# Serialization

## What is Serialization?

Serialization is nothing but storing of objects in files (or any other stream).

### Why do we need it? Why can we not store objects in a database?

Serialization is a quick and efficient way of storing objects. There is no need to establish a connection to a database for such comparatively trivial objects.

These objects can then be acted upon by the same or other application(s).

The files can be copied by any application to any other machine with ease. It is comparatively simple and fast than getting the data by establishing a connection with the database. The database approach will still involve either writing the `For XML` SQL or recreating the objects from the field values of each row.

A decision to store objects on the database may involve creation of appropriate schemas. Any change will be even more maintenance intensive. With serialization, the schema need not be decided in advance and schemas can be created, edited and discovered with simple program structures.

## What do we need to do in order to serialize objects?

The definition *Serialization is nothing but storing of objects in files* tells us that all we need is:

1. An Object to be serialized.
2. A FileStream to the file where the object will be stored (any other stream will work equally well).
3. An instance of a class which will do the action of storing.

4. Storing.

In terms of code, this is how we do it:

Step 1 involves 3 sub steps:

a) Mark the class of the object to be serialized, with the `[Serializable]` attribute.

b) Create a parameterless constructor of this class.

c) Create the object.

Step 2 is a simple file handling mechanism.

Step 3 involves creating an instance of a class called `BinaryFormatter`, say `Formatter`.

Step 4 involves calling the `Serialize` method of the `Formatter`.

# Sample code

### Class:

```
// Step 1a
[Serializable]
public class ClsSerialize
{
// Step 1b – optional if there is no other constructor
    public ClsSerialize() { }
    public string sName;
    public int iDependents;
    private int iAge;
    public int Age
    {
        get{return this.iAge;}
        set{this.iAge = value;}
    }
}
```

### Namespace needed by the client code:

```
// Namespace containing the BinaryFormatter
using System.Runtime.Serialization.Formatters.Binary;
```

### Client code:

```
// Step 1c
ClsSerialize oSerIn = new ClsSerialize();
oSerIn.sName = "Aaron";
oSerIn.iDependents = 3;
oSerIn.Age = 32;
// Step 2
string sFile = @"C:/Temp/" + oSerIn.sName + ".bin";
System.IO.FileStream fsOut;
try{
```

```
    fsOut = System.IO.File.OpenWrite(sFile);
}
catch{ return; }
// Step 3
BinaryFormatter Formatter = new BinaryFormatter();
// Step 4
try{
    Formatter.Serialize(fsOut, oSerIn);
}
finally{ fsOut.Close(); }
```

**Note:** The output file stream parameter comes before the input object parameter.

## Serialized Output

The serialized output of the above code looks like this.

```
   ÿÿÿÿ       JWinAppSerialization, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=null
WinAppSerialization.ClsSerialize————————————————————————————   sName
iDependents
————————————————————————————————————————————————————————————————————
iAge      ——————————————————————————
Aaron————————————————————————————
```

As we can see, all the public and private fields show up in the output. Most of the junk has been replaced by dashes.

## How about getting the objects back from these files?

The first three steps remain almost the same (just keep in mind that now, we are reading/retrieving instead of writing/storing). In the fourth step, instead of calling the `Serialize` method, we call the `Deserialize` method, which creates the object for us.

This is how the client code for deserializing will look like:

```
// Step 2
string sFile = @"C:/Temp/Aaron.bin";
System.IO.FileStream fsIn;
try{
    fsIn = System.IO.File.OpenRead(sFile);
}
catch { return; }
// Step 3
BinaryFormatter Formatter = new BinaryFormatter();
// Step 4
ClsSerialize oSerOut;
try{
  oSerOut = (ClsSerialize)Formatter.Deserialize(fsIn);
```

```
}
finally { fsIn.Close(); }
```

See the convenience: no hassle of identifying the table schema and then creating the object accordingly. In other words, the class knows how to get itself back. Programmatically doing this with a class will involve a sophisticated helper class. Microsoft has already done this for us!

## Soap Formatting

If we want to make the serialized output human readable, pass through firewalls and be used universally by applications, we should use the SoapFormatter class instead of the BinaryFormatter. As can be expected, this class is in the `System.Runtime.Serialization.Formatters.Soap` namespace.

**Note:** The mscorlib.dll is loaded by the Visual Studio when a project is created. This assembly contains the `System.Runtime.Serialization.Formatters.Binary` namespace, which contains the `BinaryFormatter` class. In order to work with the SoapFormatter class, we need to add a reference to the System.Runtime.Serialization.Formatters.Soap assembly which contains a namespace of the same name. This namespace, in turn, contains the `SoapFormatter` class.

# Advanced Scenarios

### Scenario 1: Prevent serialization of a field

Let's say, we have a password or a Session ID field which is retrieved from a database or a server when the object is created, and then used for the lifetime of the object. We do not want to store this field in the serialized file.

In this case, we mark it with the `[NonSerialized]` attribute.

### Scenario 2: Serializing an array or any collection

All the Microsoft classes are Serializable. This includes the array and all the collections. So, nothing more than the 4 basic steps of serialization needs to be done.

### Scenario 3: A field of a user-defined type

If a field of the class to be serialized is of a user-defined type, then, at the time of serialization, an error will be thrown. This error is the same as the one that the runtime will throw when we try to serialize an object when its class is not marked with the `[Serializable]` attribute or its class does not have a parameterless constructor. Basically, the runtime simply cascades down the object and its constituent objects to serialize the entire object. If any object in this graph is not serializable, an error is thrown.

