

Serialization: An Essential Part of Successful .NET Customization

by Chris Freeman, Technical Contributor

Introduction

Part of the power and beauty of extensible .NET products such as FarPoint Spread is the ability to create custom cell types, custom editors, and other custom classes. When developing custom classes for your application, be sure to provide serialization correctly so that you can allow copying to the Clipboard or saving to a binary stream such as a database. Remembering serialization is key to a successful customization. With the techniques described in this short technical article, you should be able to make most any custom class serializable and ensure its successful use in your application.

Whether the classes are cell type-derived classes, or custom editors, or custom renderers, or based on underlying .NET classes, these simple rules apply. To allow your classes to be serialized, there are certain interfaces you must support. To avoid serialization issues for your application, follow these steps:

1. Add the Serializable attribute to tell the framework that your class is serializable.
2. Implement the ISerializable interface to ensure both proper serialization and deserialization.
3. Add a “serialization” constructor to allow streams like the Clipboard to create instances of your class.
4. If needed, implement the IDeserializationCallback interface to ensure that all child objects are deserialized.
5. Choose a base class from which to derive your class that is also serializable or else be sure to serialize all of the properties and settings.

```
In C#:  
[Serializable]  
public class CustomTextEditor : IEditor, ISerializable, IDeserializationCallback  
{  
  
In Visual Basic .NET:  
<Serializable()> _  
Public Class CustomTextEditor  
    Implements IEditor ' FarPoint.Win.Spread.CellType  
    Implements ISerializable ' System.Runtime.Serialization  
    Implements IDeserializationCallback ' System.Runtime.Serialization
```

Add the Serializable Attribute

The Serializable attribute indicates to the framework that your class can be serialized. This is all that is necessary to support basic serialization of an object, but without an implementation of the ISerializable interface, the object uses default serialization provided by the framework. This process uses reflection to enumerate the fields in the class and to save each to the serialization stream, unless it has the NonSerialized attribute.

Implement the ISerializable Interface

Adding support for the ISerializable interface ensures that you can control your own serialization and deserialization. There is one method to implement and that is: GetObjectData. According to the .NET Framework Reference documentation, this method “Populates a SerializationInfo with the data needed to serialize the target object.” SerializationInfo is the object that holds the serialized object data. StreamingContext is the contextual information about the source or destination of the serialization.

For more information on contextual information, see the SerializationInfo class and the StreamingContext structure in the Microsoft .NET Framework Reference.

It is better to implement `ISerializable` than to depend on the default serialization because it ensures completeness. (It is also better because it is more efficient than the default implementation.) This gives you much more control over the deserialization process, especially over which fields are included in the serialization stream and how. By implementing the `ISerializable` interface, you can ensure backwards compatibility with older versions of your assembly, so that objects saved with an old version still load in the new version.

Add a Serialization Constructor

For a complete end-to-end implementation, the interface also requires implementation of a special constructor, called the “magic constructor,” which has the same parameters as the `GetObjectData` method, and should perform initialization of the object (the reverse of the serialization done in `GetObjectData`). This provides the deserialization portion of the implementation.

Implement the IDeserializationCallback Interface

Note that when an object is deserialized, the contained objects in its serialization stream have not yet been deserialized. If your code tries to call into any instance property or method on the sub object instance, it may get an exception because that instance has not been initialized yet. If you need to make calls into contained objects to complete the deserialization process, then you need to implement the `IDeserializationCallback` interface.

Adding support for the `IDeserializationCallback` interface enables your object to perform additional initialization after the deserialization of your object and all sub-objects. This interface is not called until after your object and all of its child objects have been deserialized. This interface has one method to implement and that is the `OnDeserialization` method.

Choose the Correct Base Class

In most cases your class should be derived from a base class that is also serializable or do not declare a base class (which derives your class from ‘object’). This ensures correct serialization of all necessary properties in the base class to properly instantiate your class when the time comes. If your base class does not support serialization, you are then responsible in your class’s serialization methods for persisting any properties that you need to make your class work.

Here is another tip that we have found useful. If your base class is not serializable, it may be better to create a class from ‘object’ and then create non-serialized member variables of the base class you would like to use. For example, in *FarPoint Spread*, you may want a custom editor for a cell. Instead of deriving your editor from `TextBox` (which is not serializable) and returning ‘this’ in C# (or ‘Me’ in VB) from the `GetEditorControl` method required for supporting the `IEditor` interface, you may want to create a private member variable of type `TextBox` and mark it as nonserializable and then return that `textbox` instance object from the `GetEditorControl` method. Refer to the sample source code associated with this object.

Sample Code

Along with this article, there is source code in C# and Visual Basic for supporting a custom cell type in *FarPoint Spread* that is serializable as well as a custom editor class used in the custom cell type that is also serializable. Using the sample source code as a reference, you should be able to apply these techniques to your own application to make most any class serializable to the Clipboard or binary stream.

These associated files that contain code samples are available online:

- Serializable_CustomClasses.cs –
C# source code sample
- Serializable_CustomClasses.vb –
VB source code sample