TopXML
Great XML tools

For more great .NET and XML content go to http://www.topxml.com

XMLSerializer in .NET

by Christoph Schittko

XmlSerializer in .NET

XML was designed to be a technology for data exchange across heterogeneous systems. It can be easily transmitted between distributed components because of its platform independence and its simple, text-based, self-describing format. Yet these features hardly form the basis for a solid programming platform. Text-based data does not enforce type-safety rules. Programmers are much more enticed by object-oriented programming models, because each object is of a certain type, so the compiler can warn of potential type problems and data encapsulated by an object can be easily accessed. The ideal programming environment would use an object-oriented model to build the software but leverage the benefits of XML to communicate between distributed components, over the Internet or Message Queues for example.

Unfortunately the Microsoft platform lacked an easy, integrated way to transform objects in a program into XML documents and XML documents into objects. When it came to building XML-driven applications, developers could choose to bypass all type-safety and pass XML around in their systems, but in order to build a clean, object oriented architecture they had to invest time and resources into adding functionality to every class to save itself to XML. This functionality is referred to as serialization. The reverse operation, when an object is re-created from its serialized representation is called deserialization or XML data bindng. Figure 1 illustrates this process.

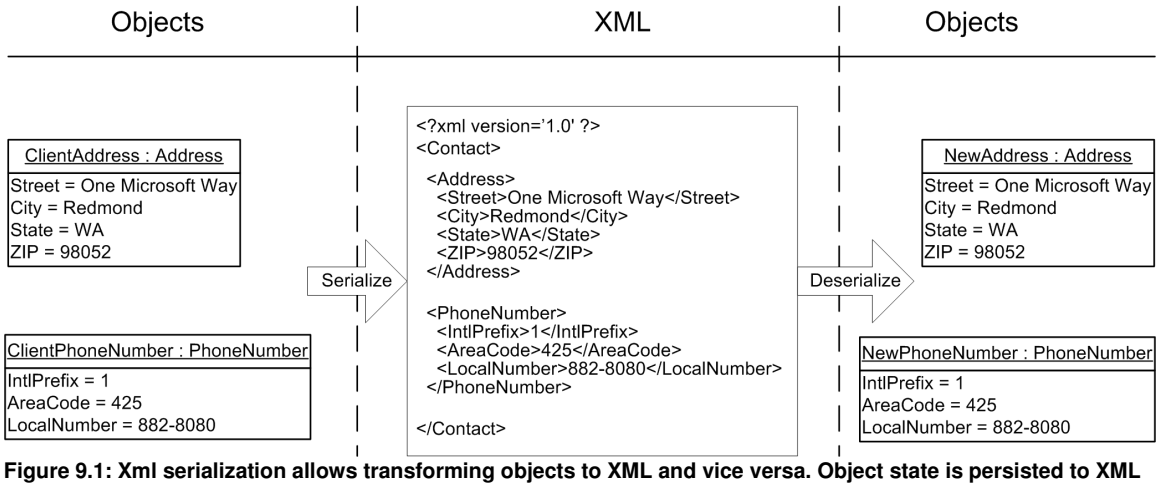


Figure 9.1: Xml serialization allows transforming objects to XML and vice versa. Object state is persisted to XML documents, which are well suited for storage or transmission. The XML documents can then be deserialized back into objects.

In this chapter we will learn about using the .NET framework to serialize objects to an XML-based representation and then deserializing the XML back into objects. You will learn how to develop classes so their XML representation will map to a given XML format. This is a common problem in applications exchanging data through an XML-based format and the .NET Framework provides a powerful solution in the System.Xml.Serialization namespace.

Later in this book we will concentrate on XML Serialization in the context of more distributed applications where we communicate through an XML-based remote procedure call (RPC) protocol named SOAP.

Prerequisites

A basic understanding of two topics is required before we dive right into serialization. The two topics are:

- .NET Metadata Attributes
- XSD schemas.

Metadata Attributes

A complete discussion of Metadata Attributes is beyond the scope of this book. Nevertheless we need to understand the concepts behind metadata attributes to understand how XML serialization works. When we talk about attributes in this section, we will always refer to metadata attributes, not XML attributes.

Attributes are a programming concept first introduced to the Microsoft platform with Microsoft Transaction Sever, which later became COM+. Attributes are annotations to an interface or a class definition to specify certain behavior. For example, no explicit coding was necessary to modify the transactional behavior of a class, it was declared by the presence of a transaction attribute. This is why the concept is also referred to as declarative programming.

The .NET platform takes attributes much further and uses them in a variety of places. Assemblies, classes, fields and methods, each can have attributes. Some are used by the compiler, some are used by the runtime, e.g. to identify a method requires a call to a web service, or how to serialize a class to XML. There is very little overhead associated when using attributes.

Attaching attributes to a class is done directly in the source code. The syntax to initialize a metadata attribute and attach it to a class or a method in C# is either:

```
[Attribute( constructor-parameters-list )]1
```

or:

```
[Attribute(constructor-parameters-list, property=value, … )]
```

This chapter uses the second variation where possible because it is more descriptive and easier to understand.

Now what does all this have to do with XML or serialization? A whole lot! The System.Xml.Serialization namespace introduces a set of attributes to control how classes are mapped to XML. Let's look at a quick example: One of the attributes used with XML serialization is the XmlRootAttribute to change the name of the root element of a serialization hierarchy. You would add the XmlRootAttribute to a class like this:

```
using System.Xml.Serialization;
[XmlRootAttribute(Name="Car", IsNullable=false)]
public class Automobile
{
      // class implementation goes here
}
```

This is as far as we'll go introducing attributes. You now know enough about attributes to use them for serialization. If you are interested to learn more about using attributes throughout the .NET platform you can find some references to more in depth discussions in the further reading section.

Serialize class

The XmlSerializer is your new best friend when it comes to serializing objects to XML documents. It will save you from writing code using with the XmlDocument or XmlTextReader/-Writer classes to save and restore objects. Many of us are probably familiar developing serialization solutions using an XML parser directly, e.g. if you have an application sending the data over the internet. In your application, the data you want to send is lives encapsulated within objects, but you need to serialize the objects in order to send the data. You probably added boiler-pate code to these classes to persist fields to XML using a DOM Document class[2]. If your application also received data in XML format you probably wrote more lines of boiler-plate code to parse the received XML and set the properties on your objects.

Now the XmlSerializer does it all for you. It handles the transformation both ways and, to make life even better, we do not even have to create the classes to serialize ourselves. If we have

[1] The VB.Net syntax identifies attributes through angle brackets (<>) instead of square brackets([]).
[2] The MSXML.DOMDocument if you are a COM developer or the org.w3.dom.Document in Java™

an XML schema describing the data layout of our transfer format we can generate .NET classes corresponding to the complex types in the schema with the Framework's XSD schema definition tool. You can find all about the XSD tool in Appendix C. In the following sections of this chapter we will learn how to use the XmlSerializer, and how we need to design classes so the XmlSerializer can use them.

9.2.1 Round-trip serialization

Let's start out developing a class for the XmlSerializer to process. The garage example document used throughout the first part of this book contained car element nodes with attributes to store some information about a car. That's a good candidate for a class right there.

When we develop .NET classes we no longer need to worry about writing methods to serialize an object to XML. This functionality is built directly into the .NET Framework. Let's start with a very simple Car class. Each Car object needs members for make, model and the year the car was built.

Listing 9.1: example class to use with the XmlSerializer

```
public class Car
{
  public Car(){}
  public string Make;
  public string Model;
  public int Year;
}
```

This class is very straightforward, all the data stored is exposed through public fields. OK, exposing fields like this is generally bad design, but 1) this is an only example and 2) there is a reason why this example class declares its members public. We will hear more about that reason shortly. In chapters 12-14 we will also meet completely different approach to object serialization that in fact can serialize non-public members. Also notice that the class does not contain any methods or derive from any base classes to enable serialization.

Serializing an object

Next, we create an instance of the car class, set the fields and use the XmlSerializer to write the state of the car object to a file:

Listing 9.1: Using the Serialize() method to serialize an object to an XML file.

```
void SerializeACar()
{
  XmlSerializer ser = new XmlSerializer(typeof(Car));
  XmlTextWriter writer = new XmlTextWriter("car.xml",
      System.Text.Encoding.UTF8);
  // write a human readable file
  writer.Formatting = Formatting.Indented;

  Car wifesCar = new Car();
  wifesCar.Make = "Ford";
  wifesCar.Model = "Explorer";
  wifesCar.Year = 1997;
```

```
  ser.Serialize(writer, wifesCar);
  writer.Close();
}
```

With two(!) lines of code you have created an XML representation of the Car object. You instantiated an XmlSerializer for the class you want to serialize. Then you called the Serialize() method with an XmlTextWriter object and the object to serialize and that was it! The XmlTextWriter (discussed in chapter 2) controls where and how the output of the method is written. Imagine how easy to build applications that exchange XML when you can transform your application objects into XML with two lines of code.

The Serialize() method makes use of the pluggable architecture we have seen throughout the discussion of the System.Xml namespace. It accepts any classes derived from System.IO.Stream, System.Xml.XmlWriter or System.IO.TextWriter, thus allowing a great deal of flexibility where to persist objects to. The following listing shows the class and the XML document of a serialized instance side-by-side.

```
public class Car
{
  public Car(){}

  public string Make;
  public string Model;
  public int Year;
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<Car
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance">
  <Make>Ford</Make>
  <Model>Explorer</Model>
  <Year>1997</Year>
</Car>
```

The XmlSerializer produced an element named after the class of the serialized instance. Each field resulted in a child element with the name of the fields. How was the XmlSerializer able to do that? Remember I said I had a reason to make them public? Here it is: Since we declared all fields public, the XmlSerializer can analyze the type of the object and access each field to get its value. Accessing private and protected fields is possible for locally installed applications because they run with unlimited security permissions by default. Applications running from the network or the Internet are not granted those permissions, but the XmlSerializer is designed to work within those types of applications to enable XML-based message exchange for example. Therefore the XmlSerializer has to restrict itself to the serialize the class members it can access even with no special permission settings.

You are probably (rightfully) concerned about exposing public fields because they break encapsulation. The good news is that you do not necessarily have to. XML serialization also works with properties as long as they provide read and write accessor methods. It will not process read- or write only properties because the XmlSerializer makes sure it only processes properties that can be transformed both ways. However, if the order of the serialized elements matters, as it does when your class maps to certain XML schema types, your class must not mix properties and fields. The XmlSerializer does not maintain the order of which fields and properties appear in the class definition. It first maps all the fields to the XML document, then all the properties.

BUG WARNING: Version 1.0 of the .NET Framework had a bug that would corrupt the order of serialized properties if the serialized class was written in Visual Basic.NET. The bug was fixed in Version 1.1.

Deserializing an object

Before we look at the Serialize() method more in depth I will show you that de-serializing an object also works with two lines of code:

Listing 9.1: Using the Deserialize() method to read an XmlDocuement

```
void DeserializeACar()
{
  XmlSerializer ser = new XmlSerializer(typeof(Car));
  XmlTextReader reader = new XmlTextReader("car.xml");
  Car wifesCar = (Car)ser.Deserialize(reader);
  // wifesCar is a 1997 Ford Explorer again
  reader.Close();
}
```

This time we instantiate an XmlTextReader for the XML file we created in the previous example. Then we pass it to the Deserialize() method and the XmlSerializer determines that it 1) needs to instantiate a new Car object, 2) creates a new Car object and 3) sets the Car object's fields to the values supplied by the XmlReader before it returns the new Car object to you. Note that the type cast of the Deserialize() method's return value to Car is necessary because the Deserialize() is declared to return only an object and the assignment would not be valid without the cast.

Imagine you are developing an application which receives large XML documents with nested types and hundreds of nodes. How many lines of code would you have to write to parse these documents and assign their content to an object hierarchy for further processing? With XML data binding you define classes that map to the XML format of the documents you are receiving and let the XmlSerializer do the mapping. Instead or writing hundreds of lines of code you get by writing four. Starting with section 9.2.2 we will learn how put this powerful functionality to work to bind objects to arbitrarily complex XML formats.

Serialize() and Deserialize() methods

Both methods, Serialize() and Deserialize() leverage the concepts of streams and abstract base classes we learned about in chapter 2. Several overloads shown in table 9.1 and table 9.2 are available for both methods to allow dealing with XML from a variety of sources.

Table 9.1: Available overloads for the Serialize() method. Each overloaded method allows different options to control encoding and formatting of the output. Some overloads also allow the declation of XML namespaces used in the created XML document.

Serialize overload	**Purpose**
public void Serialize(Stream stream, object o);	Serialize an object to any kind of stream. The XML written to the stream is always UTF-8 encoded and formatted with indentations.
public void Serialize(Stream stream, object o, XmlSerializerNamespaces namespaces);	Serialize an object to any kind of stream and declare XML namespace prefixes to use throughout the generated MXL document. The XML written to the stream is always UTF-8 encoded and formatted with

	indentations.
public void Serialize(TextWriter writer, object o);	Serialize an object to a TextWriter. The TextWriter controls encoding of the generated XML document. The document is always formatted with indentations.
public void Serialize(TextWriter writer, object , XmlSerializerNamespaces namespaces o);	Serialize an object to a TextWriter and declare XML namespace prefixes to use throughout the generated XML document. The TextWriter controls encoding of the generated XML document. The document is always formatted with indentations.
public void Serialize(XmlWriter writer, object o);	Serialize an object to an XmlWriter. The XmlWriter controls encoding and formatting of the generated XML document.
public void Serialize(XmlWriter writer, object o, XmlSerializerNamespaces namespaces);	Serialize an object to an XmlWriter and declare XML namespace prefixes to use throughout the generated XML document. The XmlWriter controls encoding and formatting of the generated XML document.