As you might have guessed, all that one has to do in this case is to

1. Mark the class of the field with the `[Serializable]` attribute.
2. Create a *parameterless constructor* of this class.

### Scenario 4: An enum field

Since underlying type of `enum` is `int`, nothing needs to be done.

### Scenario 5: Polymorphism

An object accessed as its parent class instance and then serialized still needs to have its class marked `[Serializable]` and must have a *parameterless constructor*. The process of serialization simply looks at the *serializability* of the object and does not follow the inheritance rule.

**Tip:** If you try combining a few scenarios, you might figure out new exam questions yourself!! Don't hesitate in doing so. It will be a very simple exercise. A few such scenarios are given below.

### Scenario 6: Serializing an array or collection of a user-defined type

This scenario is a combination of Scenarios 2 and 3. So, the only thing to be done in this case is to mark the class of which the object is the array as `[Serializable]` and make a *parameterless constructor* for the same.

### Scenario 7: An array field

Almost like Scenario 2 – therefore – do nothing special.

### Scenario 8: A field is an array of user-defined elements

Almost like Scenario 6 – therefore - mark the class of which the field is the array as `[Serializable]` and make a *parameterless constructor* for the same.

### Scenario 9: Polymorphism over field

Combination of Scenarios 3 and 5 – child class should be marked as `[Serializable]` and have a *parameterless constructor*.

### Scenario 10: Polymorphism over an array field

Combination of Scenarios 5 and 7 – child class should be marked as `[Serializable]` and have a *parameterless constructor*.

# Custom Serialization

## What constitutes custom serialization?

We might need to customize Serialization for any of the following reasons:

1. To avoid marking fields with serialization attribute(s). In other words, to move from declarative customization to programmatic customization.
2. To avoid having to make classes of constituent elements or their subclasses serializable.
3. To format the serialized output in any way we want.
4. To customize serialization based on user input.
5. To customize serialization based on the streaming context.
6. To perform any action before or after serialization.

## Method 1: Complete control over serialization

There is a way to completely control the output of the serialization, and thereby cover the first five aspects of customization. This is done by making the `[Serializable]` class implement the `ISerializable` interface. As can be expected, this interface entails:

1. Implementing the method `GetObjectData` which overrides the serialization performed by a formatter.
2. Implementing a special constructor which overrides the deserialization performed by a formatter.

**Note:** The class implementing the ISerializable interface should still be marked as [Serializable]. Otherwise, the BinaryFormatter or the SoapFormatter will not serialize its instance.

Let's make a few changes to our earlier code, to get more control over the serialization. Also, this time, let's use the `SoapFormatter` since the output in the Soap format will be easier to analyze.

## Sample code

### Namespace needed by the class:

```
//Namespace containing SerializationInfo and StreamingContext
using System.Runtime.Serialization;
```

### Class:

```
// Step 1a
[Serializable]
public class ClsSerialize:ISerializable
{
// Step 1b – special constructor this time
    public ClsSerialize() { }
    public ClsSerialize(
        SerializationInfo info, StreamingContext context
    ){ // implementation discussed later }
// Step 5 – new step
    public void GetObjectData(
        SerializationInfo info, StreamingContext context
    ){
        // a) Output the data in any format
        info.AddValue("Name", this.sName);
        info.AddValue(
            "NumberOfDependents", this.iDependents
        );
        // b) Make use of the client input
        object oUserData = context.Context;
        if (oUserData != null){
            info.AddValue("CompanyName", (string)oUserData);
        }
        // c) Make use of the StreamingContextStates sent by the client
        if(
            (
                ( (int)context.State ) &
                ( (int)StreamingContextStates.CrossMachine )
             ) != 0
        ){
          info.AddValue("Machine", Environment.MachineName);
        }
    }
    // The rest of the class implementation remains the same
}
```

### Include reference and namespaces needed by the client code:

```
// Add a reference to the
//    "System.Runtime.Serialization.Formatters.Soap.dll"
// Namespace containing the SoapFormatter
```

```
using System.Runtime.Serialization.Formatters.Soap;
//Namespace containing SerializationInfo and StreamingContext
using System.Runtime.Serialization;
```

### Client code:

```
// Steps 1c
// Initialize the object to be serialized - same as before
// Step 2
// Open the file stream - same as before
// Step 3
SoapFormatter Formatter = new SoapFormatter();
// Step 3b – make use of customized Serialization
//   Pass
//     (i) StreamingContextStates
//     (ii) Any object that GetObjectData() can process
StreamingContextStates stt =
    StreamingContextStates.Persistence | StreamingContextStates.Other;
Formatter.Context = new StreamingContext(stt, "NewIdea Inc");
// Step 4
// Serialize – same as before
```

## Code analysis

### Class

1. The two additions to the class, as necessitated by the `ISerializable` interface, are the `GetObjectData` function for serialization and a new constructor for deserialization (explained in the *Custom Deserialization* section).
2. A complete control is demonstrated by the `GetObjectData` function implementation.

a) We pass those fields to the output that we want.

b) The names of these fields can be anything that we want.

c) If we have a field of user-defined type, then we do not have to take care of its polymorphism since output will only be read off the fields / properties / functions of the superclass (the subclass is not known to the serializable class). If we pass a field of user-defined type directly to the `info.AddValue`, then the various scenarios of arrays and polymorphism remain in effect.

d) Polymorphism related to serializing an array follows simple polymorphism principles.

