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.