The XmlSerializer always operates on the current position of the input or output stream. Using raw streams you can even inject and retrieve XML from any kind of data stream, not only a well-formed XML document.

Table 9.2: The available overloads for the Deserialize() method allow to deserialize objects from a variety of sources: file-based, memory-based and over a network.

Deserialize Overload	**Purpose**
public object Deserialize(Stream stream);	Deserialize an object graph from any kind of stream.
public object Deserialize(TextReader reader);	Deserialize an object graph from a TextReader.
public object Deserialize(XmlReader reader);	Deserialize an object graph from a XmlReader.

Serializable Classes

As the examples demonstrated, simple serialization required no custom code to write or read the XML. We wrote the state of an object to an XML document on disk, read the document back in and turned it into an object entirely by leveraging services provided by the .NET Framework. We did not write any code to generate XML elements or attributes; all this was done behind the scenes. “Where’s the catch?” you may ask. Well, there are quite a few requirements for classes the XmlSerializer can process. The design goal for the XmlSerializer to run in applications with limited permissions is responsible for a number of restrictions on the classes that can be serialized:

- The serialized class must be public, because you cannot analyze internal and private types through without granting certain permissions to the executing application. Yet the XmlSerializer is designed to operate without any special security settings.
- The serialized class must have a default (parameter-less) constructor, because the Deserialize() method needs to be able to instantiate an object in unsafe environments before it can set all the fields. Instantiating an object without calling the constructor would require certain permissions.
- No code executing inside a property accessor may require any security privileges, because the XmlSerializer ensures that it can deserialize the types it processes in unsafe environments.
- Properties must be read/write. Read-only properties will be ignored because they cannot be set without special permissions when an object is deserialized.

- All these restrictions ensure that the XmlSerializer is fully functional even in applications running from the network or even the internet, because their permission sets are very limited.

Unfortunately there are more restrictions on the types the XmlSerializer can process. Some of them are restrictions of the current implementation of the XmlSerializer. Make sure you know about the restrictions below when you develop classes you intend to process with the XmlSerializer:

- Properties and fields may not return interfaces. Abstract base classes are OK.
- Multi-dimensional arrays can not be mapped to XML, You have to use nested arrays instead.
- Object identity is not preserved when an object is serialized with the XmlSerializer. When you serialize an object graph in which an object is referenced from multiple other objects the referenced object is serialized each time it is referenced.
- Type safety is not guaranteed when deserializing an object. You can map serialize and deserialize XML documents with different .NET classes as long as they map to the same XML layout. This is an important feature, since different applications can exchange data without requiring the same classes.
- Object graphs cannot contain circular references, i.e. you cannot serialize constructs like doubly linked lists.
- Collections must not implement IDictionary, like the Hashtable for example

These limitations pose no major difficulties in data-driven applications where the data layout is the primary focus, not object type and identity. Yet they make it very hard to serialize anything else but classes specifically designed for XML data binding scenarios.

There are some classes throughout the .NET framework designed with these requirements in mind. In fact, in some case it is the other way around. The XmlSerializer has built-in support to enable XML data binding for certain types and classes, such as:

- Array types
- Collections, implementing ICollection or IEnumerable, but not IDictionary, for example the ArrayList
- Objects of types derived from XmlNode as discussed in chapter 3.
- DateTime and TimeSpan objects.
- DataSet objects (weak and strongly-typed). DataSets are objects to access data stored in a database. They are discussed in detail in chapter 8. Chapter 10 demonstrates the customized serialization support for DataSet objects.

The .NET Framework offers alternatives to serialize objects if the restrictions do not work in your scenario: The SoapFormatter does not impose some of these limitations and serializes objects into an XML-based format also. It is intended when you use XML and the SOAP protocol to execute code on other servers. The BinaryFormatter also serializes the complete object state, ensures type-safety, but uses a binary format to store the information. Both classes are located in

the System.Runtime.Serialization namespace. We'll get back to see the SoapFormatter in action in chapters 13 and 14.

XML Serialization Attributes

In the previous section we learned how to save and restore an object with the XmlSerializer, yet we haven't actually looked at the output of the Serialize() method. We set out to write a class to obtain an XML representation that would match the car element in the XML document first used in chapter 2; the generated XML was supposed to contain a car element with attributes for make, model and year of the car. Now let's compare the output of the XmlSerializer to the format from chapter 2. The following listing shows the two side-by-side.

XML format From Chapter 2

```
<?xml version="1.0"
  encoding="utf-8"?>
<car
  make="Ford"
  model="Explorer"
  year="1997"
/>
```

XML created by the XmlSerializer

```
<?xml version="1.0"
  encoding="utf-8"?>
<Car …>
  <Make>Ford</Make>
  <Model>Explorer</Model>
  <Year>1997</Year>
</Car>
```

Close, but no cigar! All the data encapsulated by the Car object is present in the XML document, yet the structure is not quite what needed. The output document starts with a Car element following the name of the class of the serialized object. All fields of the class are represented as elements in the generated XML document, but we really wanted them to be attributes. What would we do if we had to send XML to a receiver who can only process the data in attributes? Could we still serialize with the XmlSerializer? Yes, absolutely. In fact that is what the XmlSerializer was designed for: map classes to arbitrary XML formats. The rest of this chapter will show us how this is done.

During the introduction of this chapter we mentioned the important role .NET metadata attributes are playing in XML serialization. Now we will get to know the specific attributes that control how classes are mapped to XML types and we will learn applying them, too. Table 9.3 lists the XML serialization attributes available in System.Runtime.Serialization namespace of the .NET framework. You will find a comprehensive list of their effects on the generated XML format in section 9.6.

Table 9.3 XML Serialization Attributes control Class-To-XML Type Mappings. Each attribute customizes how the XmlSerializer maps a class, field or property to an XML document. The attributes can also declare types that are not explicitly referenced in a source file.

Serialization Attribute	Purpose
`XmlRootAttribute`	Specify a name different from the class name for the root element of a serialization hierarchy
`XmlElementAttribute`	Specify the element name for a field or property. The XmlElementAttribute also omits the array node when it is applied to collections or arrays. The Type property can provide additional type information and limits the customizations of the attribute to a particular class.
`XmlAttributeAttribute`	Generate an attribute rather than an element node for the field or property and specify the XML attribute's name.
`XmlArrayAttribute`	Change the name of an array node
`XmlArrayItemAttribute`	Similar to the XmlElementAttribute. It changes array element names. The Type property can provide additional type information and limits the customizations of the attribute to a

	particular class. Use the NestingLevel property on an array of arrays to specify which level of depth the attribute applies to.
XmlIgnoreAttribute	Ignore a field or property for serialization.
XmlEnumAttribute	Change the name of an enum element
XmlTextAttribute	Serialized a field or property as XML text.
XmlChoiceIdentifierAttribute	Provide additional information to map an xsd:choice to the .NET type system
XmlNamespaceDeclarationAttribute	Declare namespace prefixes to use when serializing or deserializing an object.
XmlIncludeAttribute	Specify related types.
XmlTypeAttribute	Specify the XSD schema type and change the element name
XmlAnyAttributeAttribute	Setup a field or property to receive XML attributes that do not map to any fields or properties of the class. The field or property must be an array of XmlNode or XmlAttribute.
XmlAnyElementAttribute	Setup a field or property to receive XML elements that do not map to any fields or properties of the class. The field or property must be an array of XmlNode or XmlElement.
DefaultValueAttribute	Specifies the XSD default value for a field. The serializer will not serialize the field if it is set to the default value. The receiver has to infer the default value from the schema.

Let's see how which of these attributes we can apply to our Car class to get the desired output from the XmlSerializer. First we attach an XmlRootAttribute attribute to change the name of the root element to "car" with a lower case 'c'. The new the name of the root element is passed to the constructor of the attribute.

Second, we would like to map the fields to attributes rather than elements. We can accomplish this by applying an XmlAttribute to each field. Like the XmlRoot, this attribute also has a constructor which takes the name to use for the generated XML attribute as a parameter.

```
using System.Xml.Serialization;

[XmlRoot3("car")]
public class Car
{
  public Car(){}
  [XmlAttribute("make")]
  public string Make;
  [XmlAttribute("model")]
  public string Model;
  [XmlAttribute("year")]
  public int Year;
}
```

Now when we call Serialize() the XmlSerializer will create an XML document file of the format we wanted:

```
<?xml version="1.0" encoding="utf-8" ?>
<car […]4 make="Ford" model="Explorer" year="1997" />
```

[3] The "Attribute" suffix of the attribute's type name can be omitted in the source code.
[4] The XmlSerializer declares two namespaces by default. We omit namespaces for the remainder of this chapter.

There is another metadata attribute very similar to the two we just heard about so it needs to be mentioned here. Similar to the XmlRoot, the XmlElement maps the member name to another XML element name. While the XmlRoot can only be applied to classes, the XmlAttribute and the XmlElement are only valid on class members. The XmlElement also has some other uses, which we will learn about in 9.4.2 and 9.5.1.

Each attribute offers further properties to fine-tune the how the XmlSerializer maps an item to XML. Table 9.4 shows common properties of many attributes and explains their use. You can also find a complete reference for the metadata attributes and their properties in the appendix.

Table 9.4: Common properties exposed by many Xml serialization attributes. These properties fine-tune the behavior of the attributes. They define XML namespaces, limit the scope of the attribute to a certain type, specify the corresponding type in an XSD schema, control the form of the generated XML and define whether or not the create XML for null references.

Property	Description	Applies To
`DataType`	Specifies the XSD data type of the item, as defined by the W3 consortium. The DataType will be read by the XSD Schema tool when generating an XSD schema from the class.	`XmlRootAttribute`, `XmlTextAttribute`, `XmlElementAttribute`, `XmlAttributeAttribute`, `XmlArrayItemAttribute`
`Namespace`	Specifies which XML namespace the generated item belongs to. A namespace declaration will be added if the namespace is not already declared in the scope of the item.	`XmlArrayAttribute`, `XmlArrayItemAttribute`, `XmlAttributeAttribute`, `XmlElementAttribute`, `XmlRootAttribute`, `XmlTypeAttribute`
`Type`	Specifies which .NET types an attribute instance applies to. Used when a property or field can hold different types, e.g. inherited types	`XmlElementAttribute`, `XmlAttributeAttribute`, `XmlArrayItemAttribute`, `XmlIncludeAttribute`, `XmlTextAttribute`
`Form`	Specifies whether to treat an item as qualified or unqualified. Works with the Namespace property.	`XmlArrayAttribute`, `XmlArrayItemAttribute`, `XmlAttributeAttribute`, `XmlElementAttribute`
`IsNullable`	Specifies whether or not the XmlSerializer will generate an xsi:nil="true" attribute for fields and properties that are set to null	`XmlArrayAttribute`, `XmlArrayItemAttribute`, `XmlElementAttribute`, `XmlRootAttribute`

XmlSerializer

In the previous example, the metadata attributes overrode the XmlSerializer's default formatting, but serialization worked just fine before we attached any attributes. In some cases, however, attributes are required for the XmlSerializer to produce any output. The next three sections will show us why.

Before we go on learning about applying attributes, let's take a moment to understand what's going on inside the XmlSerializer. All the XmlSerializer constructors expect to receive some information about the types of objects they are going to serialize over their lifetime. Most overloads require passing in a type object directly. One of them expects the type information in a

pre-processed format, the XmlTypeMapping, but we ignore that one because it is intended to support ASP.NET WebServices, and is not for public consumption.

1.1 Table 9.3 The different overloads of the XmlSerializer constructor require specifying types the serializer instance will process and offers several options to customize the default class-to-XML-type mappings.

Constructor	Description
public XmlSerializer(Type *type*);	Constructs an XmlSerializer that can process objects of type *type*
public XmlSerializer(XmlTypeMapping *xmlTypeMapping*);	Constructs an XmlSerializer that can process objects described by the *xmlTypeMapping*.
public XmlSerializer(Type *type*, string *defaultNamespace*);	Constructs an XmlSerializer that can process objects of type *type* and defines the default XML namespace for all processed types.
public XmlSerializer(Type *type*, Type[] *extraTypes*);	Constructs an XmlSerializer that can process objects of type *type* and the types in *extraTypes*.
public XmlSerializer(Type *type*, XmlAttributeOverrides *overrides*);	Constructs an XmlSerializer that can process objects of type *type* and applies the XML serialization attributes to customize the class to XML type mappings.
public XmlSerializer(Type *type*, XmlRootAttribute *root*);	Constructs an XmlSerializer that can process objects of type *type* and specifies the properties of the root nodes of the serialized objects.
public XmlSerializer(Type *type*, XmlAttributeOverrides *overrides*, Type[] *extraTypes*, XmlRootAttribute *root*, string *defaultNamespace*);	Constructs an XmlSerializer that can process objects of type *type* and the types in *extraTypes*, applies the XML serialization attributes to customize the class to XML type mappings, specifies the properties of the root nodes of the serialized objects and defines the default XML namespace for all processed types. Parameters except type can be null.