In other words, all the advanced scenarios, discussed earlier, can be easily covered with this level of control. All this is done simply by using the `AddValue` method of the `SerializationInfo` parameter.

3. This mechanism gives us two additional levels of control, by providing the `StreamingContext` parameter.

a) Using the `Context` property of this parameter, we can allow the `Formatter` on the client side to pass an object (`State`) to the serialization process, which we can use. In our sample code, we are allowing the client to send a string, which we are passing onto the serialization output ("NewIdea Inc"). We can allow a `State` of any type and create a sophisticated code on the basis of this `State`.

b) The `Formatter` can specify a combination of its `StreamingContextStates` and we can base our output on that, as we have done in step c of the `GetObjectData()`.

### Client

The `Formatter` can send the `StreamingContextStates` and its `State` – both of which can be used by `GetObjectData`. The simplest way to accomplish this is to instantiate a `StreamingContext` object from these two parameters and then assigning this instance to the `Context` property of the `Formatter`. See step 3b of the client code.

The next table shows all the available `StreamingContextStates`. You should use them to make complete use of Customized Serialization.

**Flags of the `StreamingContextStates` enumeration**

| Member | Description |
|---|---|
| **All** | The serialized data can be transmitted to or received from any of the other contexts. |
| **Clone** | The object graph is being cloned. |
| **CrossAppDomain** | The source or destination context is a different AppDomain. |
| **CrossMachine** | The source or destination context is a different computer. |
| **CrossProcess** | The source or destination context is a different process on the same computer. |
| **File** | The source or destination context is a file. |
| **Other** | The serialization context is unknown. |
| **Persistence** | The source or destination context is a persisted store - a database, a file, etc. |
| **Remoting** | The data is remoted to a context in an unknown location. |

## Serialized Output

The serialized output of the above code looks like the following. Verify if this is what you expected.

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
<a1:ClsSerialize id="ref-1"
xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Proj/Proj%2C%20Version
%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
<Name id="ref-3">Aaron</Name>
<NumberOfDependents>3</NumberOfDependents>
<CompanyName id="ref-4">NewIdea Inc</CompanyName>
</a1:ClsISerialize>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

# Method 2: Perform actions before or after serialization

We may want to log somewhere that we are going to serialize an object, or that we have serialized an object. In this case, we just need to mark the functions that we want to run before serializing takes place with the `[OnSerializing]` attribute and the functions that we want to run after the serialization finishes with the `[OnSerialized]` attribute.

Like all eventhandlers, these functions must conform to a signature, which is:

```
void Func(StreamingContext context)
```

The `context` argument is the same as the one used in the `GetObjectData` function. Having learnt about its strength, use it with full ingenuity.

Here are a few interesting flexibility scenarios allowed with regards to these attributes:

1. They can be applied to any number of functions.
2. The same function can have both the attributes applied to it, as also the `[OnDeserializing]` and `[OnDeserialized]`, which are explained in the *Custom Deserialization* section.
3. If an attribute is applied to the parent class, then that function runs before the corresponding function of the child class.

**Note:** The `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]`,

`[OnDeserialized]` attributes only work for a BinaryFormatter and not for a SoapFormatter.

# Custom Deserialization

If we have customized our serialization, we will have to mirror it in our deserialization as well.

There are 3 methods of accomplishing this.

## Method 1: Special Constructor

The special constructor left out in the *Custom Serialization* section is implemented here. There will be no change in the client code from the 4 basic steps of deserialization (with a proper `Formatter` type).

### Sample code

#### Class:

```
// Step 1a
[Serializable]
public class ClsSerialize:ISerializable
{
// Step 1b – special constructor this time
    public ClsSerialize() { }
    public ClsSerialize(
        SerializationInfo info, StreamingContext context
    ){
        // (i) Get data from the serialized file
        this.sName = info.GetString("Name");
        this.iDependents = info.GetInt32 ("NumberOfDependents");
        // (ii) Fill up the data which is not expected from the
        //        serialized file
        this.iAge = this.GetAgeFromDatabase(this.sName);
        // (iii) Use the extra input given by the serialization client
        try{
            string sCompany = info.GetString("CompanyName");
            // Log somewhere the name of the company which sent this
            //   serialized file
        }
        catch (Exception ex) { }
        try{
            string sMachine = info.GetString("Machine");
            // Log somewhere that the serialized file came from some
            //   other machine
        }
        catch (Exception ex) { }
    }

    // Helper function
```

```
public int GetAgeFromDatabase(string PersonName){
    // Get age for this person from database
}
```

Note that the above code is simply a mirror image of the code in `GetObjectData()`. Simply by using the `GetString()`, `GetInt32()` and other such functions of the `SerializationInfo` parameter, the fields of the instance are created.

## Advanced Scenarios

### Scenario 1: A field which was not serialized

Populate that field, as you would normally do – just as we have done for `iAge`.

### Scenario2: Deserializing an array or any collection

As stated for serialization, nothing special to be done.

### Scenario 3: An enum field

Since underlying type of `enum` is `int`, just use the `GetInt32()` method.

### Scenario 4: A field of a user-defined type

Use the `GetValue(string, Type)` function. The first parameter is the name of the tag in the serialized file. The second parameter is the type that we are expecting. It returns an instance of `object` type. This instance can be cast to the type which we are expecting.

