17a

Data Contract Serializer

The data contract serializer supports a *data contract* model that helps you decouple the low-level details of the types you want to serialize from the structure of the serialized data. This provides excellent version tolerance, meaning you can deserialize data that was serialized from an earlier or later version of a type. You can even deserialize types that have been renamed or moved to a different assembly.

The data contract serializer can cope with most object graphs, although it can require more assistance than the binary serializer. It can also be used as a general-purpose tool for reading/writing XML files, if you're flexible on how the XML is structured. (If you need to store data in attributes or cope with XML elements presenting in a arbitrary order, you cannot use the data contract serializer.)

The Data Contract Serializer

Here are the basic steps in using the data contract serializer:

1. Decide whether to use `DataContractSerializer` or `DataContractJsonSerializer` (in .NET Framework, there's also the `NetDataContractSerializer`).
2. Decorate the types and members you want to serialize with `[DataContract]` and `[DataMember]` attributes, respectively.
3. Instantiate the serializer and call `WriteObject` or `ReadObject`.

If you chose the `DataContractSerializer`, you will also need to register "known types" (subtypes that can also be serialized), and decide whether to preserve object references.

You may also need to take special action to ensure that collections are properly serialized.

> Types for the data contract serializer are defined in the `System.Runtime.Serialization` namespace, in an assembly of the same name.

Choosing a Serializer

There are three data contract serializers:

`DataContractSerializer`
: Loosely couples .NET types to data contract types via XML

`DataContractJsonSerializer`
: Loosely couples .NET types to data contract types via JSON

`NetDataContractSerializer`
: Tightly couples .NET types to data contract types (.NET Framework only)

© Joseph Albahari, 2007-2017

The first two require that you explicitly register serializable subtypes in advance so that it can map a data contract name such as "Person" to the correct .NET type. The `NetDataContractSerializer` requires no such assistance, because it writes the full type and assembly names of the types it serializes, rather like the binary serialization engine:

```
<Person z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

Such output relies on the presence of a specific .NET type in a specific namespace and assembly in order to deserialize.

If you're saving an object graph to a "black box," you can choose any serializer, depending on what benefits are more important to you. If you're communicating through WCF, or reading/writing an XML file, you'll most likely want the `DataContractSerializer`.

We'll go into each of these topics in more detail in the following sections.

Using the Serializers

After choosing a serializer, the next step is to attach attributes to the types and members you want to serialize. At a minimum:

- Add the `[DataContract]` attribute to each type.
- Add the `[DataMember]` attribute to each member that you want to include.

Here's an example:

```
namespace SerialTest
{
  [DataContract] public class Person
  {
    [DataMember] public string Name;
    [DataMember] public int Age;
  }
}
```

These attributes are enough to make a type *implicitly* serializable through the data contract engine.

You can then *explicitly* serialize or deserialize an object instance by instantiating a serializer and calling `WriteObject` or `ReadObject`:

```
Person p = new Person { Name = "Stacey", Age = 30 };

var ds = new DataContractSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  ds.WriteObject (s, p);                          // Serialize

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) ds.ReadObject (s);                // Deserialize

Console.WriteLine (p2.Name + " " + p2.Age);       // Stacey 30
```

`DataContractSerializer`'s constructor requires the *root object* type (the type of the object you're explicitly serializing). In contrast, `NetDataContractSerializer` does not:

```
var ns = new NetDataContractSerializer();

// NetDataContractSerializer is otherwise the same to use
// as DataContractSerializer.
...
```

Both types of serializer use the XML formatter by default. With an `XmlWriter`, you can request that the output be indented for readability:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };
using (XmlWriter w = XmlWriter.Create ("person.xml", settings))
  ds.WriteObject (w, p);

System.Diagnostics.Process.Start ("person.xml");
```

Here's the result:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Stacey</Name>
</Person>
```

The XML element name `<Person>` reflects the *data contract name*, which, by default, is the .NET type name. You can override this and explicitly state a data contract name as follows:

```
[DataContract (Name="Candidate")]
public class Person { ... }
```

The XML namespace reflects the *data contract namespace*, which, by default, is *http://schemas.datacontract.org/2004/07/*, plus the .NET type namespace. You can override this in a similar fashion:

```
[DataContract (Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```

> Specifying a name and namespace decouples the contract identity from the .NET type name. It ensures that, should you later refactor and change the type's name or namespace, serialization is unaffected.