The constructors use the reflection features of the .NET framework to analyze a type's public fields and properties and then store the type's structure, i.e. fields field types and metadata attributes, in a type mapping. By default the XmlSerializer maps each field or property to an XML element with the same name, unless it finds an attribute attached to the field to change the element's name or map the field to an XML attribute. In those cases it modifies the type mapping according to the information found in the attribute.

> NOTE: It is good practice to always provide XML serialization attributes to explicitly declare corresponding element and attribute names, instead of relying on the default mapping. Without explicit mapping directives modifications to the source code might affect the serialization format, but if we declare the serialization format explicitly we can protect ourselves from problems caused by inadvertent source code changes.

These mappings are then processed into on-the-fly generated classes which are compiled into a temporary assembly to make serialization and deserialization very fast. Extracting the type information and processing it and compiling the temporary assembly, on the other hand, is a very computing intensive operation. When we design applications with the XmlSerializer, we should try to instantiate it only once and keep it around for the lifetime of an application to minimize the performance hit when we instantiate the serializer and maximize the performance gain of the cached type information.

- During serialization the serializer will query each object for its type, check the cache for a corresponding type mapping and persist the object in the format defined by the mapping. Serialize() throws an exception if it cannot locate a mapping for the exact type.
- Deserialization works just the opposite. The Deserialize() method will check the XML stream for content to identify types to instantiate. If no matching type mapping is found Deserialize() will throw an exception.

So much for the theory! How do we tell the serializer to create type mappings for types that are not directly declared? By attaching metadata attributes, of course. We can attach attributes to declare types in two places:

- On a class: Each class can declare substitute types for the type the attribute is attached to. Substitutes are typically derived classes which can occur at run-time instead of the base class.
- On a member: We can specify the types that might by assigned to a field at runtime and even customize the XML mappings dependent on the type. We can take advantage of these type declarations if we can either not provide information about substitute types at the class or if we want finer-grained control over the XML format.

XmlIncludeAttribute

An XmlInclude attribute attached to a class will let the serializer know about derived classes. The XmlSerializer cannot handle situations where an object of a derived type occurs if a base type was declared without some help from us, regardless whether the derived type occurs at the root or somewhere else in the serialized object graph. The XmlSerializer needs to know about types derived from a class at the time it does its class analysis, i.e. when the constructor runs. If we do not explicitly declare the derived type together with the base type the constructor is not able to locate the derived type on its own and cannot create and process a type mapping for the derived type.

Applying this attribute is very useful when you are developing your own class library that maps to a hierarchy of XML schema types, where a derived type can also replace a base type. Take the following XSD snippet for example, which defines a base type Vehicle and two derived type Car and Motorcycle:

```
<!-- base type -->
<xs:element name="Vehicle" nillable="true" type="Vehicle" />
<xs:complexType name="Vehicle">
  <xs:sequence>
  <xs:element minOccurs="1" maxOccurs="1" name="Make"
      type="xs:string" />
  <xs:element minOccurs="1" maxOccurs="1" name="Model"
      type="xs:string" />
    <xs:element minOccurs="1" maxOccurs="1" name="Year" type="xs:int" />
 </xs:sequence>
</xs:complexType>
<!-- 1st derived type -->
<xs:complexType name="Motorcycle">
  <xs:complexContent mixed="false">
```

```
    <xs:extension base="Vehicle">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<!-- 2nd derrived base type -->
<xs:complexType name="Car">
  <xs:complexContent mixed="false">
    <xs:extension base="Vehicle">
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
<xs:element name="Car" nillable="true" type="Car" />
<xs:element name="Motorcycle" nillable="true" type="Motorcycle" />
```

Any element in this schema of type Vehicle could also contain a Car or a Motorcycle type, because Car and Motorcycle are extensions of Vehicle – just like we can replace a base class with a derived class in a class hierarchy in an object-oriented programming environment. We can easily map this hierarchy to a set of the .NET classes. We need a base class Vehicle and two derived classes Car and Motorcycle, like the ones in listing 9.4.

Listing 9.1: Vehicle class hierarchy with the XmlInclude attribute

```
[XmlInclude(typeof(Car))]
[XmlInclude(typeof(Motorcycle))]
public class Vehicle       | #1
{
  public Vehicle(){}
  public string Make;
  public string Model;
  public int Year;
}
public class Car : Vehicle | #2
{
  public Car() {}
  public string VIN;      | #3
}
public class Motorcycle : Vehicle  | #4
{
  public Motorcycle() {}
}
```

(annotation) <#1 The base class of the hierarchy>
(annotation) <#2 The first class derived from Vehicle>
(annotation) <#3 The Car class adds an additional field>
(annotation) <#4 Another class derived from Vehicle>

We have to annotate the Vehicle class with an XmlInclude attribute for each of the derived classes, if we want the XmlSerializer to be able to serialize Car and Motorcycle objects where a Vehicle was declared.

NOTE: The XmlInclude attribute works recursively, i.e. the serializer will also include types if the XmlInclude attribute was attached to a class that was included because of a class that was already included through an XmlInclude attribute.

With the XmlInclude attributes in place an XmlSerializer will know how to serialize and deserialize Car and Motorcycle objects instead of a Vehicle object, because the XmlSerializer found the XmlInclude attributes, analyzed the types Car and Motorcycle and made a reference that each of those are valid substitutes for the Vehicle type. To prove it we can create an XmlSerializer for the Vehicle type and use it to serialize a Car object:

```
Car wifesCar = new Car();
…
XmlSerializer xs = new XmlSerializer(typeof(Vehicle));
xs.Serialize(xw, wifesCar);
```

The code doesn't throw an exception, that's already an indication that everything went as expected. Serialize() identified the type of the wifesCar object, found a type mapping for the Car type and wrote the following XML to the output:

```
<Vehicle
  xmlns:xsd=http://www.w3.org/2001/XMLSchema
  xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
  xsi:type="Car">
  <Make>Ford</Make>
  <Model>Explorer</Model>
  <Year>1997</Year>
  <VIN>1234</VIN>
</Vehicle>
```

The name of the root element indicates that the serialized object was declared to be of type Vehicle. In this case the XmlSerializer expects a Vehicle object as the serialization root, because we passed Vehicle's Type to the constructor. The actual type of the serialized object is stored in an xsi:type attribute, for deserialize to determine that the serialized object does not conform to the declared type. The type attribute belongs to the XSD schema-instance namespace and declares an element's XSD schema type. Here the type attribute indicates that the element is of type Car. Also note that all fields of the Car class are present in the XML output, not only the members of the Vehicle class.

> NOTE: The Deserialize() method will read xsi:type attributes and throw an exception if it does not find a matching mapping for the type. Without the xsi:type attribute it would deserialize the XML document to a Vehicle object and ignore the VIN element.

XmlElement Attribute Type

Attaching attributes to individual class members is another way to declare additional types. This can be useful when it is impossible to attach an XmlInclude attribute to a class to declare derived classes, but we can declare the derived classes on the field that is referencing the base class.

In the next example we create a new class VerySmallParkingLot with a field of the Vehicle base class from the previous section. We attach two XmlElement attributes to the field, one for each type we expect the member to reference at runtime. The class and its XML counterpart are shown above. Besides declaring the Car type for the field, the XmlElement attribute also causes

the XmlSerializer to identify the object type through the element name instead of an xsi:type attribute.

```
public class VerySmallParkingLot
{
  public VerySmallParkingLot (){}
  [XmlElement(
    Type=typeof(Car))]
  [XmlElement(
    Type=typeof(Motorcycle))]
  public Vehicle ParkedVehicle;
}
```

```
<VerySmallParkingLot …>

  <Car>
   <Make>Ford</Make>
    <Model>Explorer</Model>
    <Year>1997</Year>
    <VIN>1234</VIN>
  </Car>
</VerySmallParkingLot>
```

The more prevalent use-case where we need to define types through multiple XmlElement attributes is when we develop a class mapping to an XML type containing which can contain different child elements. We express this flexible setup with the <choice> model group in an XSD schema. Generally, a model group defines usage rules for a group of elements, or in XML terminology: Particles. A model group defines which particles can occur in a group, in which order they occur in and how many times. The <choice> model group in particular defines a group of mutually exclusive particles. The next XML schema snippet shows a type description for an XML type roughly corresponding to the VerySmallParkingLot class from above. Each instance of the XML type VerySmallParkingLot, can either contain a ParkedMotorcycle element of type Motorcycle or a ParkedCar element of type Car. The difference to the class above is that the element names vary with the element type because the choice model group requires unique names for each particle.

Listing 9.5 An XML type definition with the choice model group

```
<xs:complexType name="VerySmallParkingLot">
 <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element minOccurs="0" maxOccurs="1" name="ParkedMotorcycle"
        type="Motorcycle" />
      <xs:element minOccurs="0" maxOccurs="1" name="ParkedCar"
        type="Car" />
  </xs:choice>
  </xs:sequence>
</xs:complexType>
```

But specifying the element names for each particle is no big deal to XML serialization, as long as the each particle's type is unique. You simply specify the element name for each type by setting the ElementName property of the XmlElement attribute as shown below.

```
public class VerySmallParkingLot
{
  public VerySmallParkingLot (){}
  [XmlElement(
    ElementName="ParkedCar",
    Type=typeof(Car))]
  [XmlElement(
    ElementName="ParkedMotorcycle",
```

```
<VerySmallParkingLot …>

  <ParkedCar>
   <Make>Ford</Make>
    <Model>Explorer</Model>
    <Year>1997</Year>
    <VIN>1234</VIN>
  </ParkedCar>
```

```
    Type=typeof(Motorcycle))]
  public Vehicle ParkedVehicle;
}
```

```
</VerySmallParkingLot>
```

If you are interested in the more intricate case with more than one particle of the same type, you can find a detailed discussion in chapter 10.

Serializing Arrays

Until now we only considered scalar members, but what about arrays and collection types? The XmlSerializer will process them just fine, as long as it created type mappings for the types inside the collection or the array. In section 9.4 we learned how to declare additional types for scalar fields. The next sections will show us how to declare additional types for arrays and collections. We will also read about attributes to customize XML mappings for arrays and collections.

Serializing arrays works just like serializing scalar objects. Everything we've learned so far about declaring types and customizing type mappings applies just the same. We can pass an array type to the XmlSerializer constructor, call Serialize() or Deserialize() and the XmlSerializer will process an array just like it would process a scalar object:

```
XmlSerializer xs = new XmlSerializer(typeof(Vehicle[]));
Vehicle[] vehicles = new Vehicles[2];
// …
Serializer.Serialize( writer, vehicles );
```

A Vehicle array with the classes from the previous section serialized to XML would result in a structure like this:

```
<ArrayOfVehicle>
  <Vehicle xsi:type="Car">…</Vehicle>
  <Vehicle xsi:type="Motorcycle">…</Vehicle>
</ArrayOfVehicle>
```

The serializer added a parent node around all the elements to group all array elements, otherwise the output would not be well-formed XML. The objects inside the array were serialized just like they were in section 9.4.2, because the declared type of the array items was Vehicle, but in this case the items were of type Car and Motorcycle. If a class defines a field of an array type this parent node is not created automatically.

Customizing the XML layout for arrays

There are a number of options to customize how the XmlSerializer maps array types to XML types. Annotating a field of an array type with a plain XmlElement attribute, without a type or element name specification, causes the array elements to be serialized as direct children of the class root. No additional parent node is inserted around the items in the array. Just like we did with scalar members, we can also specify as many instances of the XmlElement attribute if we are mapping to a choice model group instead of a simple sequence.

The next listing shows a ParkingLot class with an array for two Vehicles and the XML structure of a serialized instance. Again, just like we did for a scalar Vehicle field in 9.4.3 we

attach an XmlElement attribute for each type derived from Vehicle. The XML layout for this ParkingLot class is also described by a <choice> model group. In contrast to the schema in listing 9.5 the maxOccurs attribute for the group would be greater than 1.

```
public class ParkingLot
{
  public ParkingLot ()
  {
    Vehicles = new Vehicle[2];
  }
  [XmlElement(
    ElementName="ParkedCar",
    Type=typeof(Car))]
  [XmlElement(
    ElementName="ParkedMotorcycle",
    Type=typeof(Motorcycle))]
  public Vehicle[] Vehicles;
}
```

```
<ParkingLot …>
<!-- no extra element here -->
  <ParkedCar>
   <Make>Ford</Make>
    <Model>Explorer</Model>
    <Year>1997</Year>
    <VIN>1234</VIN>
  </ParkedCar>
  <ParkedMotorcycle>
    <Make>Aprilla</Make>
    <Model>Mille R</Model>
    <Year>2000</Year>
  </ParkedMotorcycle>

</ParkingLot>
```

OK, now we know how to omit a parent element around array items and how to customize the array items themselves. But what if we want to customize element names and still want that intermediate node around the array items?

