc430778740)

# Standard Serialization

## XML Serialization and Deserialization: Part-1

<http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part?display=Print>

[Mayank Gupta 688](http://www.codeproject.com/script/Membership/View.aspx?mid=7572025), 2 Jan 2015 [CPOL](http://www.codeproject.com/info/cpol10.aspx)http://www.codeproject.com/App_Themes/CodeProject/Img/read32.png 264.5Khttp://www.codeproject.com/App_Themes/CodeProject/Img/bookmark32.png 245

XML serialization and deserialization using C#.

Introduction

The article talks about serialization of objects in XML format and deserialization of an XML file back to an object. Serialization is a process by which an object's state is transformed in some serial data format, such as XML or binary format. Deserialization on the other hand is used to convert the byte of data, such as XML or binary data, to object type. Serialization is the process of converting an object into a form that can be readily transported. For example, you can serialize an object and transport it over the Internet using HTTP between a client and a server. On the other end, deserialization reconstructs the object from the stream. XML serialization results in strongly typed classes with public properties and fields that are converted to a serial format (in this case, XML) for storage or transport.

Let's start with a basic example. Here is a simple class the need to be serialized :

public class AddressDetails

{

public int HouseNo { get; set; }

public string StreetName { get; set; }

public string City { get; set; }

private string PoAddress { get; set; }

}

The following points should be noted while creating a class for serialization:

1. XML serialization only serializes public fields and properties.
2. XML serialization does not include any type information.
3. We need to have a default/ non-parameterised constructor in order to serialize an object.
4. ReadOnly properties are not serialized.

Code to serialize the above class:

public static void Main(string[] args)

{

AddressDetails details = new AddressDetails();

details.HouseNo = 4;

details.StreeName = "Rohini";

details.City = "Delhi";

Serialize(details);

}

static public void Serialize(AddressDetails details)

{

XmlSerializer serializer = new XmlSerializer(typeof(AddressDetails));

using (TextWriter writer = new StreamWriter(@"C:\Xml.xml"))

{

serializer.Serialize(writer, details);

}

}

XmlSerializer (located in the System.Xml.Serialization namespace) class is used to serialize and deserialize. The class method Serialize is called. Since we have to serialize in a file we create a "TextWriter". Since TextWriter implements IDisposable, we used using so that we need not close the writer.

The output after the serialization is :

<?xml version="1.0" encoding="utf-8"?>

<AddressDetails>

<HouseNo>4</HouseNo>

<StreetName>Rohini</StreetName>

<City>Delhi</City>

</AddressDetails>

Here in the XML we can see that the Head tag of the XML created is same as that of the class name and the subtag names are same as the properties in class AddressDetails. Each public property is displayed in the form of Tags in the XML created. We can observe here that only public fields are displayed here.

### XML Serialization and Attributes

Some common attributes that are available while Serialization are:

* XmlAttribute: This member will be serialized as an XML attribute
* XmlElement: The field will be serialized as an XML element
* XmlIgnore: Field will be ignored while Serialization
* XmlRoot: Represent XML document's root Element

#### **Use of XmlElement:**

Further if we need to have different Tag name in XML from the class property Name. We can introduce the XmlElement attribute to it in the class structure.

public class AddressDetails

{

[XmlElement(*"Number"*)]

public int HouseNo { get; set; }

[XmlElement("Street")]

public string StreetName { get; set; }

[XmlElement("CityName")]

}

**[XmlElement("Number")]** specifies that the property HouseNo will be serialized with the Tag name "Number" in the Xml File. It help us to map between the XML Tag name and the class Property Name. The resultant XML string with the Custom tag name is given below :

<AddressDetails>

*<Number>*4*</Number>*

<Street>Rohini</Street>

<CityName>Delhi</CityName>

</AddressDetails>

#### Use of XmlAttribute

If we want that the property HouseNo should occur as the attribute for the Tag AddressDetails then we should use XmlAttribute. XmlAttribute serializes the object property as the attribute for the parent tag. The following code illustrates the functionality:

public class AddresssDetails

{

[XmlAttribute]("Number")]

public int HouseNo { get; set; }

[XmlElement("Street")]

public string StreetName { get; set; }

[XmlElement("CityName")]

public string City {get; set;}

}

The XML serialized output for the code will be :

<AddressDetails Number*="4"*>

<Street>Rohini</Street>

<CityName>Delhi</CityName>

</AddressDetails>

Notice here, Since the class property HouseNo is specified as XMLAttribute therefore this property is an Attribute for the parent tag AddressDetails.

#### Use to XmlIgnore

By default, all public fields and public read/write properties are serialized by the [X](http://msdn.microsoft.com/en-us/library/system.xml.serialization.xmlserializer.aspx)mlSerializer . That is, the value of each public field or property is persisted as an XML element or XML attribute in an XML-document instance. In order to override this property apply XmlIgnore attribute to it. This will remove the element from the XML. The code below explains the following:

public class AddressDetails

{

[XmlElement("Number")]

public int HouseNo;

[XmlElement("Street")]

public string StreetName;

**[XmlIgnore]**

public string City;

}

Here we can see that the property City contains XmlIgnore attribute. The resultant XML created wont contain the City tag in it.

<AddressDetails>

<Number>4</Number>

<Street>ABC</Street>

</AddressDetails>

Notice here that the property City is not serialized because of the attribute XmlIgnore placed on it.

#### Use of XmlRoot

Every XML has a root element. By default the name of the root element is same as the name of the class that is serialized. In order to give a **custom name to the root element** of XML, we use XmlRoot attribute. Implementation of this attribute is provided below:

**[XmlRoot("Root")]**

public class AddressDetails

{

[XmlElement("Number")]

public int HouseNo;

[XmlElement("Street")]

public string StreetName;

[XmlElement("CityName")]

public string City;

}

Here we can see that the attribute XmlRoot is placed over AddressDetails class. This will now override the default serialization behavior which takes xml tag root name same as the class name. The XML will now have "Root" as the root tag.

**<Root>**

<HouseNo>4</HouseNo>

Notice here that the root tag here is now "Root" and not the Class name.

