Web Content

Newsletter	
Issue Date	December, 2001
Section	
Main file name	
Accompanying ZIP file name	
Listing file name	
Sidebar file name	
Table file name	
Screen capture file names	
Infographic/illustration file names	
Photos or book scans	
Special instructions for Art dept.	
Editor	
Status	
Spellchecked (set Language to English U.S.)	
EN review	
Character count	
Newsletter blurb	

Overline:

.NET Object Serialization and Persistence

Byline:

by Juval Lowy

Technology Toolbox:
[[Art: Check box for those highlighted below.]]
VB.NET
C#
SQL Server 2000
ASP.NET
XML
VB6

Resources:

- xxxxxxx
- xxxxxxx

Head:

.NET Object Serialization and Persistence

Deck:

Software engineers model their application using objects. Objects maintain state (object's member variables) in memory, and apply business logic on that state. The question is, how is this state initially generated, and what is becoming of it when the application shuts down, or when the user selects Save.

In almost every application, object state is not created out of thin air when the application starts. The application typically loads some file containing the data, and converts the information in the file into a live object. Similarly, when the application shuts down (or when the application saves), the state of the object is persisted to a file.

Traditionally, this sequence is referred to as serialization and deserialization, because in cases of a simple binary persistent of the object state to a file, the application serially dumps the state of the object to a file, and serially reads the information in the file to a memory location, and associates that location with an object.

Traditionally, developers were left to their own devices when it came to implementing serialization. The result was not only that developers spent much of their valuable time on mundane serialization code instead of adding business value, but the resulting mechanism were proprietary and singular. There was no generic way for applications to share serialization files, because each used its own format, event if both applications run on the same platform and used the same language and the same memory layout, type size and so on.

.NET sets to improve on the existing predicament, by providing a standard, easy to use way of serializing and deserializing objects. Even though the .NET solution is trivial to use (in essence, the developer is not required to provide any explicit serialization code), it is also extensible, and developers can provide their own serialization format and code. In addition, .NET serialization is portable - .NET persists not only the object state but also its metadata, so in theory, .NET applications on different platforms (such as Windows and Linux) could exchange and share serialization files. .NET serialization is used not only with object persistence but also with marshaling by value across application domains. Please see my article on .NET Application Domains in the Visual Studio Magazine for more information on serialization and remoting.

.NET Serialization Basics

.NET serializes the object's state into a stream. A stream is a logical sequence of bytes, independent of particular devices such as files, memory, communication ports, and so on. This extra level of indirection lets developers use the same serialization code on all of these devices, by simply selecting a different stream type.

By default, objects are not serializable. The reason is, .NET has no way of knowing whether a serial dump of the object state to a stream makes sense. Maybe the object members have some transient value (such as an open connection), and serializing this member by value will produce errors during deserialization, because the object actually needs to reacquire the resource.

Serialization has to be done by consent. To make instances (objects) of your class serializable, add the [Serializable] attribute to your class definition:

```
[Serializable]
public class MyClass
{
    public int num1;
    public int num2;
}
```

In most cases, this is all a developer has to do. If you have a member variable in the class that you would like to preclude from serialization, then use the [NonSerialized] field attribute:

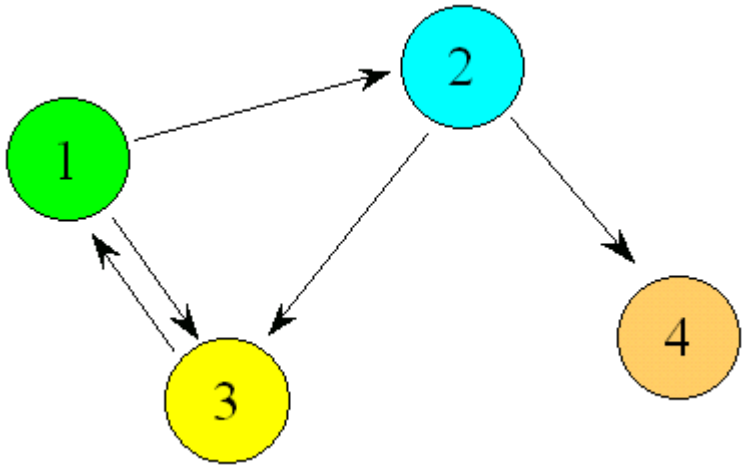
```
[Serializable]
public class MyClass
{
    public int num1; //num1 is serialized

    [NonSerialized]
    public int num2; //num2 is not serialized
}
```

in which case, the value deserialized will be the default type value (zero for integers, in the example above). The fact that a class is serializable is captured in the class's metadata, and .NET makes good use of this information in serialization and remoting scenarios.

.NET serialized the object's state including its bases class(es) members. All members are serialized, including private or protected members. .NET can do that because it accesses these members using reflection, and it knows about them from the object's metadata.

If the object has member variables that are serializable themselves, then those members are serialized too, because .NET traverses the entire objects graph (see Figure 1).

Figure 1: Serializing Object Graph. .NET serialization dumps the state of the object, including any members and their entire relationship graph. The algorithm handles cyclic references, and requires all objects to be serializable.