### Scenarios 5, 6: An array field, polymorphism

Same action as for Scenario 4.

> **Tip:** If you try to combine a few scenarios, as you did in the *Advanced Scenarios* section of serialization, you might figure out new exam questions yourself! From the Scenarios 4, 5 and 6, it might have become clear that all these combinations will require the same action as explained for Scenario 4.

# Method 2: Perform actions before or after deserialization

This method is same as described in the *Perform actions before or after serialization* section, including the note following it. Just replace `[OnSerializing]` with `[OnDeserializing]` and `[OnSerialized]` with `[OnDeserialized]`, and vice-versa.

## Method 3: Perform actions after deserialization

Remember, the `[OnDeserializing]` and `[OnDeserialized]` attributes work only for BinaryFormatter and not for a SoapFormatter. In any case, for simple actions that a `Formatter` would like to perform after deserialization, this method is suitable. All that we have to do is make the class inherit the `IDeserializationCallback` interface and implement its `OnDeserialization` method. The signature of this method is:

```
void OnDeserialization( Object sender );
```

**Tip:** The `IDeserializationCallback.OnDeserialization` implementation executes before the function(s) marked with the `[OnDeserialized]` attribute

# XML Serialization

XML Serialization gives the serialized output in the XML format. The advantages are the same as for the Soap formatted serialization: human-readability, passing through firewalls and universal applicability.

There is no difference between the 4 basic steps of Binary/Soap Serialization/Deserialization and XML Serialization/Deserialization. There are 2 minor modifications in the implementation, though.

a) Step 1a) is modified to:
   Make the class of the object to be serialized, `public`.

b) Step 3 is modified to:
   Create an instance of a class called `XmlSerializer`, say `Formatter`, passing the type of the instance to be parameterized, as a constructor argument.

## Sample code

### Class:

```
// Step 1a
public class ClsXmlSerialize
{
// Step 1b – optional if there is no other constructor
    public ClsSerialize() { }
    public string sName;
    public int iDependents;
    private int iAge;
    public int Age
    {
        get{return this.iAge;}
        set{this.iAge = value;}
```

```
    }
}
```

### Namespace needed by the client code:

```
// Namespace containing the XmlSerializer
using System.Xml.Serialization;
```

### Client code for serialization:

```
// Step 1c
ClsXmlSerialize oSerIn = new ClsXmlSerialize();
oSerIn.sName = "Eugene";
oSerIn.iDependents = 1;
oSerIn.Age = 23;
// Step 2
string sFile = @"C:/Temp/" + oSerIn.sName + ".bin";
System.IO.FileStream fsOut;
try{
    fsOut = System.IO.File.OpenWrite(sFile);
}
catch{ return; }
// Step 3
XmlSerializer Formatter = new XmlSerializer( typeof(ClsXmlSerialize) );
// Step 4
try{
    Formatter.Serialize(fsOut, oSerIn);
}
finally{ fsOut.Close(); }
```

# Serialized Output

```
<?xml version="1.0"?>
<ClsXmlSerialize xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <sName>Eugene</sName>
  <iDependents>1</iDependents>
  <Age>23</Age>
</ClsXmlSerialize>
```

`iAge` did not appear in the serialized output since it is a private field.

**Note:** Public properties, like `Age` in our example, appear in the XML serialized output.

### Client code for deserialization:

```
// Step 2
string sFile = @"C:/Temp/Eugene.xml";
System.IO.FileStream fsIn;
try{
    fsIn = System.IO.File.OpenRead(sFile);
}
catch { return; }
// Step 3
XmlSerializer Formatter = new XmlSerializer( typeof(ClsXmlSerialize) );
```

```
// Step 4
ClsSerialize oSerOut;
try{
  oSerOut = (ClsXmlSerialize)Formatter.Deserialize(fsIn);
}
finally { fsIn.Close(); }
```

So, as you can see, with two minor changes: one in marking the class and other in instantiating the `Formatter`, we take our knowledge to XML Serialization.

# Advanced Scenarios in XML Serialization

### Scenario 1: Prevent serialization of a field/property

Mark it with the `[XmlIgnore]` attribute.

### Scenario 2: Serializing array or any collection

Remember, unlike the `BinaryFormatter` or the `SoapFormatter`, the `XmlSerializer` is created on the basis of the class, the instance of which is to be serialized. Now, we have to serialize an instance of a class array. So, just replace the class with the array of the class in the instantiation of the `XmlSerializer` and you will be done!

Observe that *there is no change to the pattern of serialization*.

The output will show thus:

```
<ArrayOfCls>
  <Cls>…..</ Cls >
  < Cls >…..</ Cls >
< ArrayOfCls >
```

### Scenario 3: A field/property of a user-defined type

For the same reason as for Scenario 2, at the time of creation of the `XmlSerializer`, the classes of all the fields/properties are also marked and compiled for serialization. So, we have to do nothing special.

### Scenario 4: An enum field

Nothing special needs to be done.

### Scenario 5: Polymorphism

Mark the parent class with the `XmlInclude` attribute as:

```
[XmlInclude( typeof(ChildClass) )]
```

This will allow the `XmlSerializer` to include the definition of the child class in its fold, enabling it to serialize its instance. In other words,

Serialization does not follow the inheritance rule. It has to be specified explicitly that the children are ready to be serialized.