### Object List Serialization

Now let's try to serialize a **list of AddressDetails** object to XML file:

public static void Main(string[] args)

{

List<AddressDetails> AddressList = new List<AddressDetails>();

AddressDetails detail1 = new AddressDetails();

detail1.HouseNo ="4";

detail1.StreetName = "ABC";

detail1.City = "Delhi";

AddressDetails detail2 = new AddressDetails();

detail2.HouseNo ="3";

detail2.StreetName = "ABCD";

detail2.City = "New Delhi";

AddressList.Add(detail1);

AddressList.Add(detail2);

Serialize(AddressList);

}

public void Serialize(List<AddressDetails> list)

{

XmlSerializer serializer = new XmlSerializer(typeof(List<AddressDetails>));

using ( TextWriter writer = new StreamWriter( @"C:\Xml.txt")

{

**serializer.Serialize(writer, list)**

}

}

The XML output for the above execution will be :

<ArrayOfAddressDetails>

<AddressDetails>

<Number>4</Number>

<Street>ABC</Street>

<CityName>Delhi</CityName>

</AddressDetails>

<AddressDetails>

<Number>3</Number>

<Street>ABCD</Street>

<CityName>New Delhi</CityName>

</AddressDetails>

</ArrayOfAddressDetails>

Notice that the XML produced gives a list of AddressDetails object.

### **Serialization of classes containing other class objects**

If we have a class structure such that a class contains an object of other class and we want to include that class object also for serialization. Let's see the following example :

public class PersonalDetails

{

public string Name { get; set; }

public int Age { get; set; }

public Address address;

}

public class Address

{

public int HouseNo { get; set; }

public string StreetName { get; set; }

public string City { get; set; }

}

This is how the PersonalDetails class will be serialized:

<PersonalDetails>

<Name>Mayank</Name>

<Age>24</Age>

<Address>

<HouseNo>4</HouseNo>

<StreetName>Rohini</StreetName>

<City>Delhi</City>

</Address>

</PersonalDetails>

To add more complexity to it lets try creating the following XML structure :

<PersonalDetails>

<Name>Mayank</Name>

<Age>24</Age>

**<Address HouseNo="4">**

<StreetName>Rohini</StreetName>

<City>Delhi</City>

</Address>

</PersonalDetails>

Observe that the difference over here is that we need to have "HouseNo" as the attribute for the Address Tag.  Lets see what change will be made in class in order to create this structure:

public class PersonalDetails

{

public string Name { get; set; }

public int Age { get; set; }

public Address address;

public PersonalDetails()

{

Name = "Mayank";

Age = 24;

address = new Address();

}

}

public class Address

{

**[XmlAttribute("HouseNo")]**

public int HouseNo { get; set; }

public string StreetName { get; set; }

public string City { get; set; }

public Address()

{

HouseNo = 8;

StreetName = "Rohini";

City = "Delhi";

}

}

As per the requirement we wanted to have "HouseNo" as XML attribute instead of the normal XMLElement. Therefore we introduce "XmlAttribute" on property.

Let's try creating the following XML Structure:

<PersonalDetails>

<Name>Mayank</Name>

<Age>24</Age>

**<address HouseNo="8">Rohini</address>**

</PersonalDetails>

The difference over here is that we require the **StreetName as innertext** of the XML node "address". So in order to create such structure we have another attribute XmlText. This help us to add the particular property as innertext for a tag.

So the Code for creating such a structure is :

public class PersonalDetails

{

public string Name { get; set; }

public int Age { get; set; }

public Address address;

}

public class Address

{

[XmlAttribute("HouseNo")]

public int HouseNo { get; set; }

**[XmlText]**

public string StreetName { get; set; }

}

The "XmlText" attribute here adds StreetName as InnerText to the tag "address"

### Deserialization of XML

See the following article for details on deserialization :  [XML Serialization and Deserialization (Part-2)](http://www.codeproject.com/Articles/487571/XML-Serialization-and-Deserialization-Part-2).

