Understanding Object Serialization

The term *serialization* describes the process of persisting (and possibly transferring) the state of an object into a stream (e.g., file stream and memory stream).

The persisted data sequence contains all the necessary information you need to reconstruct (or *deserialize*) the state of the object for use later.

Using this technology makes it trivial to save vast amounts of data (in various formats). In many cases, saving application data using serialization services results in less code than using the readers/writers you find in the System.IO namespace.

For example, assume you want to create a GUI-based desktop application that provides a way for end users to save their preferences (e.g., window color and font size). To do this, you might define a class

named UserPrefs that encapsulates 20 or so pieces of field data.

Now, if you were to use a

System.IO.BinaryWriter type, you would need to save each field of the UserPrefs object *manually*.

Likewise, if you were to load the data from a file back into memory, you would need to use a

System.IO.BinaryReader and (once again) *manually* read in each value to reconfigure a new UserPrefs object. This is all doable, but you can save yourself a good amount of time by marking the UserPrefs class with the [Serializable] attribute, like so:

Be aware that you cannot inherit the [Serializable] attribute from a parent class. Therefore, if you

derive a class from a type marked [Serializable], the child class must be marked [Serializable] aswell, or it cannot be persisted.

In fact, all objects in an object graph must be marked with the

[Serializable] attribute. If you attempt to serialize a nonserializable object using the BinaryFormatter or SoapFormatter, you will receive a SerializationException at runtime.

**Public Fields, Private Fields, and Public Properties**

Notice that in each of these classes, you define the field data as public; this helps keep the example simple. Of course, private data exposed using public properties would be preferable from an OO point of

view. Also, for the sake of simplicity, this example does not define any custom constructors on these

types; therefore, all unassigned field data will receive the expected default values.

OO design principles aside, you might wonder how the various formatters expect a type’s field data to be defined in order to be serialized into a stream.

The answer is that it depends. If you persist an

object’s state using the BinaryFormatter or SoapFormatter, it makes absolutely no difference.

These types are programmed to serialize *all* serializable fields of a type, regardless of whether they are public fields,

private fields, or private fields exposed through public properties.

Recall, however, that if you have

points of data that you do not want to be persisted into the object graph, you can selectively mark public

or private fields **as [NonSerialized],**

The situation is quite different if you use the XmlSerializer type, however. This type will *only* serialize **public data fields** or private data exposed by public properties. **Private data** not exposed from properties will be **ignored.**

consider the following serializable Person type:

[Serializable]

public class Person

{

// A public field.

public bool isAlive = true;

// A private field.

private int personAge = 21;

// Public property/private data.

private string fName = string.Empty;

public string FirstName

{

get { return fName; }

set { fName = value; }

}

}

If you processed the preceding with BinaryFormatter or SoapFormatter, you would find that the isAlive, personAge, and fName fields are saved into the selected stream.

However, the XmlSerializer would *not* save the value of personAge because this piece of private data is not encapsulated by a public type property.

If you wished to persist the age of the person with the XmlSerializer, you would need to

define the field publicly or encapsulate the private member using a public property.