You can also override names for data members:

```
[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember (Name="FirstName")]  public string Name;
  [DataMember (Name="ClaimedAge")] public int Age;
}
```

Here's the output:

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Stacey</FirstName>
</Candidate>
```

`[DataMember]` supports both fields and properties—public and private. The field or property's data type can be any of the following:

- Any primitive type
- `DateTime`, `TimeSpan`, `Guid`, `Uri`, or an `Enum` value
- Nullable versions of the above
- `byte[]` (serializes in XML to base 64)
- Any "known" type decorated with `DataContract`
- Any `IEnumerable` type (see the section "Serializing Collections" later in this chapter)
- Any type with the `[Serializable]` attribute or implementing `ISerializable` (see the section "Extending Data Contracts" later in this chapter)
- Any type implementing `IXmlSerializable`

Specifying a binary formatter (.NET Framework only)

You can use a binary formatter with `DataContractSerializer` or `NetDataContractSerializer`. The process is the same:

```
Person p = new Person { Name = "Stacey", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

var s = new MemoryStream();
using (XmlDictionaryWriter w = XmlDictionaryWriter.CreateBinaryWriter (s))
  ds.WriteObject (w, p);

var s2 = new MemoryStream (s.ToArray());
Person p2;
using (XmlDictionaryReader r = XmlDictionaryReader.CreateBinaryReader (s2,
                               XmlDictionaryReaderQuotas.Max))
  p2 = (Person) ds.ReadObject (r);
```

The output varies between being slightly smaller than that of the XML formatter, and radically smaller if your types contain large arrays.

Serializing Subclasses

You don't need to do anything special to handle the serializing of subclasses with the `NetDataContractSerializer`. The only requirement is that subclasses have the `DataContract` attribute. The serializer will write the fully qualified names of the actual types that it serializes as follows:

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
```

A `DataContractSerializer`, however, must be informed about all subtypes that it may have to serialize or deserialize. To illustrate, suppose we subclass `Person` as follows:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
}
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

and then write a method to clone a `Person`:

```
static Person DeepClone (Person p)
{
  var ds = new DataContractSerializer (typeof (Person));
  MemoryStream stream = new MemoryStream();
  ds.WriteObject (stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject (stream);
}
```

which we call as follows:

```
Person  person  = new Person  { Name = "Stacey", Age = 30 };
Student student = new Student { Name = "Stacey", Age = 30 };
Teacher teacher = new Teacher { Name = "Stacey", Age = 30 };

Person  p2 =           DeepClone (person);     // OK
Student s2 = (Student) DeepClone (student);    // SerializationException
Teacher t2 = (Teacher) DeepClone (teacher);    // SerializationException
```

`DeepClone` works if called with a `Person` but throws an exception with a `Student` or `Teacher`, because the deserializer has no way of knowing what .NET type (or assembly) a "Student" or "Teacher" should resolve to. This also helps with security, in that it prevents the deserialization of unexpected types.

The solution is to specify all permitted or "known" subtypes. You can do this either when constructing the `DataContractSerializer`:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

or in the type itself, with the `KnownType` attribute:

```
[DataContract, KnownType (typeof (Student)), KnownType (typeof (Teacher))]
public class Person
...
```

Here's what a serialized `Student` now looks like:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        i:type="Student" >
  ...
<Person>
```

Because we specified `Person` as the root type, the root element still has that name. The actual subclass is described separately—in the `type` attribute.

> The `NetDataContractSerializer` suffers a performance hit when serializing subtypes—with either formatter. It seems that when it encounters a subtype, it has to stop and think for a while!
>
> Serialization performance matters on an application server that's handling many concurrent requests.

Object References

References to other objects are serialized, too. Consider the following classes:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}

[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Here's the result of serializing this to XML using the `DataContractSerializer`:

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </HomeAddress>
  <Name>...</Name>
</Person>
```

> The `DeepClone` method we wrote in the preceding section would clone `HomeAddress`, too—distinguishing it from a simple `MemberwiseClone`.