**Note:** As you will see from Scenarios 9 and 10, this is not a good approach. Try making the `XmlSerializer` for the class which you expect to Serialize.

**Tip:** You should try combining a few scenarios. Some of them are given below.

### Scenarios 6, 7, 8: Serializing an array or collection of a user-defined type, an array field, or when a field is an array of user-defined elements

All these scenarios will yield the same result, and that is: *stick with the 4-step process*.

### Scenario 9: Polymorphism over field

Combination of Scenarios 3 and 5 – parent class should be marked with an `[XmlInclude( typeof(ChildClass) )]` attribute.

### Scenario 9 again: Polymorphism over field

With the `XmlInclude` approach, the burden, of making the child class available for serialization as the parent class, lies with the parent. This may not always be possible. Most of the time, the classes are not even your own. So, the alternative is to shift that burden to the field. This is done by marking the field with the `[XmlElement]` attribute as:

```
[XmlElement( Type = typeof(ParentClass) )]
[XmlElement( Type = typeof(ChildClass) )]
```

This approach also gives us the ability to name the XML element whatever we want, based on its type. For example, if we use the following field in the class, the serialized output will show an `oField` element when the instance is of the `ParentClass` type and a `ChildField` element when the instance is of the `ChildClass` type.

```
[XmlElement( Type = typeof(ParentClass) )]
[XmlElement( Type = typeof(ChildClass), ElementName = "ChildField" )]
public ParentClass oField;
```

**Note:** We also need to specify the `XmlElement` attribute for the parent class when we shift the burden to the field.

### Scenario 10: Polymorphism over an array field

Combination of Scenarios 5 and 7 – parent class should be marked with an `[XmlInclude(typeof(ChildClass) )]` attribute.

### Scenario 10 again: Polymorphism over an array field

As in the last scenario, putting the burden, of enumerating the child classes, on the parent class is not advisable. The alternative we have in this case is the `XmlArrayItem` attribute, which is applied in the same way as `XmlElement`:

```
[XmlArrayItem ( Type = typeof(ParentClass) )]
[XmlArrayItem ( Type = typeof(ChildClass), ElementName = "ChildField" )]
public ParentClass[] ArrField;
```

In the above example, using the *optional* `ElementName` property, an array element will appear in the XML output as `ParentClass` when the instance is of the `ParentClass` type and as `ChildField` when the instance is of the `ChildClass` type.

Just as for the `XmlElement` attribute, we have to specify the `XmlArrayItem` attribute for the parent class as well.

**Note:** In all the declarative approaches to handle polymorphism, the additional child fields do appear in the serialized output.

# Custom XML Serialization

## Method 1: Declarative

We can achieve simple customization using the attributes themselves. We have discussed the `XmlIgnore`, `XmlInclude`, `XmlElement` and `XmlArrayItem` already. Other salient attributes and their prominent properties are discussed here. To learn more easily, let's distribute these attributes into 4 groups:

| **Group 1: Attributes over classes** | | |
|---|---|---|
| **Attribute** | **Property** | **Use** |
| **XmlInclude** | Type | Allow polymorphism |
| **XmlRoot** | *Works only if the instance is the only element to be serialized* | |
| | ElementName | Customize the name of the element for this class |
| | IsNullable | Specify `xsi:null` in the serialized output if instance is `null` |
| | Namespace | Specify the XML namespace of the class |
| **XmlType** | Namespace | Specify the XML namespace of the class |
| | TypeName | Customize the name of the element for this class |

| Group 2: Attributes over members of an enum | | |
|---|---|---|
| **Attribute** | **Property** | **Use** |
| **XmlEnum** | Name | Customize the name of the value to show when a field is of this enum type and has this particular enum value. |

| Group 3: Attributes over normal members | | |
|---|---|---|
| **Attribute** | **Property** | **Use** |
| **XmlIgnore** | Ignore the field from serialization | |
| **XmlAttribute** | *Specify a field/property to show up as an attribute in the serialized output.* | |
| | AttributeName | Customize the name of the attribute |
| | Namespace | Specify the XML namespace of the class of the field |
| **XmlElement** | *Specify a field/property to show up as an element in the serialized output* | |
| | ElementName | Customize the name of the element |
| | IsNullable | Specify `xsi:null` in the serialized output if instance is `null` |
| | Namespace | Specify the XML namespace of the class of the field |
| | Type | Allow polymorphism |
| **XmlText** | *Specify a field/property to show up as text (neither element nor attribute) in the serialized output.* | |
| | Type | Allow polymorphism |

| Group 4: Attributes over members which are/return arrays | | |
|---|---|---|
| **Attribute** | **Property** | **Use** |
| **XmlArray** | *Specify the XML output for the array* | |
| | ElementName | Customize the name of the element |
| | IsNullable | Specify `xsi:null` in the serialized output if instance is `null` |
| | Namespace | Specify the XML namespace of the array class |
| **XmlArrayItem** | *Specify the XML output for an array element* | |
| | ElementName | Customize the name of the element |
| | IsNullable | Specify `xsi:null` in the serialized output if instance is `null` |

| | | |
|---|---|---|
| | Namespace | Specify the XML namespace of the class of the array element |
| | Type | Allow polymorphism |

**Tips:**

1. Only one `XmlAttribute` attribute can be used for one field.
2. `XmlText` attribute can be applied to only one member in a class; otherwise you will get a `System.InvalidOperationException` at runtime.
3. Put the field with the `XmlText` attribute at the very end of the class implementation. The indentation of the XML output tends to be lost after the output of such a field.