That's what the XmlArray attribute and the XmlArrayItem attributes are for. The former creates a parent node around the array items and optionally specifies the node's name, the latter is similar to the XmlElement attribute, but provides additional functionality specific to arrays. One use of the XmlArray attribute is mapping a field to an element like ParkedVehicles in the following schema, which can contain zero or more Vehicle elements.

```
<xs:complexType name="AnotherParkingLot">
 <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="1" name="ParkedVehicles"
      type="ArrayOfVehicle" />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="ArrayOfVehicle">
  <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="unbounded" name="Vehicle"
      nillable="true" type="Vehicle" />
  </xs:sequence>
</xs:complexType>
```

The XmlArray attribute in combination with the XmlArrayItem attribute allows us to control the element names of the array items. The XmlArrayItem attribute also allows us to write classes mapping to more complex XSD constructs like a choice model group which can occur more than once. With a schema definition like the one below the ParkedVehicles element can contain any number of ParkedCar and ParkedMotorcycle elements in any order.

```
<xs:element name="BigParkingLot" nillable="true" type="BigParkingLot" />
```

```
<xs:complexType name="BigParkingLot">
  <xs:sequence>
    <xs:element minOccurs="0" maxOccurs="1" name="ParkedVehicles"
      type=" ArrayOfMixedVehicles " />
 </xs:sequence>
</xs:complexType>
<xs:complexType name="ArrayOfMixedVehicles">
  <xs:choice minOccurs="0" maxOccurs="unbounded">
    <xs:element minOccurs="1" maxOccurs="1" name="ParkedMotorcycle"
      nillable="true" type="Motorcycle" />
    <xs:element minOccurs="1" maxOccurs="1" name="ParkedCar"
      nillable="true" type="Car" />
  </xs:choice>
</xs:complexType>
```

The following code listing shows a class, which maps to the schema above. The XmlArray attribute on the Vehicles field changes the name of the array's parent node from the default Vehicles to "ParkedVehicles" as required by the schema. Since XmlElement and XmlArray attributes are mutually exclusive we have to resort to the XmlArrayItem attribute to declare the different choice particles to the XmlSerializer. We have to describe each choice particle with an XmlArrayItem attribute.

1 Listing 9.6: An application of theXmlArray attribute and the XmlArrayItem attribute

```
public class ParkingLot
{
  public ParkingLot ()
  {
   ParkedVehicles = new Vehicle[2];
  }
  [XmlArrayItem (
    ElementName="ParkedCar",
    Type=typeof(Car))]
  [XmlArrayItem (
    ElementName="ParkedMotorcycle",
    Type=typeof(Motorcycle))]
  [XmlArray(
    ElementName="ParkedVehicles")]

  public Vehicle[] Vehicles;
}
```

```
<ParkingLot …>
  <ParkedVehicles>
    <ParkedCar>
      <Make>Ford</Make>
      <Model>Explorer</Model>
      <Year>1997</Year>
      <VIN>1234</VIN>
    </ParkedCar>
    <ParkedMotorcycle>
      <Make>Aprilla</Make>
      <Model>Mille R</Model>
      <Year>2000</Year>
    </ParkedMotorcycle>
  </ParkedVehicles>

</ParkingLot>
```

Providing type information for array elements

The XmlSerializer needs to know about the type of all items inside an array to correctly map them to XML. If an array contains an element of a type that was not specified either by an XmlInclude attribute or an XmlElement attribute the serializer will throw an exception. In addition to these two attributes we can declare additional types by attaching XmlArrayItem attributes with the Type property set.

Serializing Collection Classes

The previous section showed us how to annotate source files with metadata attributes to customize how arrays are mapped to XML. Now we will see if our knowledge transfers to collection classes. These classes are very similar to arrays but don't require a fixed size, can hold unrelated types and are optimized for different usage scenarios. The .NET Framework provides a number of collection classes ready to use in the System.Collections namespace, for example an ArrayList, a Dictionary or a Hashtable just to name a few.

The XmlSerializer can process all collections that implement one of the .NET framework's collection interfaces: IEnumerable or ICollection. All collections provided by the .NET framework implement these interfaces, so we can serialize or deserialize these collections without much additional work. All of the framework's collections are weakly typed, so we have to declare the types stored inside a collection before the XmlSerializer can correctly process it. Declaring types inside a collection works just like declaring types inside an array, which we learned in section 9.5.1.

Customizing the XML layout for a collection

Customizing how a collection is mapped to XML is very much like customizing an array. Attaching the XmlArray attribute and the XmlArrayItem attribute to a collection class has the same effect as attaching them to arrays. Let's confirm this and replace the Vehicle array in the ParkingLot class from the previous example with the more flexible System.Collections.ArrayList.

2 Listing 9.7 Use of XmlArray attributes with a collection class

```
public class ParkingLot
{
  public ParkingLot ()
  {
   ParkedVehicles = new
    ArrayList();
  }
  [XmlArrayItem(
    ElementName="ParkedCar",
    Type=typeof(Car))]
  [XmlArrayItem(
    ElementName="ParkedMotorcycle",
    Type=typeof(Motorcycle))]
  [XmlArray(
    ElementName="ParkedVehicles")]
  public ArrayList ParkedVehicles;
}
```

```
<ParkingLot …>
  <ParkedVehicles>
    <ParkedCar>
      <Make>Ford</Make>
      <Model>Explorer</Model>
      <Year>1997</Year>
      <VIN>1234</VIN>
    </ParkedCar>
    <ParkedMotorcycle>
      <Make>Aprilla</Make>
      <Model>Mille R</Model>
      <Year>2000</Year>
    </ParkedMotorcycle>
  </ParkedVehicles>

</ParkingLot>
```

Even though we are now using an ArrayList instead of an array the XML representation of the new ParkingLot class has not changed.

This may seem odd to those of us already familiar with the ArrayList collection. The ArrayList exposes seven public properties; none of these properties is serialized. The public properties are considered as auxiliary information only, so whenever the Serialize() method detects an object that implements ICollection or IEnumerable it will ignore the object's public properties. Instead it

uses the interfaces to serialize all the items inside the collections. This again reminds us that the intended use of the XmlSerializer is in data-driven environments where the data and its XML representation are the primary focus, not a 100% accurate snapshot of an object.

Developing custom collections

In many cases we might like to extend the collection classes provided by the framework. Maybe we need to ensure that only objects of certain types are stored in the collection or we need a different sorting algorithm. Either way, custom collection classes will de-/serialize properly as long as they implement IEnumerable or ICollection and, of course, we supply enough information about the types stored inside the collection. Keep in mind that only the items that can be accessed through the interfaces are serialized, public properties and fields are not, unless they return a class that itself implements ICollection.

There is another caveat when you implement your own container classes: The implementation of the XmlSerializer requires the collection to have a default accessor, even though ICollection does not require it. In VB.Net a default accessor is implemented as an Item property with a single parameter of type Integer. In C# it is implemented as an indexer. The syntax for an indexer resembles a read-only property, but it uses the square brackets around the parameter. Listing 9.8 gives an example for a strongly-typed collection based on an ArrayList with a default accessor:

3 Listing 9.8 Custom collection with a default indexer.

```
public class CarArray : ICollection
{
  public CarArray() { Cars = new ArrayList(); }
  // ICollection public properties
  public int Count { get { return Cars.Count; } }
  public bool IsSynchronized { get { return Cars.IsSynchronized; } }
  public object SyncRoot { get { return Cars.SyncRoot; } }
  // ICollection public method
  public void CopyTo( Array a, int i ){ Cars.CopyTo( a, i ); }

  // IEnumerable
  public IEnumerator GetEnumerator()
  {
    return Cars.GetEnumerator();
  }

  // the default indexer
  public Car this[int i]                |#1
  {                                     |
    get                                 |
      {                                 |
        return (Car) Cars[i];           |
      }                                 |
  }                                     |

  private ArrayList Cars;              |#2
  // only add Cars and derived classes to the arraylist
  public void Add( Car c ){ Cars.Add( c ); }
```

```
}
```
(annotation) <#1 The default indexer is used to enumerate through the elements of an array or collection. The square brackets denote an indexer in C#.>
(annotation) <#2 The field to store the collection data has to be private. Because the ArrayList implements ICollection. >

NOTE: No metadata attributes were attached to the ArrayList member in the example above. The XmlSerializer can process objects of the type returned by the default accessor without attaching any additional attributes. Any other types need to be declared through an XmlInclude, an XmlElement or an XmlArrayItem attribute.

Run-time exceptions

The XmlSerializer throws exceptions to indicate all sorts of problems. In most cases an exception handler will catch a System.InvalidOperationException thrown in Serialize() or Deserialize(), which makes the StackTrace property useless because it does not offer any more insight into the root cause of the exception. To make matters worse, the exception's Message property only yields very generic information. If we are trying to serialize an undeclared type for example the Serialize() method would throw an exception with the following message:

There was an error generating the XML document.

This message is annoying at best, because we already figured that much when we saw an exception and doesn't help us troubleshooting the problem. The trick to get to the "real information" about the problem is to examine the exception's InnerException property, which contains very detailed information about the problem and where it occurred.

The InnerException's message is usually very descriptive, pinpoints the problem and, in many cases, even offers a possible solution. When we are trying to serialize an undeclared type the InnerExcpetion reads something like this:

```
The type XmlSerializationApp.XmlCar was not expected. Use the XmlInclude
or SoapInclude attribute to specify types that are not known statically.
```

The following listing demonstrates how to set up the exception handler and how to access the InnerException property.

4 Listing 9.9 Handling an Exception from the XmlSerializer and displaying the embedded information

```
using System;

public static ParkingLot DeserializeParkingLot( XmlReader reader )
{
  ParkingLot lot = null;
  try
  {
    XmlSerializer ser = new XmlSerializer( typeof(ParkingLot));
    lot = (ParkingLot) ser.Deserialize( reader );
  }
  catch( Exception ex )                |#1
  {                                    |
    DumpException( ex );               |
  }                                    |
```

```
  return lot;
}
public static void DumpException( Exception ex )
{
  WriteExceptionInfo( ex );                        |#2
  if( null != ex.InnerException )                  |
  {                                                |
    WriteExceptionInfo( ex.InnerException );       |
  }
}
public static void WriteExceptionInfo( Exception ex )
{
  Console.WriteLine( "--------- Exception Data ---------" );          |#3
  Console.WriteLine( "Message: {0}", ex.Message );                     |
  Console.WriteLine( "Exception Type: {0}", ex.GetType().FullName );   |
  Console.WriteLine( "Source: {0}", ex.Source );                       |
  Console.WriteLine( "StrackTrace: {0}", ex.StackTrace );              |
  Console.WriteLine( "TargetSite: {0}", ex.TargetSite );               |
}
```

(annotation) <#1 Catch all exceptions>
(annotation) <#2 Display the properties for the exception and its InnerException>
(annotation) <#3 Display all properties with information about the problem>

XmlSerializer Attributes

We have seen many how options we have to tailor a .NET class to the format of an XML type. If you are developing an application to bind XML data in a format described by an XML schema then you are in luck because you can create the classes corresponding to the schema types with the XSD tool discussed in Appendix C. However, there are quite a few alternative format description languages for XML out there that the XSD tool cannot convert into classes, DTDs and Relax-NG for example. If you need to develop classes to map types from schema formats other than XSD and XDR then this section is for you! It is intended to provide quick answers to "What attribute do I need to map this code to my XML"-type questions when you have to develop serialization classes "by hand". The left column shows C# code fragments with serialization attributes for class definitions, the right column shows how the XmlSerializer maps this code construct to XML. Remember that the attributes' properties are optional in most cases.

1.2 Table 8.1 Metadata attributes control how the XmlSerializer maps classes to XML documents. Each attribute allows further customization of the XML format through its properties.