If you're using a `DataContractSerializer`, the same rules apply when subclassing `Address` as when subclassing the root type. So, if we define a `USAddress` class, for instance:

```
[DataContract]
public class USAddress : Address { }
```

and assign an instance of it to a Person:

```
Person p = new Person { Name = "John", Age = 30 };
p.HomeAddress = new USAddress { Street="Fawcett St", Postcode="02138" };
```

p could not be serialized. The solution is either to apply the KnownType attribute to Address:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

or to tell DataContractSerializer about USAddress in construction:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (USAddress) } );
```

(We don't need to tell it about Address because it's the declared type of the HomeAddress data member.)

Preserving object references

The NetDataContractSerializer always preserves referential equality. The DataContractSerializer does not, unless you specifically ask it to. (The DataContractJsonSerializer cannot preserve referential integrity at all.)

This means that if the same object is referenced in two different places, a DataContractSerializer ordinarily writes it twice. So, if we modify the preceding example so that Person also stores a work address:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address HomeAddress, WorkAddress;
}
```

and then serialize an instance as follows:

```
Person p = new Person { Name = "Stacey", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```

we would see the same address details twice in the XML:

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</WorkAddress>
```

When this was later deserialized, WorkAddress and HomeAddress would be different objects. The advantage of this system is that it keeps the XML simple and standards-compliant. The disadvantages of this system include larger XML, loss of referential integrity, and the inability to cope with cyclical references.

You can request referential integrity by specifying true for the IsReference parameter of DataContractAttribute

```
[DataContract(IsReference=true)] public class Person
```

or for preserveObjectReferences when constructing a DataContractSerializer:

```
var ds = new DataContractSerializer (typeof (Person),
                                     null, 1000, false, true, null);
```

The third argument is mandatory when `preserveObjectReferences` is `true`: it indicates the maximum number of object references that the serializer should keep track of. The serializer throws an exception if this number is exceeded (this prevents a denial of service attack through a maliciously constructed stream).

Here's what the XML then looks like for a `Person` with the same home and work addresses:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        z:Id="1">
  <Age>30</Age>
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Stacey</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

The cost of this is in reduced interoperability (notice the proprietary namespace of the `Id` and `Ref` attributes).

Version Tolerance

You can add and remove data members without breaking forward or backward compatibility. By default, the data contract deserializers do the following:

- Skip over data for which there is no `[DataMember]` in the type.
- Won't complain if any `[DataMember]` is missing in the serialization stream.

Rather than skipping over unrecognized data, you can instruct the deserializer to store unrecognized data members in a black box, and then replay them should the type later be reserialized. This allows you to correctly round-trip data that's been serialized by a later version of your type. To activate this feature, implement `IExtensibleDataObject`. This interface really means "IBlackBoxProvider." It requires that you implement a single property, to get/set the black box:

```
[DataContract] public class Person : IExtensibleDataObject{
  [DataMember] public string Name;
  [DataMember] public int Age;

  ExtensionDataObject IExtensibleDataObject.ExtensionData { get; set; }
}
```

Required members

If a member is essential for a type, you can demand that it be present with `IsRequired`:

```
[DataMember (IsRequired=true)] public int ID;
```

If that member is not present, an exception is then thrown upon deserialization.

Member Ordering

The data contract serializers are extremely fussy about the ordering of data members. The deserializers, in fact, *skip over any members considered out of sequence*.

Members are written in the following order when serializing:

1. Base class to subclass
2. Low `Order` to high `Order` (for data members whose `Order` is set)
3. Alphabetical order (using *ordinal* string comparison)

So, in the preceding examples, `Age` comes before `Name`. In the following example, `Name` comes before `Age`:

```
[DataContract] public class Person
{
```

```
  [DataMember (Order=0)] public string Name;
  [DataMember (Order=1)] public int Age;
}
```

If `Person` has a base class, the base class's data members would all serialize first.

The main reason to specify an order is to comply with a particular XML schema. XML element order equates to data member order.

If you don't need to interoperate with anything else, the easiest approach is *not* to specify a member `Order` and rely purely on alphabetical ordering. A discrepancy will then never arise between serialization and deserialization as members are added and removed. The only time you'll come unstuck is if you move a member between a base class and a subclass.

Null and Empty Values

There are two ways to deal with a data member whose value is null or empty:

1. Explicitly write the null or empty value (the default).
2. Omit the data member from the serialization output.

In XML, an explicit null value looks like this:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

Writing null or empty members can waste space, particularly on a type with lots of fields or properties that are usually left empty. More importantly, you may need to follow an XML schema that expects the use of optional elements (e.g., `minOccurs="0"`) rather than `nil` values.