Let's use these elements in the following sample code.

## Sample code

### Namespace needed by the classes and the client code:

```
// Namespace containing the Xml attributes and the XmlSerializer
using System.Xml.Serialization;
```

### Classes:

```
[XmlType(TypeName="House"), XmlInclude(typeof(ClsApt))]
public class ClsXmlHouseAttribs {
    [XmlElement(ElementName = "objNumber", Type = typeof(object))]
    [XmlElement(ElementName = "strNumber", Type = typeof(string))]
    [XmlElement(ElementName = "Number",    Type = typeof(int))   ]
    public object sNumber;

                                              public string sStreet;
    [XmlIgnore]                               public string sState;
    [XmlAttribute(AttributeName = "ZipCode")] public string sZip;

    [XmlArray(ElementName="People")]
    [XmlArrayItem(Type = typeof(object))]
    [XmlArrayItem(ElementName = "strOccupant", Type = typeof(string))]
    [XmlArrayItem(ElementName = "OccupyingPerson",
                         Type = typeof(ClsPerson))]
    public object[] Occupants;

    [XmlText] public string sCity;
}

public class ClsApt : ClsXmlHouseAttribs { public string sOwner; }

public class ClsPerson { public AgeGroup AgeGrp; }

public enum AgeGroup {
    Below18, Over65, [XmlEnum(Name="EarningMember")] Between18And65
}
```

### Client code:

```
// Step 1c
ClsXmlHouseAttribs oSerIn = new ClsXmlHouseAttribs();
oSerIn.sNumber = "24-E";    oSerIn.sStreet = "Washington St";
oSerIn.sCity = "New York";  oSerIn.sState = "NY";  oSerIn.sZip = "10003";
// Array field with elements of different types
oSerIn.Occupants = new object[] { "Harry", new object() };

// Class to be serialized is child of the class expected to be serialized
ClsApt oSerIn2 = new ClsApt();
oSerIn2.sNumber = 241;              oSerIn2.sStreet = "W St";
oSerIn2.sCity = "New York";         oSerIn2.sState = "NY";
oSerIn2.sZip = "10001";             oSerIn2.sOwner = "Bleem Inc";

// Testing the use of [XmlEnum] Attribute
ClsPerson P1 = new ClsPerson();     P1.AgeGrp = AgeGroup.Between18And65;
ClsPerson P2 = new ClsPerson();     P2.AgeGrp = AgeGroup.Over65;
oSerIn2.Occupants = new ClsPerson[] { P1, P2 };

ClsXmlHouseAttribs[] ArrHouses =
    new ClsXmlHouseAttribs[] { oSerIn, oSerIn2 };

// Step 2
string sFile = @"C:/Temp/ArrHouses.xml";
System.IO.FileStream fsOut;
try{
    fsOut = System.IO.File.OpenWrite(sFile);
}
catch { return; }
// Step 3 – XmlSerializer for an array
XmlSerializer Formatter =
    new XmlSerializer( typeof(ClsXmlHouseAttribs[]) );
// Step 4
try{
    Formatter.Serialize(fsOut, ArrHouses);
}
finally { fsOut.Close(); }
```

## Serialized Output

Here is the serialized output of the above code. Verify if this is what you expected.

```
<?xml version="1.0"?>
<ArrayOfHouse xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <House ZipCode="10003">
    <strNumber>24-E</strNumber>
    <sStreet>Washington St</sStreet>
    <People>
      <strOccupant>Harry</strOccupant>
      <anyType />
    </People>New York</House>
  <House xsi:type="ClsApt" ZipCode="10001">
    <Number>241</Number>
```

```
    <sStreet>W St</sStreet>
    <People>
      <OccupyingPerson>
        <AgeGrp>EarningMember</AgeGrp>
      </OccupyingPerson>
      <OccupyingPerson>
        <AgeGrp>Over65</AgeGrp>
      </OccupyingPerson>
    </People>New York<sOwner>Bleem Inc</sOwner></House>
</ArrayOfHouse>
```

# Method 2: Programmatic

To achieve even more control over the XML serialization, make the class implement the `IXmlSerializable` interface. This will entail implementing 3 functions: `WriteXml`, `ReadXml` and `GetSchema`.

## Sample code:

### Namespaces needed by the class and the client code:

```
// Namespace containing IXmlSerializer
using System.Xml.Serialization;
// Namespace containing XmlWriter and XmlReader
using System.Xml;
```

### Modified class:

```
public class ClsXmlHouseAttribs : IXmlSerializable {
    // Same fields as before

    // Functions required by the IXmlSerializable interface
    public void WriteXml(XmlWriter writer) {
        writer.WriteAttributeString("ZipCode", this.sZip);
        if (this.sNumber is int) {
           writer.WriteElementString ("Number", this.sNumber.ToString());
        }
        else {
           writer.WriteElementString("sNumber", this.sNumber.ToString());
        }
        writer.WriteString(this.sCity);
    }

    public void ReadXml(XmlReader reader)
    { // implementation discussed later }

    public System.Xml.Schema.XmlSchema GetSchema() { return null; }
 }
```

### Client code:

```
// Step 1c
// Initialize the object to be serialized – same as before
// Step 2 - XmlWriter instead of FileStream
```

```
string sFile = @"C:/Temp/ArrHousesIXmlSerializable.xml";
XmlWriter fsOut;
try {
    fsOut = XmlWriter.Create(sFile);
}
catch { return; }
// Step 3 - XmlSerializer for an array – same as before
// Step 4 – Serialize – same as before
```

**Note:** A class implementing the IXmlSerializable interface cannot have any XML attribute applied to its declaration. If you try, you will get a runtime error.