Code with Metadata Attributes	XML format
[XmlRoot(ElementName="Automobile", **Namespace="urn:my-ns",** **IsNullable="true"))]** public class Car { … }	**<Automobile xmlns="urn:my-ns">** … **</Automobile>**
public class Car { **[XmlElement(ElementName="CarMake",** **Namespace="urn:my-ns",** **IsNullable="true"]** public string Make; … }	<Car> **<CarMake xmlns="urn:my-ns"></CarMake>** … </Car> *or if Make = null* <Car> **<CarMake xsi:nil="true" />** </Car>
Public class ParkingLot { **[XmlElement]**	<ParkingLot> <! -- no parent for the array elements - ->

```
  public Car[] Cars;
  public Car[] MoreCars;
}
```

```
<Car></Car>
<! - - enclosing element for the array elements - - >
<MoreCars>
  <Car></Car>
  <Car></Car>
</MoreCars>
</ParkingLot>
```

```
public class Car {
  [XmlAttribute(
    AttributeName="CarMake",
    Namespace="urn:my-ns")]
  public string Make;
… }
```

```
<Car n1:CarMake="" xmlns:p1="urn:my-ns">
…
</Car>
```

```
public class Car {
   [XmlIgnore]
   public string Make
… }
```

```
<Car>
…
</Car>
```

```
public class Car {
  public string Make = "Ford";
  [XmlText]
  public string Desc = "sedan";
}
```

```
<Car>
<Make>Ford</Make>
sedan
</Car>
```

```
public class ParkingLot {
  [XmlArray(
    ElementName="ParkedCars",
    Namespace="urn:my-ns",
    IsNullable="true")]
  public Car[] Cars;
}
```

```
<ParkingLot>
  <ParkedCars xmlns="urn:my-ns">
    <Car>…</Car>
    <Car xsi:nil="true" />
  </ParkedCars>
</ParkingLot>
```

```
public class ParkingLot {
  [XmlArrayItem(
    ElementName="ParkedCar",
    Namespace="urn:my-ns",
    IsNullable="true")]
  public Car[] Cars;
… }
```

```
<ParkingLot>
  <Cars>
    <ParkedCar><ParkedCar>
    <ParkedCar xsi:nil="true"/>
  </Cars>
</ParkingLot>
```

```
public enum Makes {
  [XmlEnum("FCar")]
  Ford,
  Toyota
}
public class Car {
   public Makes Make;
}
```

```
If Make == Makes.Ford :
<Car>
  <Make>FCar</Make>
</Car>
If Make == Makes.Toyota :
<Car>
  <Make>Toyota</Make>
</Car>
```

```
public class Car
{
  [XmlAnyElement]
  XmlElement[] extraElements;
}
```

```
extraElements contains 2 nodes: Color and Wheels:
<Car>
  <Color>Red</Color>
  <Wheels>Alloy</Wheels>
</Car>
```

```
public class Car
{
  [XmlAnyAttribute]
  XmlAttribute[] extraAttribs;
}
```

```
ExtraAttribs contains 2 attributes: vin and miles
<Car vin="12335" miles="123">
</Car>
```

```
public class Car
{
  [DefaultValueAttribute("Ford")]
  public string Make;
…
}
```

```
If Make == "Ford":
<Car>
  <!- -  default has to be derived from the schema -->
</Car>
Otherwise:
<Car>
  <Make>Toyota</Make>
</Car>
```

```
public class Car {
  [XmlNamespaceDeclarations]
```

```
namespaces maps the prefix c to "urn:christoph-cars"
```

```
  public XmlSerializerNamespaces
    namespaces;
…
}

[XmlType(
  TypeName="Car_T",
  Namespace="urn:christoph-car")]
public class Car
{
  public string Make;
}
```

```
<Car
  xmlns:c ="urn:christoph-cars">

…
</Car>
<Car_T>
  <Make xmlns="urn:christoph-car">
  </Make>
</Car_T>
```

Advanced XmlSerializer

The .NET framework offers more ways to process XML than just parsing it. The System.Xml.Serialization namespace contains classes to create an XML representation for an object or initialize an object directly from XML. The focus of this of this chapter was on developing classes with an XML representation conforming to the XSD standard, which is widely used to describe XML formats in data exchange scenarios.

Using XML serialization will reduce the amount of code you have to develop for an XML-based data exchange application. You no longer have to parse XML to initialize objects, neither do you have to develop code for objects to persist themselves to XML. Once you defined what the XML format you use to exchange data looks like, you can quickly develop classes that can automatically store their data to the XML format or objects can be automatically created from XML.

Besides the runtime support for serialization, the .NET Framework also supplies a tool to generate source code for serializable classes from XSD schemas. Once you start developing solutions with XSD schemas and XML serialization make sure you read the chapter read about the XSD schema definition tool chapter in the appendix.

Previously we introduced XML serialization. We learned how classes are mapped to XML when we transform them with the XmlSerializer. We can use the techniques we learned to quickly develop an XML-driven application and get the best of both worlds: Objects for programming and XML for data transfer or storage. The XmlSerializer does the transformations from one representation to the other.

This chapter will cover some advanced techniques for customizing output of the XmlSerializer. We focused on developing classes to match an XML format from scratch. Unfortunately there are some scenarios where this is not good enough. Real world projects usually include classes where we can not just go and attach an attribute when the class does not exactly map to the XML format we need, so we will learn how to customize XML serialization at run-time. We will also learn to manage namespaces in the serialized XML and how to inject and retrieve XML that does not map to any class members.

This chapter is the “advanced” chapter on XML serialization, so you need to be familiar with the basics: How to attach metadata attributes to classes in order to customize their XML representation as well as some other concepts explained in the previous chapter. You also need to be familiar with:

- The XmlNode class hierarchy
- Events and delegates.
- XML Namespaces

Events and Delegates

Events and Delegates are a programming concept heavily used throughout the .NET programming model. They provide an object-oriented and type-safe model to register call-back methods. When a class exposes an event of a delegate type other classes can bind event handlers to the event to receive event notifications. If you're not already familiar with the use of .NET events you can find a deeper discussion on event and delegates in the .NET Framework SDK documentation.

XML Namespaces

Often applications need to add new information to existing XML documents or combine existing XML documents. To avoid naming conflicts in these scenarios ("from which document is the <description> element?"), the W3 consortium standardized XML namespaces. You can think of a namespace as a last name for elements and attributes. Calling for somebody in a crowd just by their first name might cause many people to respond, if you call for somebody by their first and last name you can address the right person. XML namespaces work the same way: the first name is an XML element or attribute; the last name is the namespace URI. Using both, you can uniquely identify attributes and elements in an XML document.

If you are not yet familiar how the classes in System.Xml support namespaces you can find an introductory discussion in chapter 3.

Customizing Xml Serialization

Previously we covered developing classes containing metadata attributes that the XmlSerializer maps to an XML type. This is great when we develop classes ourselves, but we can not apply this technique when we are in not in control of the source code. But does that mean we are out of luck when we need to customize how classes from third-party libraries map to XML or when the mappings defined by a class need modification?

No, it does not. Fortunately there are alternatives to attaching attributes in code. There are several overloads for the constructor of the XmlSerializer to customize type mappings at runtime. Table 10.1 shows the available overloads. These constructors can customize type mappings and declare additional types to the XmlSerializer.

1.3 Table 10.1: All available overloads for the constructor of the XmlSerializer class. Each overload allows customizing the serialization at run-time

Constructor Signature	Description
`XmlSerializer(Type type);`	`Create an XmlSerializer for type`
`XmlSerializer(XmlTypeMapping mapping);`	Provide a custom type mapping to use for serialization
`XmlSerializer(Type type, string defaultNamespace);`	Create an XmlSerializer for `type` and specify the default namespace for all serialized objects
`XmlSerializer(Type type, Type[] extraTypes);`	Create an XmlSerializer for `type` and the types in the `extraType` array
`XmlSerializer(Type types, XmlAttributeOverrides overrides);`	Create an XmlSerializer for `type` and apply customization the attributes in `overrides`
`XmlSerializer(Type type, XmlRootAttribute root);`	Create an XmlSerializer for `type` and change the root element to `root`
`XmlSerializer(Type type, XmlAttributeOverrides overrides, Type[] extraTypes, XmlRootAttribute root, string defaultNamespace);`	Create an XmlSerializer for `type` and the types in the `extraType` array, apply customization the attributes in `overrides`, change the root element to `root` and specify the default namespace for all serialized objects

Declaring Types at Runtime

Previously we learned that the XmlSerializer has to analyze the types it is going to process before it is ready to convert XML to objects and vice-versa. We also learned several techniques based on metadata attributes to explicitly declare derived types, which the serializer can not automatically discover during the analysis. During the class analysis, the serializer checks for the attributes and records the type substitutions they declare. Obviously theses techniques only work to declare substitutions with types that exist at the time we write the declarations. We need a different solution for substitutions we can not declare in code. Take a collection class, like the ArrayList. There is no way Microsoft could declare all the types people are going to store inside an ArrayList. In fact they did not attach attributes inside the ArrayList class. Hence, serializing an ArrayList referencing anything else than objects of type object like in the following example causes an exception.

```
public static void BlowUpTheSerializer()
{
  ArrayList list = new ArrayList();
  list.Add( "aString" );
  XmlTextWriter writer = new
    XmlTextWriter( "List.xml", System.Text.Encoding.UTF8 );
  XmlSerializer serializer = new
      XmlSerializer(typeof(ArrayList));
  serializer.Serialize( writer, list ); // BOOM!
}
```

Adding Global Type Declarations

Serialize() throws the exception because the XmlSerializer's initial type analysis for the ArrayList found that all elements inside the ArrayList are declared to be of type object because the ArrayList needs to be able to reference ANY type. If we want to serialize an ArrayList referencing anything other than objects we need to declare the types differently.

The XmlSerializer allows declaring type global globally, by passing a Type array with the global types to the constructor. The following line, for example, would declare the string type within the ArrayList to avoid the exception in the example above:

```
  XmlSerializer serializer = new
      XmlSerializer(typeof(ArrayList),  // primary type declaration
        new Type[] {typeof(string)} );  // global type declaration.
```

With the global type declaration in place, the ArrayList in the example above serializes to the following XML document.

```
<?xml version="1.0" encoding="utf-8" ?>
<ArrayOfAnyType
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <anyType xsi:type="xsd:string">aString</anyType>

</ArrayOfAnyType>
```

The root element's name follows the XmlSerializer's naming scheme for all arrays and collections: It appends the type of the items within the array to "ArrayOf". The array item's node name identifies the declared item type ("anyType" is the XSD equivalent of the Object class, i.e. the root class of all classes). To identify the type of the actual object, the XmlSerializer added an xsi:type="xsd:string" attribute to the element.

Type Resolution With Global Type Declarations

For every field of an object the XmlSerializer serializes, it first checks the referenced object's type against the declared type of the field. If the object's type does not match the declared type, it checks against types declared by an XmlIncludeAttribute attached to the declared type. Finally, if the object's type does not match any included type, it checks the global type declarations. If the serializer finds a match it proceeds to serialize the type. Otherwise it will throw an exception.

When the XmlSerializer deserializes an XML type it checks whether an element matches the corresponding field name. If the name matches, it checks the element for an xsi:type attribute and compares the attribute's values against the global type declarations. When it finds a match it deserializes the object in the stream based on the type mapping for the matching type.

Runtime Customization

Modifying the default behavior of the XmlSerializer at runtime goes further than providing type information for undeclared types. We can perform (almost) all the same customizations we do statically through attributes dynamically as well. The XmlSerializer class defines two constructor overloads to override the default behavior or hard-coded attributes:

XmlSerializer(Type types, XmlAttributeOverrides overrides);

```
XmlSerializer(Type type, XmlAttributeOverrides overrides,
  Type[] extraTypes, XmlRootAttribute root, string defaultNamespace);
```

XmlAttributeOverrides Class

These constructors are helpful when we need to adjust static attributes only in certain cases, or if we have to support an extended XML format but we cannot change the attributes in the source code. The constructors accept the overriding attributes inside a special container object of the XmlAttributeOverrides class. This class exposes two overloaded Add() methods to apply attributes either on a class or an individual class member:

```
public void Add(Type type, XmlAttributes overrides);

public void Add(Type type, string elementName,
  XmlAttributes overrides);
```

The type parameter identifies the class to which we want to add serialization attributes, the elementName parameter identifies the class member. For fields, we attached attributes for each type we expected the field to reference at runtime. In the same vein we can assign an XmlAttributes object for each type a field might reference.

NOTE: You need to make sure you supply the same set of attributes if you use different instances of the XmlSerializer to serialize and deserialize a type. This happens automatically when the attributes are attached in source code, but not when you supply them at runtime. If you do not supply the same set of attributes the type mappings are different and your deserializing the XML stream might fail.

The XmlAttributes Class

The attributes themselves are stored in a yet another container object of the XmlAttributes class. This class exposes properties for each metadata attribute defined in the System.Xml.Serialization namespace, e.g. an XmlElements collection for XmlElementAttributes, an XmlRoot property for an XmlRootAttributes and so on. The only attribute we can not attach at runtime is the XmlIncludeAttribute, i.e. we cannot declare type substitutions at runtime. We can, however, declare types globally as we have seen in section 10.2.1.1 or declare substitute types for individual fields by dynamically attaching XmlElement or XmlArrayItem attributes.

1.4 Table 10.2 The properties of the XmlAttributes class store serialization attributes to customize the XmlSerializer's class-to-type mapping at runtime.

Property	Type	Access	Description
XmlAnyAttribute	XmlAnyAttributeAttribute	read/write	Contains the XmlAnyAttributeAttribute object marking field with literal XML attributes
XmlAnyElements	XmlAnyElementAttributes	read/write	Contains the XmlAnyElementAttributes collection marking fields with literal XML elements
XmlArray	XmlArrayAttribute	read/write	Contains the XmlArrayAttribute defining the name of an array root element.
XmlArrayItems	XmlArrayItemAttributes	read/write	Contains the XmlArrayItemAttributes collection defining properties of array item elements.
XmlAttribute	XmlAttributeAttribute	read/write	Contains an XmlAttribute object defining the XML attribute properties of a field.
XmlChoiceIdentifier	XmlChoiceIdentifierAttribute	read/write	Contains an XmlChoiceIdentifier referring to a field that identifies a choice particle.
XmlDefaultValue	Object	read/write	Contains the default value for an XML element or attribute.
XmlElements	XmlElementAttributes	read/write	Contains a collection of XmlElement objects defining the XML element properties of a field.
XmlEnum	XmlEnumAttribute	read/write	Contains an XmlEnumAttribute defining the enumeration properties of a field.
XmlIgnore	Boolean	read/write	Contains a flag whether or not to ignore a public field for serialization and deserialization.
Xmlns	Boolean	read/write	
XmlRoot	XmlRootAttribute	read/write	Contains an XmlRoot object defining properties of the root element if an object is at the top of a serialization hierarchy
XmlText	XmlTextAttribute	read/write	Contains an XmlText object marking a field as the container for XML text
XmlType	XmlTypeAttribute	read/write	Contains an XmlType object explicitly defining the corresponding XML schema type of a class

The constructor of the XmlSerializer throws an InvalidOperationException if the overriding attributes are incompatible with the type. For example, the XmlSerializer does not support dynamically adding an XmlArrayItem attribute to arrays or collections at the root of the serialized object graph.