You can instruct the serializer not to emit data members for null/empty values as follows:

```
[DataContract] public class Person
{
  [DataMember (EmitDefaultValue=false)] public string Name;
  [DataMember (EmitDefaultValue=false)] public int Age;
}
```

`Name` is omitted if its value is `null`; `Age` is omitted if its value is `0` (the default value for the `int` type). If we were to make `Age` a nullable `int`, then it would be omitted if (and only if) its value was null.

> The data contract deserializer, in rehydrating an object, bypasses the type's constructors and field initializers. This allows you to omit data members as described without breaking fields that are assigned nondefault values through an initializer or constructor. To illustrate, suppose we set the default `Age` for a `Person` to 30 as follows:
>
> ```
> [DataMember (EmitDefaultValue=false)]
> public int Age = 30;
> ```
>
> Now suppose that we instantiate `Person`, explicitly set its `Age` from 30 to 0, and then serialize it. The output won't include `Age`, because 0 is the default value for the `int` type. This means that in deserialization, `Age` will be ignored and the field will remain at its default value—which fortunately is 0, given that field initializers and constructors were bypassed.

Data Contracts and Collections

The data contract serializers can save and repopulate any enumerable collection. For instance, suppose we define `Person` to have a `List<>` of addresses:

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
```

```
}

[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Here's the result of serializing a `Person` with two addresses:

```
<Person ...>
  ...
  <Addresses>
    <Address>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Address>
    <Address>
      <Postcode>6152</Postcode>
      <Street>Comer St</Street>
    </Address>
  </Addresses>
  ...
</Person>
```

Notice that the serializer doesn't encode any information about the particular *type* of collection it serialized. If the `Addresses` field was instead of type `Address[]`, the output would be identical. This allows the collection type to change between serialization and deserialization without causing an error.

Sometimes, though, you need your collection to be of a more specific type than you expose. An extreme example is with interfaces:

```
[DataMember] public IList<Address> Addresses;
```

This serializes correctly (as before), but a problem arises in deserialization. There's no way the deserializer can know which concrete type to instantiate, so it chooses the simplest option—an array. The deserializer sticks to this strategy even if you initialize the field with a different concrete type:

```
[DataMember] public IList<Address> Addresses = new List<Address>();
```

(Remember that the deserializer bypasses field initializers.) The workaround is to make the data member a private field and add a public property to access it:

```
[DataMember (Name="Addresses")] List<Address> _addresses;

public IList<Address> Addresses { get { return _addresses; } }
```

In a nontrivial application, you would probably use properties in this manner anyway. The only unusual thing here is that we've marked the private field as the data member, rather than the public property.

Subclassed Collection Elements

The serializer handles subclassed collection elements transparently. You must declare the valid subtypes just as you would if they were used anywhere else:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}

public class USAddress : Address { }
```

Adding a `USAddress` to a `Person`'s address list then generates XML like this:

```
...
  <Addresses>
    <Address i:type="USAddress">
```

```
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

Customizing Collection and Element Names

If you subclass a collection class itself, you can customize the XML name used to describe each element by attaching a `CollectionDataContract` attribute:

```
[CollectionDataContract (ItemName="Residence")]
public class AddressList : Collection<Address> { }

[DataContract] public class Person
{
  ...
  [DataMember] public AddressList Addresses;
}
```

Here's the result:

```
...
  <Addresses>
    <Residence>
      <Postcode>6020</Postcode
      <Street>Odo St</Street>
    </Residence>
    ...
```

`CollectionDataContract` also lets you specify a `Namespace` and `Name`. The latter is not used when the collection is serialized as a property of another object (such as in this example), but it is when the collection is serialized as the root object.

You can also use `CollectionDataContract` to control the serialization of dictionaries:

```
[CollectionDataContract (ItemName="Entry",
                         KeyName="Kind",
                       ValueName="Number")]
public class PhoneNumberList : Dictionary <string, string> { }

[DataContract] public class Person
{
  ...
  [DataMember] public PhoneNumberList PhoneNumbers;
}
```

Here's how this formats:

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
    </Entry>
  </PhoneNumbers>
```

Extending Data Contracts

This section describes how you can extend the capabilities of the data contract serializer through serialization hooks, `[Serializable]` and `IXmlSerializable`.

Serialization and Deserialization Hooks

You can request that a custom method be executed before or after serialization, by flagging the method with one of the following attributes:

`[OnSerializing]`
: Indicates a method to be called just *before* serialization

`[OnSerialized]`
: Indicates a method to be called just *after* serialization

Similar attributes are supported for deserialization:

`[OnDeserializing]`
: Indicates a method to be called just *before* deserialization

`[OnDeserialized]`
: Indicates a method to be called just *after* deserialization

The custom method must have a single parameter of type `StreamingContext`. This parameter is required for consistency with the binary engine, and it is not used by the data contract serializer.

`[OnSerializing]` and `[OnDeserialized]` are useful in handling members that are outside the capabilities of the data contract engine, such as a collection that has an extra payload or that does not implement standard interfaces. Here's the basic approach:

```
[DataContract] public class Person
{
  public SerializationUnfriendlyType Addresses;

  [DataMember (Name="Addresses")]
  SerializationFriendlyType _serializationFriendlyAddresses;

  [OnSerializing]
  void PrepareForSerialization (StreamingContext sc)
  {
    // Copy Addresses—> _serializationFriendlyAddresses
    // ...
  }

  [OnDeserialized]
  void CompleteDeserialization (StreamingContext sc)
  {
    // Copy _serializationFriendlyAddresses—> Addresses
    // ...
  }
}
```

An `[OnSerializing]` method can also be used to conditionally serialize fields:

```
public DateTime DateOfBirth;

[DataMember] public bool Confidential;

[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;

[OnSerializing]
void PrepareForSerialization (StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

Recall that the data contract deserializers bypass field initializers and constructors. An [OnDeserializing] method acts as a pseudoconstructor for deserialization, and it is useful for initializing fields excluded from serialization:

```
[DataContract] public class Test
{
  bool _editable = true;

  public Test() { _editable = true; }

  [OnDeserializing]
  void Init (StreamingContext sc)
  {
    _editable = true;
  }
}
```

If it wasn't for the Init method, _editable would be false in a deserialized instance of Test—despite the other two attempts at making it true.

Methods decorated with these four attributes can be private. If subtypes need to participate, they can define their own methods with the same attributes, and they will get executed, too.

Interoperating with [Serializable]

The data contract serializer can also serialize types marked with the binary serialization engine's attributes and interfaces. This ability is important, since support for the binary engine has been woven into much of what was written prior to Framework 3.0—including .NET itself!

The following things flag a type as being serializable for the binary engine:

- The [Serializable] attribute
- Implementing ISerializable

Binary interoperability is useful in serializing existing types as well as new types that need to support both engines. It also provides another means of extending the capability of the data contract serializer, because the binary engine's ISerializable is more flexible than the data contract attributes. Unfortunately, the data contract serializer is inefficient in how it formats data added via ISerializable.

A type wanting the best of both worlds cannot define attributes for both engines. This creates a problem for types such as string and DateTime, which for historical reasons cannot divorce the binary engine attributes. The data contract serializer works around this by filtering out these basic types and processing them specially. For all other types marked for binary serialization, the data contract serializer applies similar rules to what the binary engine would use. This means it honors attributes such as NonSerialized or calls ISerializable if implemented. It does not *thunk* to the binary engine itself—this ensures that output is formatted in the same style as if data contract attributes were used.

Types designed to be serialized with the binary engine expect object references to be preserved. You can enable this option through the DataContractSerializer (or by using the NetDataContractSerializer).

The rules for registering known types also apply to objects and subobjects serialized through the binary interfaces.

The following example illustrates a class with a [Serializable] data member:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address MailingAddress;
}
[Serializable] public class Address
{
```

```
  public string Postcode, Street;
}
```

Here's the result of serializing it:

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

Had `Address` implemented `ISerializable`, the result would be less efficiently formatted:

```
  <MailingAddress>
    <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
      i:type="d3p1:string" xmlns="">str</Street>
    <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
      i:type="d3p1:string" xmlns="">pcode</Postcode>
  </MailingAddress>
```

Interoperating with IXmlSerializable

A limitation of the data contract serializer is that it gives you little control over the structure of the XML. In a WCF application this can actually be beneficial, in that it makes it easier for the infrastructure to comply with standard messaging protocols.

If you do need precise control over the XML, you can implement `IXmlSerializable` and then use `XmlReader` and `XmlWriter` to manually read and write the XML. The data contract serializer allows you to do this just on the types for which this level of control is required. We further describe the `IXmlSerializable` interface in Chapter 17.