The fields can have XML attributes applied to them *but these attributes will have no effect on serialization*.

## Serialized output

```
<?xml version="1.0" encoding="utf-8"?><ArrayOfClsXmlHouseAttribs
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <ClsXmlHouseAttribs ZipCode="10003"><sNumber>24-E</sNumber>
        New York</ClsXmlHouseAttribs>
    <ClsXmlHouseAttribs ZipCode="10001"><Number>241</Number>
        New York</ClsXmlHouseAttribs>
</ArrayOfClsXmlHouseAttribs>
```

## Code analysis

Basically, besides replacing a `FileStream` with an `XmlWriter`, you are doing nothing but creating an XML document yourself. So, there is nothing extraordinary to memorize here.

As in `GetObjectData()` for binary/soap serialization, we have complete control over XML Serialization.

a) We pass those fields to the output that we want.

b) The names of these fields can be anything that we want.

c) If we have a field of user-defined type, then we do not have to take care of its polymorphism since output will only be read off the fields / properties / functions of the superclass (the subclass is not known to the serializable class). Passing a field of user-defined type or an array type directly to a *Write* method is tricky, so avoid that.

d) Polymorphism related to serializing an array follows simple polymorphism principles.

For simplification, we are leaving out any output related to the `Occupants` field, the `ClsPerson` class and the `AgeGroup` enum. We can get a customized output in the same manner as has been shown for the other fields.

One trivially interesting portion is the `GetSchema` method which returns `null`. Let it suffice to say that this exact implementation is advised by Microsoft.

The real interesting portion is the *Write function set* available to the `XmlWriter` class, which we can use, although no one stops us from writing the entire output using its `WriteValue` or `WriteRaw` method.

Below is a list of salient *Write* functions of this class, which you can use with complete ingenuity. You can find the complete list on the MSDN website.

| `XmlWriter` functions | |
|---|---|
| **Method** | **Description** |
| **WriteAttributeString** | Writes an attribute with the specified value. |
| **WriteChars** | Writes text one buffer at a time. |
| **WriteComment** | Writes out a comment <!--...--> containing the specified text. |
| **WriteElementString** | Writes an element containing a string value. |
| **WriteEndAttribute** | Closes the last WriteStartAttribute call. |
| **WriteEndDocument** | Closes any open elements or attributes and puts the writer back in the Start state. |
| **WriteEndElement** | Closes one element and pops the corresponding namespace scope. |
| **WriteRaw** | Writes raw markup manually. |
| **WriteStartAttribute** | Writes the start of an attribute. |
| **WriteStartDocument** | Writes the XML declaration. |
| **WriteStartElement** | Writes the specified start tag. |
| **WriteValue** | Writes a single simple-typed value. |

# Custom XML Deserialization

## Method 1: Declarative

Since the class knows itself, the recreation of the fields based on any criterion (attribute or element, a particular element name or another, etc.) is a moot question.

The only scenario worth thinking about is the field that was not serialized. Remember, when we use `XmlSerializer`, the burden of managing serialization / deserialization is on this `XmlSerializer` rather

than the class. So, the logical way to get something done after deserialization finishes, is to create a callback or simply write the code after calling the `Deserialize` method. There is a delegate available (`XmlSerializationReadCallback`) for the first approach but Microsoft does not recommend using it. The second approach is self-explanatory.

# Method 2: Programmatic

We accomplish this by using the `ReadXml` method of the `IXmlSerializable` interface. Just remember that it should be a mirror image of `WriteXml`. Here is the implementation of `ReadXml` method for our `ClsXmlHouseAttribs` class:

```
public void ReadXml(XmlReader reader)
{
    this.sZip = reader.GetAttribute("ZipCode");
    reader.ReadStartElement();// Read off beginning of ClsXmlHouseAttribs
    string sNumber = reader.ReadElementContentAsString();
    int iNumber;
    if (int.TryParse(sNumber, out iNumber)) {
        this.sNumber = iNumber;
    }
    else {
       this.sNumber = sNumber;
    }
    this.sCity = reader.ReadContentAsString();
    reader.ReadEndElement();  // Read off end of ClsXmlHouseAttribs
}
```

## Advanced Scenarios

### Scenario 1: A field which was not serialized

Populate that field, as you would normally do – just as in *Advanced Scenarios* under *Custom Deserialization*.

### Scenario2: Deserializing an array

Nothing special.

### Scenario 3: An enum field

Since the underlying type of `enum` is `int`, `reader.ReadContentAsInt` followed by conversion to `enum` will do our task.

### Scenario 4: A field of a user-defined type

It was advised in the code analysis of `WriteXml` not to use any *Write* method to write a field of user-defined type directly. Conversely,

populate the fields of such a field by reading off the XML elements/attributes one by one.

### Scenarios 5, 6: An array field, polymorphism

Same action as for Scenario 4.