.NET serialization can even handle cyclic references, because during the recursive serialization, it tags objects it has already serialized. If even one of the object's members (or one of the nested members) is not serializable, then .NET throws a runtime exception. However, it is up to the developer to make sure when marking a class as serializable, that all its base classes are serializable too. Not doing so may yield non-deterministic results.

You can serialize the object in either binary or SOAP format. Binary format results in a more compact dump because it is more or less a memory image of the object (although .NET adds class name, assembly name and version). Binary format is fast to serialize and deserialize, because no parsing of the information is required. When deserializing, the binary image is loaded (more or less as is) straight into memory.

SOAP format represents the object's state in SOAP XML-based tags, and it requires composing the SOAP description during serialization and parsing during deserialization. As a result, it is slower than binary format, but the gain is portability – you can share the serialization information with any .NET application, on any platform.

Binary Serialization

To serialize an object to a binary format, use the BinaryFormatter class, defined in the System.Runtime.Serialization.Formatters.Binary namespace.

BinaryFormatter's Serialize() method dumps object state to a stream, and the Deserialize() creates a new object out of a given stream. Note, that the stream can be any stream such as file or memory stream.

All a developer has to do is create the BinaryFormatter object, create a stream (usually a file stream around a file) and call Serialize():

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

FileInfo fileInfo;fileInfo = new FileInfo(@"C:\temp\obj.bin");
BinaryFormatter formatter = new BinaryFormatter();
FileStream stream = fileInfo.Open(FileMode.OpenOrCreate);
MyClass obj = new MyClass();formatter.Serialize(stream,obj);
stream.Close();
```

When you are done with the stream, you should close it (just good housekeeping, nothing specific to serialization).

To deserialize an object, create a stream and a BinaryFormatter object, and call Deserialize(). Note that Deserialize() returns an object, so you have to downcast to the actual type:

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

FileInfo fileInfo;fileInfo = new FileInfo(@"C:\temp\obj.bin");
BinaryFormatter formatter = new BinaryFormatter();
FileStream stream = fileInfo.Open(FileMode.Open);
MyClass obj; //No new!!!
obj = (MyClass)formatter.Deserialize(stream);
stream.Close();
```

The downcast is safe, because the serialized information contains the object metadata, and .NET will throw an exception if you try and cast to an invalid type.

SOAP Serialization

To serialize an object using SOAP format, use the SoapFormatter class, defined in the System.Runtime.Serialization.Formatters.Soap namespace. The code for serialization and deserialization is virtual identical to binary format. To serialize, create the SoapFormatter object, create a stream file, and call Serialize():

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

FileInfo fileInfo;fileInfo = new FileInfo(@"C:\temp\obj.xml");
SoapFormatter formatter = new BinaryFormatter();
FileStream stream = fileInfo.Open(FileMode.OpenOrCreate);
MyClass obj = new MyClass();formatter.Serialize(stream,obj);
stream.Close();
```

To deserialize an object, create a stream and a SoapFormatter object, and call Deserialize():

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

FileInfo fileInfo;fileInfo = new FileInfo(@"C:\temp\obj.xml");
SoapFormatter formatter = new BinaryFormatter();
FileStream stream = fileInfo.Open(FileMode.Open); MyClass obj; //No new!!!
obj = (MyClass)formatter.Deserialize(stream);
stream.Close();
```

The reason using binary and SOAP formatters is so alike is because both formatters implement the IFormatter interface, defined as:

```
public interface IFormatter
{
   //Properties
   SerializationBinder Binder
   {get;set;}
   StreamingContext    Context
   {get;set;}
   ISurrogateSelector  SurrogateSelector
   {get;set;}

   //Methods
   object Deserialize(Stream serializationStream);
   void Serialize(Stream serializationStream,object graph);
}
```

The fact that both formatters are polymorphic with IFormatter allows you to write format-independent serialization code, as in the Serialization Demo application, discussed next.

The Serialization Demo Application

To demonstrate the key serialization points covered so far, download the Serialization Demo application from www.idesign.net

The application uses generic, format independent (using IFormatter) serialization code.