Conclusion

Serialization is a very efficient way to convert the object to XML. This save lots of saving time and effort.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | [Some improvements](http://www.codeproject.com/Messages/5047055/Some-improvements.aspx) [Pin](http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part-1)  \_Noctis\_23-Apr-15 19:48 | professional | [\_Noctis\_](http://www.codeproject.com/script/Membership/View.aspx?mid=9699670) | 23-Apr-15 19:48 | |
| |  |  |  |  | | --- | --- | --- | --- | | [upvote](http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part-1#xx5047055xx) [downvote](http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part-1#xx5047055xx)  http://codeproject.cachefly.net/script/Forums/Images/t.gif | |  |  | | --- | --- | | |  | | --- | | One was already spotted, but here's what i've found:   After: "*The output after the serialization is :* " the following line should be part of the code.  missing property name after: **"[XmlElement("CityName")]"**  in "Use of XmlAttribute", extra square closing bracket in: "[XmlAttribute~~]~~("Number")]"  "  *" should be number* ***8*** *really.* | | | |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | [Possible errors?](http://www.codeproject.com/Messages/5024414/Possible-errors.aspx) [Pin](http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part-1)  Nitancy19-Mar-15 14:37 | member | [Nitancy](http://www.codeproject.com/script/Membership/View.aspx?mid=4243971) | 19-Mar-15 14:37 | |
| |  |  |  |  | | --- | --- | --- | --- | | [upvote](http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part-1#xx5024414xx) [downvote](http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part-1#xx5024414xx)  http://codeproject.cachefly.net/script/Forums/Images/t.gif | |  |  | | --- | --- | | |  | | --- | | I think I have run across some errors in the code in the article by “Mayank Gupta 688” entitled "XML Serialization and Deserialization: Part-1". I have found that:  #1 in Main  Hide   Copy Code  <pre lang="xml">details.StreeName = "Rohini"; *// This was originally .StreeName*  Hide   Copy Code  <pre lang="xml">details.StreetName = "Rohini"; *// But should have been .StreetName*  #2 the xml that is generated looks like this:  Hide   Copy Code  <?xml version="1.0" encoding="utf-8"?>  <AddressDetails xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">  <HouseNo>4</HouseNo>  <StreetName>Rohini</StreetName>  <City>Delhi</City>  </AddressDetails>  instead of what is in the article:  Hide   Copy Code  <AddressDetails>  <HouseNo>4</HouseNo>  <StreetName>Rohini</StreetName>  <City>Delhi</City>  </AddressDetails>  #3 in **Use of XmlAttribute** this code in the "class AddressDetails" generates this error: Invalid token '(' in class, struct, or interface member declaration  Hide   Copy Code  /public class AddresssDetails  //{  // [XmlAttribute]("Number")] //Error:  but removing the first "]" in the code fixes it:  Hide   Copy Code  [XmlAttribute("Number")]  #4 in **Object List Serialization** the detail1.HouseNo and detail2.HouseNo are both passed strings intead of ints generating this error: "Cannot implicity convert a type string to an int.". Removing the quote marks fixes these problems.   However, Serialize(AddressList); generates the Error message "An object reference is required for the non-static field, method, or property". I haven't been able to puzzle this one out yet.   Does anyone have any comments on how to fix this? I tried to contact the author ( [@Mayank](http://www.codeproject.com/Members/mayank) Gupta 688 ) but could not find a method to do so. XML Serialization and Deserialization: Part-2 [Mayank Gupta 688](http://www.codeproject.com/script/Membership/View.aspx?mid=7572025), 6 Nov 2012 [CPOL](http://www.codeproject.com/info/cpol10.aspx)http://www.codeproject.com/App_Themes/CodeProject/Img/read32.png 94.8Khttp://www.codeproject.com/App_Themes/CodeProject/Img/bookmark32.png 88  Serialization and deserialization of XML.  Introduction  My previous article [XML Serialization and Deserialization (Part-1)](http://www.codeproject.com/Articles/483055/XML-Serialization-and-Deserialization-Part-1), talk about serialization of object to XML form. In this article we will discuss about "**Deserialization of XML**" back to object form. ***Deserialization*** is used to convert bytes of data, such as XML or binary data, to "Object" type. An XML file can be reconverted back to an Object using deserialization.  Let's start with the basic example. Here is the XML file that need to be deserialized:  <AddressDetails>  <HouseNo>4</HouseNo>  <StreetName>Rohini</StreetName>  <City>Delhi</City>  </AddressDetails>  So in order to Deserialize this XML file, we need to create a class:  public class Address  {  public int HouseNo { get; set; }  public string StreetName { get; set; }  public string City { get; set; }  }  This class contain variable name same as that of XML tags, XML tag values by default get mapped to the corresponding variable in the class. "HouseNo" in class "Address" will be automatically mapped to XML tag "HouseNo".  Now lets see a basic program which will map this XML to the class object:  public static void Main(string[] args)  {  XmlSerializer deserializer = new XmlSerializer(typeof(Address));  TextReader reader = new StreamReader(@"D:\myXml.xml");  object obj = deserializer.Deserialize(reader);  Address XmlData = (Address)obj;  reader.Close();  }   "deserializer.Deserialize" function is used to deserialize the XML data which is there in XML file. Now since we have deserialized the XML file structure to object form, we can now access the Xml tag values  :  Address.HouseNo  Address.StreetName  Address.City  Following point should be noted while creating a class for **Deserialization** :   1. Class variable/property should always be declared as **public** 2. We need to have ***Default/ Non Parameterised*** Constructor in order to deserialize.   Any class without "***Default/ Non Parameterised***" constructor will result into an error since "deserializer.Deserialize(reader)" have no provision to pass value to parameterised constructor.  In the above code, we have simple XML Elements present with no sub Elements. Lets explore further and deal with some complicated situations where the XML Element may have further sub Tags:  Lets complicate the situation further and try to "**deserialize**" following "XML":  <?xml version="1.0"?>  <AddressDirectory>  <Address>  <HouseNo>1</HouseNo>  <StreetName>Pitampura</StreetName>  <City>Delhi</City>  </Address>  <Address>  <HouseNo>4</HouseNo>  <StreetName>Rohini</StreetName>  <City>Delhi</City>  </Address>  </AddressDirectory>  Let's see the difference over here. In the following XML, we have multiple "Address" tags. And also the Address tag contains further sub Tags. Therefore we need to create our class such that it can hold multiple "Address" tags and its sub tags. Let's see how we can create the class:  public class AddressDirectory  {  **[XmlElement(*"Address"*)]**  public **List<Address>** addressList = new List<Address>();  }  public class Address  {  public int HouseNo { get; set; }  public string StreetName { get; set; }  public string City { get; set; }  }  Remember here that when we create a class for **deserialization** of any tag, We can only drill down to a single level. To explain you this in simple words, let me take an example. In the above xml we have "AddressDirectory" tab. In order to Deserialize, we create a class for "AddressDirectory" tag. Now this class can have the **details of following** for  "AddressDirectory" XML tag:   1. Attribute of "AddressDirectory" tag if present 2. Its **Childnodes** example "Address" (can only access "Address tags", cannot drill down to child nodes of "Address" Tag) 3. InnerText (if present)   The class AddressDirectory cannot extract information about the child tags of "Address" and the "Address" tag attributes. In order to fetch info about the "***ChildNodes***" of "Address" tag, we need to create another class that can store the attribute info and childnode info (up to first level).  