Applying Runtime Overrides

Now let's put the pieces together and see how we can override hard coded serialization attributes. Imagine our parking lot application sends updates about the parked cars to interested parties in an XML format. For some reason there is one client who needs the data in a slightly different format[5]. Instead of writing an entire different set of classes to produce the custom format we simply customize the ones we already have to change the element names for ParkingLot and Car objects. You can find the class definitions in 9.4.1.1:

The example starts out by instantiating two XmlAttributes collections, one for the attributes for ParkingLot class, another one to apply attributes to the Cars field.

Next, we assign an XmlRootAttribute object to the XmlRoot property to change the root element name for ParkingLot objects.

Then we change the element name for a Car item in the Cars ArrayList to "ParkedCar" by assigning an XmlArrayItemAttribute to XmlArrayItems property.

Finally, we call the overloaded Add() method to add the two XmlAttributes objects to an XmlOverrides object.

Now the XmlOverrides object is set up and we can pass it to to the constructor of the XmlSerializer.

5 Listing 10.1: Serialization using attribute overrides

```
public static void SerializeCustomParkingLot(XmlWriter writer,
  ParkingLot parkingLot)
{
  XmlAttributes carsAttributes = new XmlAttributes();
  XmlAttributes classAttributes =
    new XmlAttributes();

  classAttributes.XmlRoot =
    new XmlRootAttribute("ParkingLotRoot");

  carsAttributes.XmlArrayItems.Add( new
    XmlArrayItemAttribute("ParkedCar", typeof(Car)));

  XmlAttributeOverrides overrides =
    new XmlAttributeOverrides();

  overrides.Add(typeof(ParkingLot), classAttributes);
  overrides.Add(typeof(ParkingLot), "Cars", carsAttributes);

  try
  {
    XmlSerializer xs = new XmlSerializer(
      typeof(ParkingLot), overrides );

    xs.Serialize( writer, parkingLot );
  }
  catch( InvalidOperationException ex )
```

[5] And the client had enough leverage for us to support a one-off solution

```
  {
    System.Console.WriteLine( "Bad override attributes" );
  }
}
```

The output of the Serialize() method (Listing 10.2) shows that XML nodes corresponding to items in the ArrayList "Cars" are now named "ParkedCar" as specified by the XmlArrayItem attribute. The XmlRootAttribute applied to the ParkingLot class changes the name of the root node from ParkingLot to "ParkingLotRoot".

6 Listing 10.2: A serialized ParkingLot object with and without the overriding attributes from Listing 10.1

```
<?xml version="1.0"
  encoding="utf-8" ?>
<ParkingLotRoot>
  <Cars>
    <ParkedCar>
      <Make>Ford</Make>
      <Model>Explorer</Model>
      <Year>1997</Year>
    </ParkedCar>
  </Cars>
</ParkingLotRoot>
```

```
<?xml version="1.0"
  encoding="utf-8" ?>
<ParkingLot>
  <Cars>
    <anyType xsi:Type="Car">
      <Make>Ford</Make>
      <Model>Explorer</Model>
      <Year>1997</Year>
    </anyType >
  </Cars>
</ParkingLot>
```

Generic XmlSeralizer

There are additional overloads of the XmlSerializer constructor that we have not discussed, but they merely offer different combinations of the parameters we just learned to use. One overload combines the functionality of all the other constructors: You can specify all the types the serializer can serialize, attach attributes to the types the serializer can handle, specify a root element name for each object the serializer reads or writes and define a default namespace:

```
XmlSerializer(Type type, XmlAttributeOverrides overrides,
  Type[] extraTypes, XmlRootAttribute root, string defaultNamespace);
```

One application of this overload is to create a generic serializer that can process many different root object types. Considering that instantiating an XmlSerializer object is an expensive operation, we can improve the performance of our applications by instantiating as few serializers as possible.

We can set up such a generic serializer by specifying the object type as the root type. The "real" types the constructor needs for setting up the serializer instance are all specified through the extraTypes parameter. Declaring object as the root type causes Serialize() to name the element for each object we pass in "anyType". That's hardly desirable because it does not map well to the real world, where the element names typically convey some information. To change the element names to something more descriptive we can either set the root parameter of the constructor, but this still only specifies one root element name for all serialized object graphs. Populating an XmlOverrides object, on the other hand, allows specifying a distinctive element

name for each type we serialize. The following example for a constructor illustrates the concept of the generic serializer. This serializer instance can serialize, otherwise unrelated, string, int and Car objects and changes the root element name to "MyObject" for each type.

```
XmlSerializer serializer = new XmlSerializer( typeof(object),
  null, // XmlOverrides, use to customize root element names
  new Type[] { typeof(string), typeof(int), typeof(Car) },
  new XmlRootAttribute( "MyObject" ),
  null ); // XML namespace
```

Choice Model Groups

In all cases we discussed so far, mapping a .NET class to an XML complexType was easy because the schema provided an unambiguous mapping from one type system to the other. Now, there are cases where the schema-type-to-.NET-type mapping is ambiguous, because the XML schema type defines ambiguous child element with the <choice> model group. Generally, a model group defines usage rules for a group of elements, or in XML terminology particles. The rules of a model group define which particles can occur in a group, in which order they occur and how many times. The <choice> model group defines a group of mutually exclusive particles. We have to examine this group a little bit deeper, because there is no true counterpart to the choice model group in object oriented programming languages. In a programming language each field of a class is present in each instance. The <choice> group on the other hand defines a set of "fields" and only one of them may be present in any given instance. It is up to the parsing logic in the application processing the XML to interpret the semantic differences between the elements. In the following section we learn how we design classes to bridge the two type systems and process XML types defined with the <choice> model group.

Mapping a Single-Value <choice>

First let's look at the simple case where a <choice> defines a single occurrence of a single element. The following schema describes a Car element with exactly four elements. The first three are always Make, Model and Year. Every Car has exactly one out of two additional elements because the <choice> occurs exactly once (minOccurs=1, maxOccurs=1) and either element within the groups occurs exactly once. The fourth element can be either a LeasePayment or a FinancePayment element.

7 Listing 10.3 A schema with a <choice> model group and two plossible instances of the Car_T type

```
<xs:schema id="Car"
      targetNamespace="http://tempuri.org/Car.xsd"
      xmlns="http://tempuri.org/Car.xsd"
      xmlns:xs=http://www.w3.org/2001/XMLSchema …>

  <xs:complexType name="Car_T" mixed="false">
    <xs:sequence minOccurs="1" maxOccurs="1">
      <xs:element name="Make" type="xs:string" />
      <xs:element name="Model" type="xs:string" />
      <xs:element name="Year" type="xs:int" />
      <xs:choice minOccurs="1" maxOccurs="1">
        <xs:element minOccurs="1" maxOccurs="1" name="LeasePayment"
```

```
          type="xs:int" />
        <xs:element minOccurs="1" maxOccurs="1" name="FinancePayment"
          type="xs:int" />
      </xs:choice>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="Car" type="Car_T"></xs:element>

</xs:schema>
```

```
<Car>
  <Make>Ford</Make>
  <Model>Explorer</Model>
  <Year>2002</Year>
<FinancePayment>699</FinancePayment>
</Car>
```

```
<Car>
  <Make>Ford</Make>
  <Model>Expedition</Model>
  <Year>2002</Year>
  <LeasePayment>429</LeasePayment>
</Car>
```

Identifying The Choice Particle

We could design a .NET class that maps to the Car type if the two elements were of different types. One technique would be to declare a field of type object and attach XmlElement attributes to resolve the two element names to the same field payment, but attaching two XmlElement attributes in only possible if the two different elements are of different types. LeasePayment and FinancePayment on the other hand are of the same type, which negates applying two XmlElements, because Serialize() cannot look at the type object the payment field refers to in order to figure out if the field refers to a LeasePayment or a FinancePayment. Equally, if we only provide a single field for the two payment types, Deserialize() can only store the amount in the XML stream, but not which type of payment type the XML stream specified to.

What we need to solve this problem is a second field to store the auxiliary information, which we will call the "choice field". In the example above the choice field would hold information about the payment. The type of the choice field is an enumeration of the elements in the <choice> that we need to disambiguate. The Serialize() method can now consult the choice field whether to generate a FinancePayment or a LeasePayment element when it processes a Car object. Likewise, Deserialize() can set the choice field to convey whether the data was deserialized from a LeasePayment or a FinancePayment. But wait, how does the XmlSerializer know about the choice field? We have to identify a choice field by attaching an XmlChoiceIdentifierAttribute to the field we need to disambiguate. The XmlSerializer detects the attribute when it analyzes the type and links the choice field to the data field.

Mapping a Class to a Single-Value Choice

Let's develop a class that maps to the XML type Car described by the schema above. Besides the fields for make, model and year we also have to supply two fields for the payment field: One for the data and one to indicate whether the data refers to a LeasePayment or a FinancePayment element.

First we create a public enumeration called ItemChoiceType with values named after the ambiguous elements in the <choice> model group.

Then we create the Car class with the two fields for the <choice>. The field Item to stores the element data and ItemElementName to identifies whether the data is a LeasePayment or a FinancePayment.

Next, we attach two XmlElementAttribute attributes to the data field so Deserialize() will know to store the data of either XML element in the Item field.

We also attach an XmlChoiceIdentifierAttribute pointing to ItemElementName to clarify the particle Item refers to. Deserialize() will set ItemElementName to reflect which <choice> element was present in the deserialized XML stream, Serialize() will read ItemElementName when it processes Item to determine what element to generate.

```
using System.Xml.Serialization;

[XmlType(Namespace="http://tempuri.org/Car.xsd",
  IncludeInSchema=false)]
public enum ItemChoiceType                              | #1
{                                                       |
    LeasePayment,                                       |
    FinancePayment,                                     |
}                                                       |

[XmlType(Namespace="http://tempuri.org/Car.xsd")]
[XmlRoot("Car", Namespace="http://tempuri.org/Car.xsd",
  IsNullable=false)]
public class Car_T
{
  public string Make;
  public string Model;
  public int Year;

  [XmlElement("LeasePayment", typeof(int))]
  [XmlElement("FinancePayment", typeof(int))]
  [XmlChoiceIdentifier("ItemElementName")]
  public int Item;

  [XmlIgnore()]                                 | #2
  public ItemChoiceType ItemElementName;        |
}
```

(annotation) <#1 The enumeration defines a value for each of the ambiguous elements>
(annotation) <#2 We have to attach an XmlIgnoreAttribute attribute to the ItemElementName field because it does not map to the <choice> model group. It is only used by the XmlSerializer.>

TIP: While we can create the class and the enumeration manually, but it is by far easier to let the XSD schema definition tool provided with the Framework SDK generate the complete class definition. You can find a discussion of this tool in appendix C.

Multi-Value Choice Model

The more complicated case to map a <choice> model group to a .NET class arises when the model group can occur more than once in an instance of the schema type, i.e. if the maxOccurs attribute on the group is greater than one. For example if we changed the schema definition from listing 10.3 to the following:

```
  <xs:choice minOccurs="1" maxOccurs="unbounded">
    <xs:element minOccurs="1" maxOccurs="1" name="LeasingRate;"
type="xs:int" />
    <xs:element minOccurs="1" maxOccurs="1" name="FinancePayment"
type="xs:int" />
  </xs:choice>
```

Then there is no restriction how many and in which order payment elements appear in a type, because each payment element is viewed as an instance of the model group, which can appear an "unbounded" number of times, like in the next XML fragment.

```
<Car>
  <Make>Ford</Make>
  <Model>Explorer</Model>
  <Year>2002</Year>
  <FinancePayment>699</FinancePayment>
  <FinancePayment>799</FinancePayment>
  <LeasePayment>429</LeasePayment>
  <FinancePayment>899</FinancePayment>
</Car>
```

This may sound a bit complicated, but what we learned in the previous section easily extends to <choice> definitions allowing multiple occurrences. First of all, we need to declare a data field of an array type to store all the data. Then we also make the choice field an array to clarify which particle of the model group the data items corresponds to. Once again, the type of the choice field has to be an enumeration with values for the ambiguous particle names. The items in the choice field array clarify the particle of the item in the choice array at the same position. Finally we attach an XmlChoiceIdentifierAttribute to the data field array to signal the XmlSerializer which two arrays contain the information to map objects of this class to the <choice> model group.