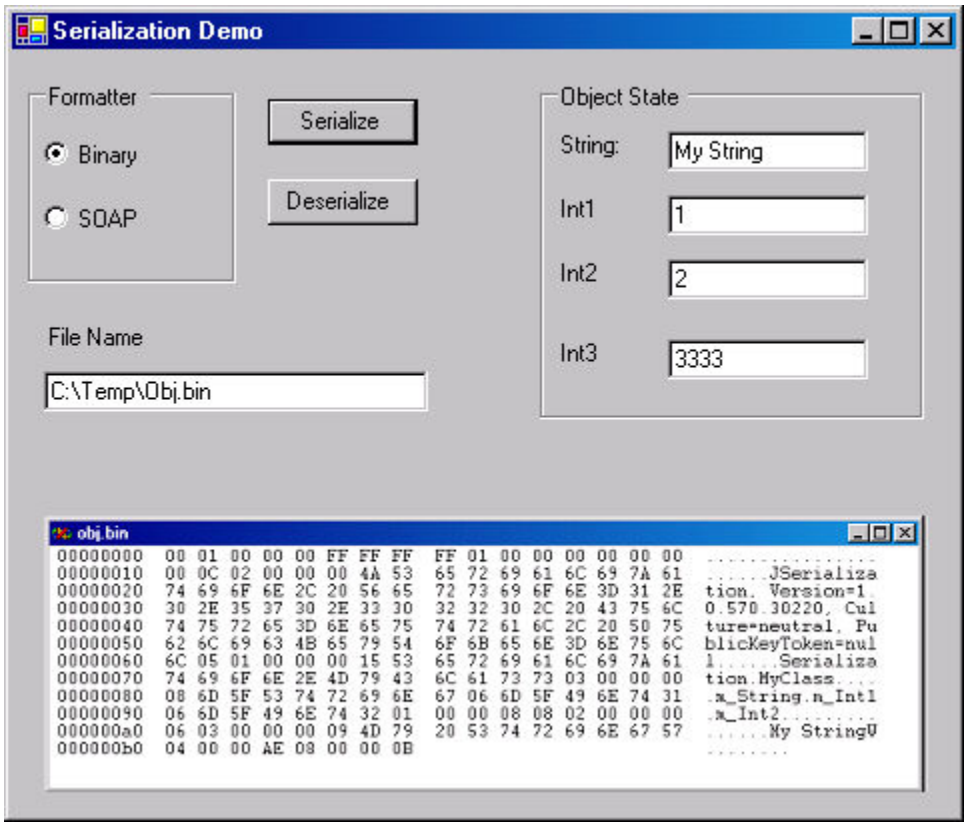
The application contains the MyClass definition:

```
[Serializable]
public class MyClass
{
    public MyClass()
    {
      m_String = "My String";
      m_Int1 = 1;
      m_Int3 = 3;
    }
    public void SomeMethod()
    {
      Debug.Assert(m_Int1 == 1);
    }
    public  string m_String;
    public  int m_Int1;
    private int m_Int2;
    public  int Int2
    {
      set
      {
         Int2 = value;
      }
      get
      {
         return m_Int2;
      }
    }
    [NonSerialized]
    public int m_Int3;
}
```

For demonstration purposes, I used the [NonSerialized] attribute to prevent a particular member from being serialized, and to see what happens instead.

The client form lets you select the serialization format using the Formatter radio button (see Figure 2).

Figure 2: The Serialization Demo Application. The application uses generic formatting code via IFormatter (actual format selected using the radio button). The form presents the objects state before and after serialization and the state filename.



The client form has a helper method called GetFormatter() that returns IFormatter on a binary or SOAP formatter, based on the radio box selection:

```
private IFormatter GetFormatter()
{
    if(m_BinaryRadio.Checked)
    {
       return new BinaryFormatter();
    }
    else
    {
       return new SoapFormatter();
    }
}
```

The client actually does not have a MyClass object as a member variable. It has a set of controls in the Object State group. When asked to serialize, the client converts the values

of the controls into fields in a new object, and serializes it. When asked to de-serialize, the client creates a new object from a previously saved serialization file, and copies the object fields to the controls.

Custom Serialization

Sometimes, the default serialization using [Serializable] is not good enough. Imagine situations where you want to encrypt the objects state in the stream (the file usually), or when you need to reacquire resources.

In these cases, you can still rely on .NET serialization infrastructure, but provide your own custom behavior. All you have to do is implement the ISerializable interface, defined as:

```
public interface ISerializable
{
   void GetObjectData(SerializationInfo info,StreamingContext context);
}
```

.NET checks (via metadata) that the object implements ISerializable, and calls GetObejctData() when the objects is serialized. At this point, it is up to the object to provide the state information. The info parameter provides AddValue() method, that lets you hand over a named value/key pair for each element of the object's state.

The context parameter lets your object know why it is being serialized. Possible reasons are remoting, cross-context, cross-app domain, file dump and so on. The context parameter is largely ignored, and is used only in advanced scenarios.

To support deserialization, the object must provide special parameterized constructor with this signature:

```
<Class Name>(SerializationInfo info,StreamingContext context);
```

.NET calls this constructor during deserialization. The constructor can (and should) be private, to prevent normal clients from calling it. .NET will use reflection to invoke it.

The info parameter provides Get<Type> methods, that allow the object to retrieve its state.

Listing 1 shows how to provide custom serialization for a class with one member variable of type int.

Listing 1: Implementing ISerializable. The class should implement the GetObjectData() method and a special constructor for deserialization. The class uses named value/key pair to store and retrieve the state.

```
using System.Runtime.Serialization;

public class MyClass : ISerializable
{
   private int m_MyInt;
   public void GetObjectData(SerializationInfo info,StreamingContext context)
   {
      info.AddValue("My int",m_MyInt);
   }
   private MyClass(SerializationInfo info,StreamingContext context)
   {
      m_MyInt = info.GetInt32("My int");
   }
}
```