Over here we have created a class AddressDirectory that maps to the root tag element of "XML". And the root tag further contains "Address" tags. We can have multiple tags for "Address" over here, therefore we have created list of class "Address" in "AddressDirectory" class so that multiple "Address" tag info can be stored. Here in the class we can see XmlElement written over addressList. This attribute is used since the name of the class variable is different from that in XML file therefore in order to map the class variable with the XML tag, we use the keyword XmlElement. We will discuss about this later in the Article.  We will see more examples about this further in the article.  The program to be executed in order to deserialize the XML will be:  XmlSerializer deserializer = new XmlSerializer(typeof(AddressDirectory));  TextReader reader = new StreamReader(@"D:\myXml.xml");  object obj = **deserializer.Deserialize(reader);**  AddressDirectory XmlData = (AddressDirectory)obj;  reader.Close();  The resultant object "XmlData" will contain a list of object of type "Address". We can access the data for the first Address tag as:  XmlData.addressList[0].HouseNo;  XmlData.addressList[0].StreetName;  XmlData.addressList[0].City;  The  XML can be further complicated, let's see the following XML file structure and its class representation:  <?xml version="1.0"?>  <AddressDirectory>  <Owner>Mayank</Owner>  <Age>24</Age>  <Company>BIPL</Company>  <Address>  <HouseNo>1</HouseNo>  <StreetName>Pitampura</StreetName>  <City>Delhi</City>  </Address>  <Address>  <HouseNo>4</HouseNo>  <StreetName>Rohini</StreetName>  <City>Delhi</City>  </Address>  </AddressDirectory>  Here we can see some additional tags inside the "AddressDirectory" like "Owner", "Age", "Company" along with the list of "Address" tags. So the class structure for the XML would be:  public class AddressDirectory  {  public string Owner { get; set; }  public string Age { get; set; }  public string Company { get; set; }  [XmlElement("Address")]  public List<Address> addressList = new List<Address>();  }  public class Address  {  public string HouseNo { get; set; }  public string StreetName { get; set; }  public string City { get; set; }  }  The childnodes of "AddressDirectory" tags are present inside the AddressDirectory class and the childnodes of Address tab are present inside the address class.  Note: What is important to observe here is that the class can contain only those "tag values" which are their  immediate childnodes. I.e., AddressDirectory can only contain the information about their **immediate childnode** like "Owner", "Company", "Age" and "Address". But here "Address" is further containing more tags. The childnodes for "Address" tag cannot be represented by the class "AddresssDirectory". Therefore we require another class for "Address" tag that store the childnode information about "Address" class. The "Address" class will further contain the value of their immediate childnode "HouseNo", "StreetName", "City". Since we have multiple Address tags therefore we have a "List" of "Address" class. XML attributes while deserialization Attributes that can be useful while deserialization are:   1. XmlElement 2. XmlAttribute 3. XmlText   These three attributes provide mapping information. It provides information about which element of the XML tag will be mapped to which variable of the class.  Observe the following XML:  <?xml version="1.0" encoding="utf-8" ?>  <AddressDirectory id="1">  <DirectoryOwner>Mayank</DirectoryOwner>  <PinCode>110085</PinCode>  <Designation place="Delhi">Engineer</Designation>  <Address AddressId="12">  <HouseNo>4</HouseNo>  <StreetName>Rohini</StreetName>  <City>Delhi</City>  </Address>  <Address AddressId="13">  <HouseNo>4</HouseNo>  <StreetName>Rohini</StreetName>  <City>Delhi</City>  </Address>  </AddressDirectory>  Let's observe the different components of this XML File:   1. AddressDirectory is the **root node** of the XML file 2. AddressDirectory contains an "XmlAttribute" as "id" containing value "1" 3. "AddressDirectory" contains "XmlElement" like  DirectoryOwner, Address, Designation, Address 4. "Designation" tab contains an a "XmlAttribute" ("place") and an "XmlText" ("Delhi")   So from the above XML we can figure out what are "XmlElement", "XmlAttribute", "XmlText". While deserializing such complex XML where we can have all the three component we need to explicitly specify whether the class variable store "Element", "Attribute" or "XmlText".  Let's try to desterilize this XML:  public class AddressDirectory  {  **[XmlElement(*"DirectoryOwner"*)]**  public string DirectoryOwner { get; set; }  [XmlElement("PinCode")]  public string PinCode { get; set; }  [XmlElement("Address")]  public List<Address> Address { get; set; }  [XmlElement("Designation")]  public Designation designation { get; set; }  }  Here we have mapped the class variable DirectoryOwner to DirectoryOwner tag of XML file.  Observe here that the class AddressDirectory contains the child node of the AddressAttribute tag. It drills down to the first level only, i.e., it cannot retrieve values about the Attribute of Designation and neither it can fetch information about the **childnodes** of Address tag. Therefore in order to extract these info we need to create another class for Address and Designation. Since we are having multiple Address tags therefore we are having a list of Address class in AddressDirectory.  Let's explore the Address class and the Designation class:  public class Designation  {  **[XmlAttribute(*"place"*)]**  public string place { get; set; }  **[XmlText]**  public string JobType { get; set; }  }  The Designation class here contains two variables, one for storing the innerText and other for storing the place attribute for the Designation tags.  public class Address  {  [**XmlAttribute(*"AddressId"*)**]  public string AddressId { get; set; }  [XmlElement("HouseNo")]  public string HouseNo { get; set; }  [XmlElement("StreetName")]  public string StreetName { get; set; }  [XmlElement("City")]  public string City { get; set; }  }  The Address class further contains a variable that can store the attributes and child node details of Address tags.  The program to be executed in order to deserialize the XML will be:  XmlSerializer deserializer = new XmlSerializer(typeof(AddressDirectory));  TextReader reader = new StreamReader(@"D:\myXml.xml");  object obj = **deserializer.Deserialize(reader);**  AddressDirectory XmlData = (AddressDirectory)obj;  reader.Close();  One more thing that needs to be kept in mind is that, the keywords XmlElement, XmlAttribute, and XmlText are used to map information inside the XML tag to the class variable. The class variable name can be different from that in XML. Example:  [XmlElement("HouseNo")]  public string Number { get; set; }  Here we can see that the XML element HouseNo will be mapped to the class variable Number.  Conclusion:  Deserialization and serialization is a very efficient way to convert the object to XML and vice versa. This save lots of saving time and effort. | | | |
| |  |  |  |  | | --- | --- | --- | --- | | [Checks needed here](http://www.codeproject.com/Messages/4569457/Checks-needed-here.aspx) [Pin](http://www.codeproject.com/Articles/487571/XML-Serialization-and-Deserialization-Part-2)  Markus Ulbricht21-May-13 5:51 | member | [Markus Ulbricht](http://www.codeproject.com/script/Membership/View.aspx?mid=916273) | 21-May-13 5:51 | |
| |  |  |  |  | | --- | --- | --- | --- | | [upvote](http://www.codeproject.com/Articles/487571/XML-Serialization-and-Deserialization-Part-2#xx4569457xx) [downvote](http://www.codeproject.com/Articles/487571/XML-Serialization-and-Deserialization-Part-2#xx4569457xx)  http://codeproject.cachefly.net/script/Forums/Images/t.gif | |  |  | | --- | --- | | |  | | --- | | Since you never know if your received XMLString is valid you should add checks.  And please use XMLTextReader instead of Streamreader.  XmlTextReader xml\_reader = default(XmlTextReader); xml\_reader = new XmlTextReader(File.OpenRead(aFile)); if ((xmlserializer.CanDeserialize(xml\_reader))) {       DeserializedObject = xmlserializer.Deserialize(xml\_reader); }  With the CanDeserialize method you are able to check if the deserialization will work instead of running into an exception. | | | |