The modifications to the Car T class from the previous example to handle multiple occurrences of the <choice> are as simple as changing the data field and the choice field to array types as shown in the code fragment shown below:

```
  [XmlElementAttribute("LeasingRate", typeof(int))]
  [XmlElementAttribute("FinancePayment ", typeof(int))]
  [XmlChoiceIdentifierAttribute("ItemsElementNames")]
  public int[] Items;

  [XmlIgnoreAttribute()]
  public ItemsChoiceType[] ItemsElementNames;
```

Serializing XML nodes

Up until now our focus was on mapping elements and attributes in an XML document to fields of a .NET class. We have yet to talk about how we can process other features of XML documents, e.g. comments or processing instructions, with the XmlSerializer. These features are more rooted in document processing than data exchange. Processing instructions, for example, convey application specific information directly to the processor of the document, they should not carry data like elements and attributes do. Nevertheless, the XmlSerializer allows us to output these types of XML nodes through serialization or to retrieve them from documents through deserialization.

Processing XML documents with the XmlSerializer still requires an object-centered approach. It is not a general purpose parsing solution like the XmlDocument or the XmlTextReader and -Writer. We still have to serialize or deserialize objects to write or read the XML content. After all, it is still the XmlSerializer that is doing the work.

The key is to design classes with fields of the XML node classes in the System.Xml namespace to produce the required document items. If we exchange documents with an application that requires special processing instructions, for example, we can add a field of type XmlProcessingInstruction to the class that we serialize to produce the document. The XmlSerializer places the processing instruction directly into the output stream eliminating the need for any post-serialization processing steps to inject the processing instruction.

This concept works for all objects of classes derived from XmlNode: XmlDocument, XmlComment, XmlCDataSection, just to name a few more. When the XmlSerializer serializes these types of objects, it writes their OuterXml property, which provides the literal XML encapsulated by the object, verbatim to the output stream. Serializing an object of the following class, for example, will produce an XML document containing a processing instruction.

```
public class CarWithPI
{
  public CarWithPI() {}
  public XmlProcessingInstruction Instruction;
  public static void Main()
  {
    CarWithPI car = new CarWithPI();
    Instruction = doc.CreateProcessingInstruction("park", "and lock");

    XmlSerializer xs = new XmlSerializer(typeof(CarWithPI));
    XmlTextWriter writer = new XmlTextWriter("car.xml", Encoding.UTF8);
    writer.Formatting = Formatting.Indented;
    xs.Serialize( writer, car );
    writer.Close();
  }
}
```

The serialized representation of the object looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<CarWithPI …>
  <Instruction>
    <?park and lock?>
```

```
  </Instruction>
</CarWithPI>
```

The XmlSerializer created an element node for the Instruction field. You can see now that the XmlSerializer created an XML processing instruction inside the Instruction element. It did not create an element structure for the public properties of the XmlProcessingInstruction class.

> WARNING: We can only serialize XmlDocument objects that contain XML fragments, because a valid XML document must not have two document declarations (<?xml version …). The XmlSerializer creates the first declaration automatically, serializing a complete document would create a second one, but the XmlTextWriter in charge of creating the output document detects the second declaration and throws and exception.

This brings us already very close to being able to produce XML documents with information items besides elements and attributes with the XmlSerializer, but the element surrounding the items still undesirable in most cases.

XmlAnyElement Attribute

The .NET Framework supplies a metadata attribute to get rid of the surrounding element when we serialize an object derived from the XmlNode class. Attaching the XmlAnyElement attribute to a field instructs Serialize() to skip the enclosing element and write XML content of an XmlNode directly to the output. When we attach the attribute to the instruction field from the class above

```
[XmlAnyElement]
public XmlProcessingInstruction Instruction;
```

the instruction element is no longer part of the output.

```
<?xml version="1.0" encoding="utf-8"?>
<CarWithPI …>
  <?park and lock?>
</CarWithPI>
```

When we attach more than one XmlAnyElementAttribute inside a class we have to assign a different name to each instance, otherwise the XmlSerializer constructor cannot keep all the different instances apart and throws an exception. The XmlSerializer ignores the provided name unless the attribute is attached to a field of type XmlElement or XmlElement[]. In those cases, named XmlAnyElement attributes restrict the fields to element with the names specified by the attached attribute(s). The XmlSerializer will throw an exception to enforce this restriction whenever it detects a mismatch.

Deserializing Xml Nodes

Accessing XML nodes when we deserialize objects from an XML document does not work as well as creating them through serialization. Of all the XmlNode derived classed, the XmlSerializer deserializes only entity references and elements. It ignores comments, processing instructions and documents, regardless if we attach an XmlAnyElement attribute. It also

deserializes fields of type XmlElement, but they play a special role as we will see in the next section.

These limitations are not a show stopper given the XmlSerializer's focus of data-driven applications, since neither comments nor processing instructions are part of the XSD type system. Yet, it is interesting to note that we can serialize XmlNode derived objects to create additional XML content, but we cannot access them through deserialization. Being able to produce literal XML and declare fields of type XmlElement to allow for increased flexibility parsing XML documents also has a special application as we see in the following section.

Mapping XML Types Wildcards

So far we always maintained a close relationship between a .NET class and an XML schema type. However, in many cases XML schema types feature some room for extensibility or customization. Take the types in the SOAP protocol for message exchange between applications for example. The SOAP XML format defines types as structured containers to transmit application specific information, like in the following outline:

```
<Envelope>
  <Header>
    <!-- application specific headers go here -->
  </Header>
  <Body>
    <!-- application specific message content goes here -->
  </Body>
</Envelope>
```

The XmlSerializer also enables us to design classes for these container types as well. With these classes we can modes of the structure of the types and provide room for application specific content without having to write custom classes for each XML message.

Since most of the content of a SOAP message is application specific, the SOAP schema has to describe the overall structure of the message without restricting the content of the Header or Body elements. For cases like this, the XML schema standard defines the <any /> content model for arbitrary XML content and the <anyAttribute /> item to allow extensible document definitions for cases like this. A simplified version of the Envelope and the Body definitions in the SOAP schema looks like this:

8 Listing 10.4: Two extensible XML types (adapted from the SOAP protocol)

```
<xs:element name="Envelope" type="Envelope_T" />
<xs:complexType name="Envelope_T" >
  <xs:sequence>
    <xs:element ref="Header" minOccurs="0" />
    <xs:element ref="Body" minOccurs="1" />
    <xs:any minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:anyAttribute />
</xs:complexType>

<xs:element name="Body" type="Body_T" />
<xs:complexType name="Body_T" >
```

```
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded" />
  </xs:sequence>
  <xs:anyAttribute />
</xs:complexType>
```

This concept of open extensibility does not map well to what we have learned so far about mapping .NET classes to XML types with the XmlSerializer. We always required a one-to-one mapping between an XML element or attribute and a field in a .NET class. What we have yet to learn is how we can design classes to access XML content that was not explicitly defined in an XML schema.

Serializing Objects

The way to serialize classes with customized XML builds on what we have learned in the previous section. We can design classes with special fields to store the custom XML. Fields of type XmlElement[], XmlDocument or XmlDocumentFragment can store the extension part of an XML type defined with the <any /> model. When we serialize an object of this class, the XmlSerializer will serialize the XML content as well formed XML into the output stream. We also need to attach an XmlAnyElement attribute to he field to prevent an additional XML element around the XML stored inside.

Developing classes to map XML types with custom attributes works just the same. The class needs a field which will store all custom attributes. For the XmlSerializer to create extra attributes we have store the attributes in an array of type XmlAttribute. Instead of an XmlAnyElement attribute we need to attach an XmlAnyAttribute attribute and our class is ready to go. An example featuring both, custom attributes and elements is the Body type of the SOAP protocol from listing 10.4. Listing 10.5 shows a class, which provides the extensibility through an XmlElement[] and an XmlAttribute[] field.

9 Listing 10.5 A class to map an extensible XML type.

```
[XmlRoot("Body")]
public class SoapBody
{
  public SoapMessage() {}
  [XmlAnyAttribute]
  public XmlAttribute[] BodyAttributes;
  [XmlAnyElement]
  public XmlElement[] BodyContent;
}
```

Deserializing Objects

Deserializing extensible XML content is slightly trickier than serializing it. By default, the XmlSerializer ignores any nodes that do not map to a field in the class, but we can access any unmapped nodes on one of two ways:

- The XmlSerializer stores unmapped nodes in designated fields of the deserialized object.
- The XmlSerializer notifies the application in control of deserializing the XML stream of unmapped XML content.

The two ways are mutually exclusive and the former takes precedence over the latter.

Deserializing Unmapped Nodes

The first approach we learn to gain access to unmapped nodes in XML stream is designate fields in the deserialized class to receive the unrecognized nodes. This approach works well for scenarios where we anticipate XML documents with a fixed XML structure and some with customizable sections, like in the example of the SOAP protocol. The XmlSerializer stores all extra XML nodes in the designated fields of the deserialized object and the unmapped nodes are available to the application as part of the deserialized object.

We designate the fields again by attaching the XmlAnyAttribute or the XmlAnyElement attribute to a field of type XmlNode[], XmlAttribute[] or XmlElement[]. Unrecognized elements are added to the array tagged with the XmlAnyElement attribute, unrecognized attributes are added to the array tagged with the XmlAnyAttribute attribute.

NOTE: We can attach only one instance of the XmlAnyAttribute and the XmlAnyElement of these attributes inside a class or the Deserialize() method would not know where to store the extra nodes.

The designated fields receive only the unrecognized nodes found within the scope of the deserialized class. Any unmapped nodes that are not immediate children of the top level element for a class belong to a different class. The XmlSerializer discards any unmapped nodes if the deserialized class does not designate any fields to receive them. This sounds more complicated than it is. Figure 10.1 illustrates how unrecognized nodes in an XML document, shown on the right hand side, are mapped to the class hierarchy on the left hand side.

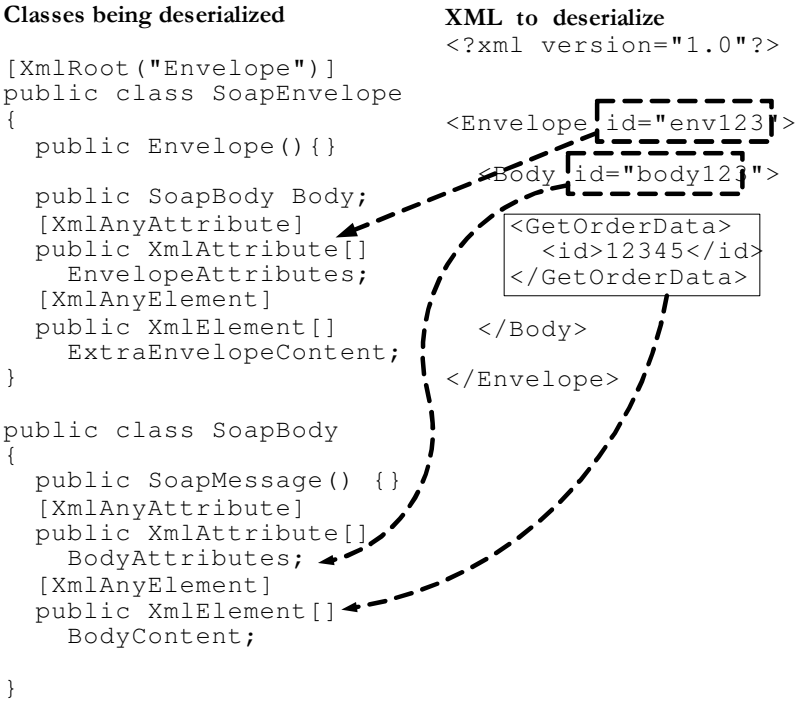


Figure 10.1 Fields decorated with the XmlAnyArrayAttribute and the XmlAnyElementAttribute receive XML nodes that do not map to a field or property. Nodes are discarded if they do not directly map to a field and the currently deserialized object declared no fields with one of these attributes is available.

On the left hand side of the figure we see how we could design a class to process the extensible types from listing 10.4. These classes loosely relate to XML messages formatted according to the SOAP protocol standard, but they do not nearly reflect all features of a SOAP message. The SoapEnvelope class is the root of the deserialized hierarchy. The SoapEnvelope class defines a for the Body part of the message of the message. For optional other elements in the Envelope the class defines the ExtraEnvelopeContent field with the XmlAnyElement attribute attached. The SoapBody class does not define any fields besides an XmlElement array for its content, because the XML schema fragment from listing 10.4 does not define any mandatory elements.

On the right hand side of the figure we see an XML document that resembles a SOAP message. When we pass this XML document to the Deserialize() method of the XmlSerializer, it will store all content inside the Body in the BodyContent field. The two id attributes are not part of the standard SOAP format; hence none of the two classes defines fields for them. Both classes do define a field for the XmlSerializer to store any attributes on their respective top level elements, Envelope and Body. Note that the XmlSerializer would discard the Body's id element if the SoapBody class would not provide its own field marked with the XmlAnyAttribute attribute.

Event notifications

The second way to gain access to unmapped XML nodes in an XML stream that is deserialized is to register event handlers with the XmlSerializer. Attaching the XmlAnyAttribute or XmlAnyElement attributes works great where we anticipate the content we process to contain unmapped nodes on a regular basis. Registering event handlers for unmapped nodes on the other hand works better for cases where we want an application to handle them as an exception rather than as the rule. The XmlSerializer raises four different events for unmapped nodes when it is deserializing a class without XmlAnyAttribute or XmlAnyElement attributes. We can register event sinks for the four events listed in table 10.2 to receive notifications for unexpected XML in the stream.

1.5 Table 10.2 Events defined by the XmlSerializer

Event	Description
void UnknownAttribute(Object sender, XmlAttributeEventArgs args)	Fires when the XmlSerializer encounters an XML attribute without a corresponding class field during deserialization.
void UnknownElement(Object sender, XmlElementEventArgs args)	Fires when the XmlSerializer encounters an XML element without a corresponding class field during deserialization.
void UnknownNode(Object sender, XmlNodeEventArgs args)	Fire when the XmlSerializer encounters an XML node of unknown type during deserialization.
void UnreferencedObject(Object sender, UnreferencedObjectEventArgs args)	Fires during deserialization of a SOAP-encoded XML stream. → ?? KELLY ?? ←

The XmlSerializer fires one of these events for each unmapped node it encounters while deserializing an object, but object's class does not designate fields with the XmlAnyAttribute or the XmlAnyElement. Each event provides details about the unmapped node in the form of a event specific arguments class passed to the event handler. For example, when the XmlSerializer fires

and for an unmapped attribute in the XML stream, it passes a reference to itself and an XmlAttributeEventArgs object to the registered event handler. The arguments object contains the line number and position of the attribute within the deserialized XML document, as well as the attribute itself.

The following example shows how to set up event handlers to log the event details about nodes the XmlSerializer could not map to any class members to the console.

10 Listing 10.5 Logging nodes the XmlSerializer can not map to class fields during deserialization