**Tip:** Try combining a few scenarios, as you have been doing.

Just like `XmlWriter`, `XmlReader` also has many useful functions, some of which are given below. For the exam, knowledge of all the methods is not necessary. Nonetheless, you can get the complete list from the MSDN website.

| `XmlReader` functions | |
|---|---|
| **Method** | **Description** |
| **GetAttribute** | Gets the value of an attribute. |
| **IsStartElement** | Tests if the current content node is a start tag. |
| **LookupNamespace** | Resolves a namespace prefix in the current element's scope. |
| **MoveToAttribute** | When overridden in a derived class, moves to the specified attribute. |
| **MoveToContent** | If the node is not a content node, the reader skips ahead to the next content node or end of file. |
| **MoveToElement** | Moves to the element that contains the current attribute node. |
| **MoveToFirstAttribute** | Moves to the first attribute. |
| **MoveToNextAttribute** | Moves to the next attribute. |
| **Read** | Reads the next node from the stream. |
| **ReadContentAs** | Reads the content as an object of the type specified. |
| **ReadContentAs<Type>** | Reads the content at the current position as the type in the method name. Examples are: `ReadContentAsBoolean`, `ReadContentAsDateTime`, `ReadContentAsDecimal`, `ReadContentAsDouble`, `ReadContentAsFloat`, `ReadContentAsInt`, `ReadContentAsLong`, `ReadContentAsObject`, `ReadContentAsString`. |
| **ReadElementContentAs** | Reads the current element and returns the contents as an object of the type specified. |
| **ReadElementContentAs<Type>** | Reads the current element value as the type in the method name. Examples are: |

| | |
|---|---|
| | `ReadElementContentAsBoolean`, `ReadElementContentAsDateTime`, `ReadElementContentAsDecimal`, `ReadElementContentAsDouble`, `ReadElementContentAsFloat`, `ReadElementContentAsInt`, `ReadElementContentAsLong`, `ReadElementContentAsObject`, `ReadElementContentAsString`. |
| **ReadElementString** | Helper method for reading simple text-only elements. |
| **ReadEndElement** | Checks that the current content node is an end tag and advances the reader to the next node. |
| **ReadInnerXml** | Reads all the content, including markup, as a string. |
| **ReadOuterXml** | Reads the content, including markup, representing this node and all its children. |
| **ReadStartElement** | Checks that the current node is an element and advances the reader to the next node. |
| **ReadString** | Reads the contents of an element or text node as a string. |
| **ReadToDescendant** | Advances the XmlReader to the next matching descendant element. |
| **ReadToFollowing** | Reads until the named element is found. |
| **ReadToNextSibling** | Advances the XmlReader to the next matching sibling element. |
| **Skip** | Skips the children of the current node. |

# Method 3: Programmatic - Use WriteXml and ReadXml directly

This method is just a slight (and easier) twist on the last method. Instead of creating an `XmlSerializer` and then calling its `Serialize` / `Deserialize` method, we can serialize an instance by calling its `WriteXml` method and deserialize by calling its `ReadXml` method.

**Note:** The `Dataset` class implements `IXmlSerializer`, just as we have done, enabling the client to call the `WriteXml` and `ReadXml` methods on its instance.

# Memory Sheet

## Binary Serialization

```
class –
  public constructor,[Serializable]

BinaryFormatter.Serialize(
  Stream, Object)
    OR
Object =
BinaryFormatter.Deserialize(Stream)
```

## Advanced Scenarios

```
Prevent serialization of a field –
  [NonSerialized]

Serialize an array or any
collection - Nothing special

Polymorphism/field of user-defined
type - make appropriate classes
serializable
```

## Custom Serialization

```
ISerializable.GetObjectData
  SerializationInfo.AddValue,
  StreamingContext

[OnSerializing], [OnSerialized]
```

## Custom Deserialization

```
ISerializable: special constructor
  SerializationInfo.GetValue,
  StreamingContext

[OnDeserializing], [OnDeserialized]
```

## XML Serialization

```
class – public, public constructor

Formatter =
  new XmlSerializer( typeof(Cls) )
```

## Advanced Scenarios

```
Prevent serialization of a field –
  [XmlIgnore]

Serialize an array or any
collection / field of user-defined
type or array type –
  Nothing special

Polymorphism –
  [XmlInclude] on parent class

Polymorphism over field –
  [XmlElement]

Polymorphism over array field –
  [XmlArrayItem]

Other attributes:
  [XmlEnum], [XmlAttribute],
  [XmlText], [XmlArray]
```

## Programmatic Custom XML Serialization

```
IXmlSerializable.WriteXml:XmlWriter
  WriteAttributeString,
  WriteElementString, WriteString,
  WriteValue, WriteRaw
```

## Programmatic Custom XML Deserialization

```
IXmlSerializable.ReadXml: XmlReader
  GetAttribute,
  ReadElementContentAsString,
  ReadString
```

# References

http://msdn.microsoft.com/en-us/library/ms973893.aspx#objserializ_topic5

http://msdn.microsoft.com/en-us/library/system.runtime.serialization.streamingcontextstates.aspx

http://msdn.microsoft.com/en-us/library/system.runtime.serialization.onserializingattribute.aspx

http://msdn.microsoft.com/en-us/library/system.xml.xmlwriter_methods.aspx

http://msdn.microsoft.com/en-us/library/system.xml.serialization.ixmlserializable.getschema.aspx

http://msdn.microsoft.com/en-us/library/system.xml.xmlreader_members.aspx