# Custom Serialization

Custom Serialization Example

<http://www.codeproject.com/Articles/22787/Custom-Serialization-Example>

An example of implementing custom serialization, how to serialize a collection, and using a File Serialization utility class

* [**Download FileSerializer\_v2.zip - 11.41 KB**](http://www.codeproject.com/KB/cs/SerializeUtility/FileSerializer_v2.zip)

Introduction

Being able to persist your objects to disk and reload them at a later time is actually a very easy task in .NET and I'll show you how to use Custom Serialization to Serialize your objects to disk with a handy File Serializer utility class. I've also provided a sample of how to persist a collection of objects.

Background

This sample code originated from a tutorial at <http://blog.paranoidferret.com/index.php/2007/04/27/csharp-tutorial-serialize-objects-to-a-file/>

and I've decided to extend it a little further and provide you with the ability to download the sample project.

Using the code

The sample project is a Windows Application written in C# using .NET 2.0. When the Form loads, I simply call RunExample() to demonstrate how to serialize an object and a collection of objects.

The RunExample() method is:

public void RunExample()

{

Car car1 = new Car("Ford", "Mustang GT", 2007);

Car car2 = new Car("Dodge", "Viper", 2006);

car1.Owner = new Owner("Rich", "Guy");

car2.Owner = new Owner("Very", "RichGuy");

*//save cars individually*

FileSerializer.Serialize(@"C:\Car1.dat", car1);

FileSerializer.Serialize(@"C:\Car2.dat", car2);

*//save as a collection*

Cars cars = new Cars();

cars.Add(car1);

cars.Add(car2);

FileSerializer.Serialize(@"C:\Cars.dat", cars);

*//now read them back in*

Car savedCar1 = FileSerializer.Deserialize<Car>(@"C:\Car1.dat");

Car savedCar2 = FileSerializer.Deserialize<Car>(@"C:\Car2.dat");

*//and for the collection…*

Cars savedCars = FileSerializer.Deserialize<Cars>(@"C:\Cars.dat");

}

Seems simple enough? Well, it is and I'll show you how.

By the way, the example above is written to be stepped-through using a debugger and examining the variables. You could add some logging if you want to verify that things are working.

The first thing we did was to create a Car...

using System;

using System.Collections.Generic;

using System.Text;

using System.Runtime.Serialization;

using System.Security.Permissions;

namespace MyUtilities

{

[Serializable]

public class Car : ISerializable

{

private readonly string \_make;

private readonly string \_model;

private readonly int \_year;

private Owner \_owner = null; *//this is variable and can change*

private Car() {} *//default ctor not valid - we want to enforce initializing our data*

public Car( string make, string model, int year )

{

\_make = make;

\_model = model;

\_year = year;

}

public Owner Owner

{

get { return \_owner; }

set { \_owner = value; }

}

public string Make

{

get { return \_make; }

}

public string Model

{

get { return \_model; }

}

public int Year

{

get { return \_year; }

}

*//note: this is private to control access; the serializer can still access this constructor*

private Car( SerializationInfo info, StreamingContext ctxt )

{

this.\_make = info.GetString("Make");

this.\_model = info.GetString("Model");

this.\_year = info.GetInt32("Year");

this.\_owner = (Owner)info.GetValue("Owner", typeof(Owner));

}

[SecurityPermission(SecurityAction.LinkDemand, Flags = SecurityPermissionFlag.SerializationFormatter)]

public void GetObjectData( SerializationInfo info, StreamingContext ctxt )

{

info.AddValue("Make", this.\_make);

info.AddValue("Model", this.\_model);

info.AddValue("Year", this.\_year);

info.AddValue("Owner", this.\_owner, typeof(Owner));

}

}

}

You may have noticed that I use a slightly different coding convention to distinguish between private data members by prefixing them with an underscore (\_). I just find this easier to read and stems from old habits Smile | :)

I also write code to enforce how a class is to be used by enforcing when data members should be readonly (not changeable after initialized) and enforcing when the default constructor is not valid by making it private. It is important to note that sometimes the underlying structure needs a default constructor; this is especially true if you want to be able to use XML to serialize your class too. In this case, I make sure to include a comment that the default constructor is for internal use only and does not guarantee proper initialization.

Ok, back to our main topic: Serialization! To begin with, you need to add a "using" statement to reference the System.Runtime.Serialization:

using System.Runtime.Serialization;

I also make it a habit to include security permissions...

using System.Security.Permissions;

The next thing you must do is add the Serializable attribute to the class and implement the ISerializable interface.

[Serializable]

public class Car : ISerializable

We want to implement the ISerializable interface so that we have control over how and what gets serialized; this is refered to as "Custom Serialization". By default, a class must have the Serializable attribute to support serialization; however, if we did not implement the ISerializable interface, then all of our public and private data members are serialized behind the scenes and we would need a default constructor, much like the requirements to use the default XML Serialization. There are several other attributes that we can apply to properties and methods to control serialization; however, I've found the easiest and best way is to provide Custom Serialization, which will also aid in supporting different versions, which I'll discuss shortly.

Taking a look at the ISerializable interface, there is only one method for us to implement:

void GetObjectData( SerializationInfo info, StreamingContext context );

Now, technically, we should declare this virtual in our base class, if we are going to allow other class to derive from it; otherwise, we should mark our class as sealed.

public sealed class Car : ISerializable

...or...

public virtual void GetObjectData( SerializationInfo info, StreamingContext ctxt )

The GetObjectData is called when the class is requested to provide the information (info) that it wants to store. If we are in a class that has derived from another class that supports serialization, then we must call the base's GetObjectData first.

In the Car class example, we have simple data types to store, so we only need to supply two pieces of information to the info.AddValue method; a unique name\identifier and a value.

info.AddValue("Make", this.\_make);

You have to make sure not to duplicate the name, so to avoid this, try not to use common names, like ID, Name, Value, etc. An exception is thrown if a duplicate name is detected. Conflicts like this can occur if you allow your class to be derived from; the name must be unique across the entire object's structure.

If you have a user defined type\class, then you must also supply type information to the AddValue method. The type can be an interface, base type, or concrete type, the main thing to remember here is that what ever type it is, it must support Serialization.

info.AddValue("Owner", this.\_owner, typeof(Owner));

What actually happens here is the Owner class (object) will get it's chance to serialize it's info with us, then the call returns back to us to continue.

Now that we have defined how our class gets serialized, we need to define how it deserializes itself. This is done by providing a special constructor that takes a SerializationInfo and StreamingContext.

private Car( SerializationInfo info, StreamingContext ctxt )

You might have noticed that the constuctor is marked as private! This is because the visibility constraints are ignored during the Deserializtion process; however, if this class is not marked as sealed, then the best practice is to mark visibility as protected. It is not recommend to mark this constructor as public, as this could pose a security risk and exposes the constructor, when in fact it is for internal use.

To Deserialize the info, we simply "Get" the data\values you previously stored; however, we need to specify the unique name we used and we must specify the data type we are expecting. For common data types, we can call GetString, GetInt32, etc. If we have a user\class type, then we must provide type information, as well as having to cast the returned object value to the expected type.

this.\_make = info.GetString("Make");

this.\_owner = (Owner)info.GetValue("Owner", typeof(Owner));

Before moving on, because we are supporting Custom Serialization, we can store other information, such as version information that we could retrieve during the deserialization process and use this information to determine which name\value pair(s) should exist or if a value was stored as a different data type. This is a very powerful way to handle changes you make to the Serialization process, but still want to support older versions.

Hopefully you have a good understanding how to define a class to support Custom Serialization, now on to applying this to a collection.

In this example, there is a Cars collection, which derives from a List of type Car.

[Serializable]

public sealed class Cars : List<Car>, ISerializable

{

public Cars()

: base()

{

}

public Cars( int capacitiy )

: base(capacitiy)

{

}

public Cars( IEnumerable<Car> steps )

: base(steps)

{

}

private Cars( SerializationInfo info, StreamingContext context )

{

int count = info.GetInt32("NumOfCars");

for (int ix = 0; ix < count; ix++)

{

string key = "Car\_" + ix.ToString();

Car car = (Car)info.GetValue(key, typeof(Car));

this.Add(car);

}

}

[SecurityPermission(SecurityAction.LinkDemand, Flags = SecurityPermissionFlag.SerializationFormatter)]

public void GetObjectData( SerializationInfo info, StreamingContext context )

{

info.AddValue("NumOfCars", this.Count);

int ix = 0;

foreach (Car car in this)

{

string key = "Car\_" + ix.ToString();

info.AddValue(key, car, typeof(Car));

ix++;

}

}

}

Now this may not be the perfered way of implementing a collection of Cars, because we can not control access to the List; whereas, we could if we contained it as a data member, but this is a simple example.

One thing I'd like to point out is that we can add Serialization at any layer; the List does not support serialization, so we have to manage serializing it's contents in our derived class. It is also important to point out that Serialization can be broken if a deriving class does not properly support serialization from a base class that does.

If you recall, we can not have duplicate names when storing values, so I've appended an index value to a common name to make it unique. The other piece of information that we need to store is how many items we are storing so we can effiecently get the items during the deserializtion process. Other than that, the process of Adding and Getting values is just like the Car class example.

You can do the same thing to support a Dictionary container; however, there are two methods for storing the "keys" from the Dictionary; one method is to store the "key" value using a unique name with index value (i.e. "ItemKey\_0") and to do the same to store the associated "value" from the dictionary (i.e. "ItemValue\_0"). During the Deserialization process, Get both the ItemKey and ItemValue and put it into your Dictionary. The other method involves storing only the "Value" from the Dictionary, much like the List example. Using this process implies that the "key" information is also contained in the "value" object. During the deserialization process, we get the ItemValue, retrieve it's "key" information and then store it into the dictionary. This can improve performance and reduce the file size if you are duplicating information. Something to think about when you develop classes that need to serialize a dictionary.

Ok, now you are probably wondering how do we take this information and store it as a file on disk and later read it back in... at last, the FileSerializer utility class.

public static class FileSerializer

{

public static void Serialize( string filename, object objectToSerialize )

{

if (objectToSerialize == null)

throw new ArgumentNullException("objectToSerialize cannot be null");

Stream stream = null;

try

{

stream = File.Open(filename, FileMode.Create);

BinaryFormatter bFormatter = new BinaryFormatter();

bFormatter.Serialize(stream, objectToSerialize);

}

finally

{

if (stream != null)

stream.Close();

}

}

public static T Deserialize<T>( string filename )

{

T objectToSerialize = default(T);

Stream stream = null;

try

{

stream = File.Open(filename, FileMode.Open);

BinaryFormatter bFormatter = new BinaryFormatter();

objectToSerialize = (T)bFormatter.Deserialize(stream);

}

catch (Exception err)

{

MessageBox.Show("The application failed to retrieve the inventory - " + err.Message);

}

finally

{

if (stream != null)

stream.Close();

}

return objectToSerialize;

}

}

You will note that this class is marked as static, this means that we do not need to create (i.e. new) a FileSerializer to use it. It's methods are also marked as static. Having a class defined like this is typically refered to as a Utility Class or Helper Class. It's main purpose to to contain some methods that we can call upon throught our code.

In the case of our FileSerializer, we only have two methods:

public static void Serialize( string filename, object objectToSerialize )

public static T Deserialize<T>( string filename )

To Serialize our object to a file, we simply have to provide a filename and the object that we want to serialize. The Serialize method will store the object in a binary string format.

To Deserialize an object, we must supply the type of object we expect. By using Generics, we save ourselves from having to supply type information as an argument and we do not have to cast the return value to the expected type. Basically, you specify "T" as the expected type and the method will do the rest.

Car savedCar1 = FileSerializer.Deserialize<Car>(@"C:\Car1.dat");

Note that you can specify any file extension, I used "dat" in this example.

This example could use additional error checking and handling, but should serve as a good foundation to get you going.

As an extra (not included in the attached source code example) is a utility to serialize/deserialize your object to store in a database)

public static class BinarySerializer

{

/// *<span class="code-SummaryComment"><summary></span>*

/// *Get the serialized data/object in a storable binary string*

/// *<span class="code-SummaryComment"></summary></span>*

/// *<span class="code-SummaryComment"><returns>objectAsBinaryString</returns></span>*

public static string Serialize( object obj )

{

try

{

BinaryFormatter bf = new BinaryFormatter();

MemoryStream ms = new MemoryStream();

*// Serialize the data*

bf.Serialize(ms, obj);

return FromArray(ms.GetBuffer());

}

catch (Exception err)

{

Debug.Print(err.Message);

if (System.Diagnostics.Debugger.IsAttached)

System.Diagnostics.Debugger.Break();

throw err;

}

}

public static T Deserialize<T>( string data )

{

try

{

MemoryStream ms = new MemoryStream(ToArray(data));

BinaryFormatter bf = new BinaryFormatter();

T obj = (T)bf.Deserialize(ms);

return obj;

}

catch (Exception err)

{

Debug.Print(err.Message);

if (Debugger.IsAttached)

Debugger.Break();

throw err;

}

}

public static byte[] ToArray( string data )

{

return Convert.FromBase64String(data);

}

public static string FromArray( byte[] data )

{

return Convert.ToBase64String(data);

}

}

Refer to your database documentation on the data type required to store a large binary data (CLOB\BLOB). All you may need is an additional field to store the name or other information so that you can identify it and retrieve it at a later time.

Additional Food for Thought

Additional food for thought: when you specify the type information, you do not have to specify the most derived type, in fact, you could specify a interface, as long as it is derived from ISerializable. The serialization process knows how to store the complete object, as long as it has implemented serialization properly. All of the types must be known, you can not try to deserialize an object with unknown types.

As an example; if we allowed our Car class to be derived from and we implemented an ICar Interface, which also derived from ISerializable, then we can reference a Car by ICar or Car or the deriving type. Let's say we created a class called Ford that derived from Car; we would need to at the [Serializable] attribute to the Ford class, override the GetObject method (must be made virtual in the Car class) and call the base's implementation first to support Serialize. We would also need to call the base's special constructor from our special constructor to support deserialize (hence why we should make this protected if our class is not sealed). We now can serialize\deserialize by type ICar, Car, or Ford. Pretty cool...this is something that you will find useful if you want to have a collection of base types (i.e. ICar or Car)... the Ford object will work just find.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | [Different type for object to be serialized](http://www.codeproject.com/Messages/2770196/Different-type-for-object-to-be-serialized.aspx) [Pin](http://www.codeproject.com/Articles/22787/Custom-Serialization-Example)  Håkan Andersson17-Oct-08 20:32 | member | [Håkan Andersson](http://www.codeproject.com/script/Membership/View.aspx?mid=4505621) | 17-Oct-08 20:32 | |
| |  |  |  |  | | --- | --- | --- | --- | | [upvote](http://www.codeproject.com/Articles/22787/Custom-Serialization-Example#xx2770196xx) [downvote](http://www.codeproject.com/Articles/22787/Custom-Serialization-Example#xx2770196xx)  http://codeproject.cachefly.net/script/Forums/Images/t.gif | |  |  | | --- | --- | | |  | | --- | | Great article, but wouldn't it be better to use ISerializable as the type for the objectToSerialize parameter? This way only objects that really can be serialized can be passed to the Serialize method. | | | |
| |  |  |  |  | | --- | --- | --- | --- | | [slow](http://www.codeproject.com/Messages/2510138/slow.aspx) [Pin](http://www.codeproject.com/Articles/22787/Custom-Serialization-Example)  Kam16-Apr-08 10:17 | member | [Kam](http://www.codeproject.com/script/Membership/View.aspx?mid=1360306) | 16-Apr-08 10:17 | |
| |  |  |  |  | | --- | --- | --- | --- | | [upvote](http://www.codeproject.com/Articles/22787/Custom-Serialization-Example#xx2510138xx) [downvote](http://www.codeproject.com/Articles/22787/Custom-Serialization-Example#xx2510138xx)  http://codeproject.cachefly.net/script/Forums/Images/t.gif | |  |  | | --- | --- | | |  | | --- | | Good discussion however this method is extremely slow if you are working with large data files. I did a test with 200,000 cars (on a 3.2GHz Quad-Core Intel Xeon) here is what I've got:  Serialization took 6'39" and 51 MB of memory Deserialization took 18'39" and 263 MB of memory  The file size is 7.1MB.  My typical data files are of order 2 million nodes, fully populated with vectors and arrays. I suggest using the Fast Serialization method discussed by <http://www.codeproject.com/KB/cs/FastSerialization.aspx>. | | | |

# A Fast Serialization Technique

<http://www.codeproject.com/Articles/14164/A-Fast-Serialization-Technique>

[**Tim Haynes**](http://www.codeproject.com/script/Membership/View.aspx?mid=893367), 19 May 2006 http://www.codeproject.com/App_Themes/CodeProject/Img/read32.png154.1Khttp://www.codeproject.com/App_Themes/CodeProject/Img/download32.png 2.3Khttp://www.codeproject.com/App_Themes/CodeProject/Img/bookmark32.png 134

Transparently boosting serialization performance and shrinking the serialized object's size.

* [**Download source - 4.62 Kb**](http://www.codeproject.com/KB/cs/FastSerialization/FastSerializationSource.zip)

Introduction

Serialization is everywhere in .NET. Every parameter you pass to or from a remoted object, web service, or WCF service gets serialized at one end and deserialized at the other. So why write about fast serialization? Surely, the standard BinaryFormatter and SoapFormatter are pretty quick, aren't they?

Well, no. When passing a reasonably substantial object from one process to another using Remoting, we find that performance was topping out at 300 calls per second. Investigation showed that each serialization/deserialization cycle was taking 360 microseconds, which would be fine except that 300 per second means that 11% of the CPU is being consumed by the serialization alone!

Background

Some form of custom serialization would be an option. An object knows exactly what types of what fields it wants to serialize. It doesn't need all the general purpose overheads and Reflection to work this out and extract the data - it can do it all by itself, much more efficiently. The result is generally much more compact. There is an example in [.Shoaib's article](http://www.codeproject.com/csharp/CompactSerialization.asp), which demonstrates these benefits.

The problem with custom serialization is that the interface is different, requiring the calling code to be changed. It also doesn't help the automated serialization in .NET's remote access mechanisms, unless you manually serialize to a byte array and then pass this as a parameter. This isn't very type-safe!

What I cover below is a simple way to retain the benefits of custom serialization, while retaining the standard serialization interface and all the benefits that confers.

Using the code

As is often the case in matters of complex serialization, the solution lies in implementing the ISerializable interface (see [here](http://www.codeproject.com/csharp/objserial.asp) for a primer). Here's a much simplified version of the object we are using:

[Serializable]

public class TestObject : ISerializable {

public long id1;

public long id2;

public long id3;

public string s1;

public string s2;

public string s3;

public string s4;

public DateTime dt1;

public DateTime dt2;

public bool b1;

public bool b2;

public bool b3;

public byte e1;

public IDictionary<string,object> d1;

}

To serialize an object, ISerializable requires us to implement GetObjectData to define the set of data to be serialized. The trick here is to use custom serialization to merge all the fields into a single buffer, then to add this buffer to the SerializationInfo parameter to be serialized by the standard formatters. This is how it's done:

*// Serialize the object. Write each field to the SerializationWriter*

*// then add this to the SerializationInfo parameter*

public void GetObjectData (SerializationInfo info, StreamingContext ctxt) {

SerializationWriter sw = SerializationWriter.GetWriter ();

sw.Write (id1);

sw.Write (id2);

sw.Write (id3);

sw.Write (s1);

sw.Write (s2);

sw.Write (s3);

sw.Write (s4);

sw.Write (dt1);

sw.Write (dt2);

sw.Write (b1);

sw.Write (b2);

sw.Write (b3);

sw.Write (e1);

sw.Write<string,object> (d1);

sw.AddToInfo (info);

}

The SerializationWriter class extends BinaryWriter to add support for additional data types (DateTime and Dictionary) and to simplify the interface to SerializationInfo. It also overrides BinaryWriter's Write(string) method to allow for null strings. I won't go into the implementation detail here. There is lots of explanation in the code for those who are interested.

ISerializable also requires us to define a constructor to deserialize a stream to a new object. The process here is just as simple as that above:

*// Deserialization constructor. Create a SerializationReader from*

*// the SerializationInfo then extract each field from it in turn.*

public TestObject (SerializationInfo info, StreamingContext ctxt) {

SerializationReader sr = SerializationReader.GetReader (info);

id1 = sr.ReadInt64 ();

id2 = sr.ReadInt64 ();

id3 = sr.ReadInt64 ();

s1 = sr.ReadString ();

s2 = sr.ReadString ();

s3 = sr.ReadString ();

s4 = sr.ReadString ();

dt1 = sr.ReadDateTime ();

dt2 = sr.ReadDateTime ();

b1 = sr.ReadBoolean ();

b2 = sr.ReadBoolean ();

b3 = sr.ReadBoolean ();

e1 = sr.ReadByte ();

d1 = sr.ReadDictionary<string,object> ();

}

Similarly, SerializationReader extends BinaryReader for the same reasons as above.

Over time, I'll probably be extending the set of types which the writer and reader can handle efficiently. There are already the WriteObject() and ReadObject() methods which will write any arbitrary type, but this just falls back to standard binary serialization (unless it's one of the supported fast types).

Results

The test program included in the download simply creates and populates the TestObject, and times its serialization and deserialization, in microseconds per cycle, averaged over 250K cycles. All timings are done on a 1.5GHz Pentium M laptop. The results are:

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Formatter** | **Size (bytes)** | **Time (uS)** |
| Standard serialization | Binary | 2080 | 364 |
| Fast serialization | Binary | 421 | 74 |
| Fast serialization | SOAP | 1086 | 308 |

So, the fast serialization technique below can cut both the size and serialization-deserialization time to about a fifth of the out-of-the box serialization. Even SOAP serialization (normally 2 to 3 times slower than binary) is faster than the standard binary serialization.

Summary

Combining custom serialization with ISerializable in this way delivers major performance gains without any change to the handling of the objects in question. It allows fast serialization to be transparently added to specific objects where a performance issue has been identified.

In our own case, throughput increased from 300 Remoting calls per second to over 700, just by changing this for one key object. No other changes were necessary.

There is also one other unexpected benefit from this. You'll notice that there are no comparative figures above for the SoapFormatter, which is because MS has not equipped the SoapFormatter to handle generic types. Using the technique above means that the SoapFormatter never sees the generic type which has been custom serialized to a byte array, so this restriction is removed.

Combining custom serialization with ISerializable is never going to be as fast as pure custom serialization alone. However, the added benefit of remaining within the standard serialization framework makes this a useful technique for boosting performance without impacting other code.

History

* First version - 19 May 2005.

This is my first post on CodeProject - so please be gentle!

# Complex Object serialization

<http://www.codeproject.com/Articles/30270/XML-Serialization-of-Complex-NET-Objects>