```
public void BindSerializerEvents( XmlSerializer ser )
{
  ser.UnknownAttribute +=                                  | #1
    new XmlAttributeEventHandler(OnUnknownAttribute);      | #2

  ser.UnknownElement +=
    new XmlElementEventHandler(OnUnknownElement);

  ser.UnknownNode +=
    new XmlNodeEventHandler(OnUnknownNode);
}

public static void OnUnknownAttribute(Object sender,
      XmlAttributeEventArgs args)
{
  string typeName = "N/A";
  if( args.ObjectBeingDeserialized != null )
  {
    typeName =
      args.ObjectBeingDeserialized.GetType().FullName;
  }
  Console.WriteLine(                                      |#3
    @"Unknown attribute {0}:{1}='{2}' at line {3},        |
    position {4}, type: {5})",                            |
    args.Attr.NamespaceURI,   args.Attr.LocalName,        |
    args.Attr.Value, args.LineNumber,                     |
    args.LinePosition, typeName );
}

public static void OnUnknownElement(Object sender,
      XmlElementEventArgs args)
{
  string typeName = "N/A";
  if( args.ObjectBeingDeserialized != null )
  {
    typeName =
      args.ObjectBeingDeserialized.GetType().FullName;
  }
  Console.WriteLine(                                     |#4
    "Unknown element {0}:{1}='{2}' at line {3},          |
    position {4}, type: {5})",                           |
    args.Element.NamespaceURI,                           |
```

```
    args.Element.LocalName, args.Element.InnerXml,     |
    args.LineNumber, args.LinePosition, typeName);     |
}

public static void OnUnknownNode(Object sender,
      XmlNodeEventArgs args)
{
  string typeName = "N/A";
  if( args.ObjectBeingDeserialized != null )
  {
    typeName =
      args.ObjectBeingDeserialized.GetType().FullName;
  }
  Console.WriteLine(                                    |#5
    "Unkown Node type {6} {0}:{1}='{2}' at line {3},    |
    position {4}, type: {5})",                          |
    args.NamespaceURI, args.LocalName, args.Text,       |
    args.LineNumber, args.LinePosition, typeName,       |
    args.NodeType.ToString() );                         |
}
```

(annotation) <#1 The += operator is used to register an event sink >
(annotation) <#2 We have to instantiate a delegate object for the event sink method>
(annotation) <#3 The XmlAttributeEventArgs object contains detailed information about the unmapped attribute.>
(annotation) <#4 The XmlElementEventArgs object contains detailed information about the unmapped element.>
(annotation) <#5 The XmlNodeEventArgs object contains detailed information about the unmapped node.>

Serialization Namespaces

All the way through the last one and a half chapters we have barely mentioned XML namespaces, but that does not imply that the XmlSerializer does not support them. In fact, the XmlSerializer recognizes the important role XML namespaces are playing when systems exchange data through XML and supports them through a variety of features:

- Default Namespaces per serializer instance
- Static XML Namespace declaration per class or field in the class' source code
- Static namespace prefix declaration in the class' source code
- Dynamic XML Namespace declaration per class or field at runtime
- Dynamic namespace prefix declaration at runtime

This section will take a closer look at each one of them, enabling us develop XML namespace enabled solutions with the XmlSerializer.

Declaring namespaces in the source code

Two of the namespace support features of the XmlSerializer enable namespace management inside the source code of a class. When we develop classes to bind data from XML schema types we can add the namespaces directly to the source code by attaching metadata attributes. The XmlSerializer reads the namespace information and produces the namespace declarations upon serialization of an object. Of course, it will also account for namespace correct-ness when it is deserializing an XML stream. If a metadata attributes defines for a field to bind to an XML node

from a specific namespace then the XmlSerializer will only populate that field from an XML node where node name and namespace match the attribute.

The seconds feature allows providing prefix declarations for namespaces, but more on that later, first we will see how to define namespace relationships in the source code.

Declaring namespaces for individual attributes and elements

You may have noticed the Namespace property on most of the metadata attributes when you studied table 9.4, but we have yet to examine what they do. When we attach XML serialization attribute to a field, we can set the attribute's Namespace property to tell the XmlSerializer which XML namespace the element or attribute corresponding to the field belongs to. We can declare XML namespaces at different levels inside a class.

At the highest level, we can specify the XML namespace of the schema type with class' XmlRoot attribute. However, the XmlSerializer does not consider XmlRoot when an object is not at the top of a serialization hierarchy. This is not a problem if all classes in an object model correspond to types from the same XML schema. However, if we model a schema that imports types from different namespaces, our classes have to reflect this at the field and property definitions. We declare the external namespaces by attaching metadata attributes to fields referencing an imported type.

Let's put see how this works when we actually design a class! We start out with an XML schema to describe a type named Car_T. The schema imports types from two other namespaces. One is referenced by an attribute; one is referenced by an element of the Car_T type. The Car_T type declares another element, Year, which is part of the schema's namespace (listing 10.7).

11 Listing 10.7 A schema importing types from two other namespaces

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema xmlns:tns="urn:christoph-types-cars"
  targetNamespace="urn:christoph-types-cars"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:import namespace="urn:cars-models" />
  <xs:import namespace="urn:cars-makes" />

  <xs:element name="Car" type="tns:Car_T" />
  <xs:complexType name="Car_T">
    <xs:sequence>
      <xs:element minOccurs="0" maxOccurs="1"
        xmlns:q1="urn:cars-models" ref="q1:Model" />
      <xs:element minOccurs="1" maxOccurs="1" name="Year" />
    </xs:sequence>
    <xs:attribute xmlns:q3="urn:cars-makes" ref="q3:Make" />
  </xs:complexType>
</xs:schema>
```

Then we write a Car class that the XmlSerializer will relate to the Car_T type. To make sure the XmlSerializer includes the correct namespace declaration when we serialize a Car object we

specify the namespace through an XmlRoot attribute attached to the class. We also have to attach XmlAttribute and XmlElement attributes on the fields corresponding to the types imported from another namespace. Note that we have to declare namespaces on fields even when their classes already contain XmlRoot attributes to declare their XML namespace.

12 Listing 10.8 A class with XML namespace declarations

```
[XmlRoot("Car", Namespace="urn:christoph-types" )]
public class Car
{
  public Car() {}
  [XmlAttribute(Namespace="urn:cars-makes")]
  public string Make;
  [XmlElement(Namespace="urn:cars-models")]
  public string Model;
  public int Year;
}
```

When the XmlSerializer serializes an instance of this class, it automatically includes the namespace declarations from attached attributes. The root element declares the XML namespace we specified to the XmlRoot attribute as the type's default namespace. It also generates the declaration of the Make attribute with a dummy prefix, since the attribute does not belong to the default namespace. Finally, it declares the Model element's namespace through another local namespace declaration on the model element.

13 Listing 10.9 A serialized instance of the Car class

```
<Car
  d3p1:Make="Ford"
  xmlns:d3p1="urn:cars-makes"
  xmlns="urn:christoph-types-cars" …>

      <Model xmlns="urn:cars-models">Explorer</Model>
      <Year>1997</Year>
</Car>
```

Namespace Prefixes

We can also have the XmlSerializer create prefix declarations and qualified names instead of repeating namespace declarations on every element or attribute. The first approach puts the control over prefix declarations inside the serialized object. The XmlSerializer checks objects it serializes for a field of type XmlSerializerNamespaces adorned with an XmlNamespaceDeclarations attribute. If it finds such a field, it writes the prefix declarations contained in the XmlNamespaceDeclarations object with the top element of the object graph. Whenever a field or child object references a namespace declared in the XmlNamespaceDeclarations, the XmlSerializer generates the declared prefix instead of writing out another local namespace declaration.

The code in Listing 10.10 demonstrates the use of an XmlSerializerNamespaces field to declare prefixes for the namespaces of the Make, Model and Year fields.

14 Listing 10.10 A class with a field for namespace prefix declarations.

```
public class CarWithQualifiedNames
```

```
{
  [XmlAttribute(Namespace="urn:cars-makes")]
  public string Make;
  [XmlElement(Namespace="urn:cars-models")]
  public string Model;
  public int Year;
  [XmlNamespaceDeclarations]
  public XmlSerializerNamespaces Namespaces;
}
public static void SerializeCar(CarWithQualifiedNames car,
  XmlTextWriter writer )
{
  car.Namespaces = new XmlSerializerNamespaces();
  car.Namespaces.Add( "mk", "urn:cars-makes" );
  car.Namespaces.Add( "md", "urn:cars-models" );

 XmlSerializer serializer = new XmlSerializer(
    typeof(CarWithQualifiedNames) );
  serializer.Serialize( writer, car );
}
```

When we serialize an object of this class, the XmlSerializer declares the prefixes we added to the Namespaces map before we called Serialize() on the start tag of the object. While these declarations are in scope, i.e. for all fields or the CarWithQualifiedNames class and its children, these namespaces are referenced through these prefixes (listing 10.11).

15 Listing 10.11 A serialized Car objects with namespace prefixes

```
D:\christoph\c - #<CarWithQualifiedNames
  xmlns:mk="urn:cars-makes"
  xmlns:md="urn:cars-models"
  mk:Make="Ford" …>

  <md:Model>Explorer</md:Model>
  <Year>1997</Year>

</CarWithQualifiedNames>
```

The single prefix declaration in this example does not illustrate the benefit of prefix declarations well. However, when you serialize large hierarchies with types from many different namespaces, the resulting documents quickly become cluttered with local namespace declarations.

> WARNING: Declaring prefixes is a good and widely-used approach to tidy up XML documents, but declaring prefixes in the class code is very risky. You should always define namespace prefixes in the scope of the document in which they are valid to maintain flexibility into which documents you serialize you classes. Always leave prefix declarations up to the application, not to the serialized objects.

The Namespaces collection also serves a second purpose. The XmlSerializer stores the prefixes found the in the XML document when it deserializes an object. Nevertheless, you should not care about the namespace prefixes defined throughout a particular document instance.

Namespaces at runtime

There are several ways we can declare namespaces and prefixes at runtime. The XmlSerializer itself allows us to define a global default namespace for all objects it processes as well as supplying prefix declarations for namespaces references by the serialized objects. While we can declare namespaces at runtime by dynamically attaching XML serialization attributes, there are hardly any use-cases that warrant doing so.

You can also leverage the namespace prefix support built into the XmlTextWriter if you are serializing more than one object into the same XmlTextWriter instance, but in this section we focus on the capabilities of the XmlSerializer.

Default Namespace Declaration

We can declare a default namespace for all serialized and deserialized objects by passing the namespace's URI to the XmlSerializer constructor as shown in the following fragment.

```
public static void SerializeACarWithDefaultNamespace( Car aCar,
XmlWriter writer )
{
  XmlSerializer xs =
      new XmlSerializer( typeof( ParkingLot ),
      "urn:christoph-cars" );
  xs.Serialize( writer, aCar );
}
```

The XmlSerializer adds the declaration for this namespace to the output of all serialized objects that do not declare themselves a namespace through and XmlRoot attribute. Likewise, serialized objects of all classes that do not explicitly declare a different namespace have to be part of the default namespace to be properly deserialized.

Declaring Namespace Prefixes

Declaring XML namespaces at runtime bears a distinct advantage over declaring them at compile time. Imagine if we defined a namespace prefix inside a class, but that prefix is already in use for a different namespace in the document we are serializing into. While it is technically possible to declare the same prefix for different namespaces as long as the scoping of the declarations is clear, it is very confusing to a human reader and negates the use of prefixes altogether. Furthermore, the XmlSerializer does allow conflicting prefix declarations in the same scope, which makes declaring namespaces in class code downright dangerous.

We can declare prefixes at runtime by setting up an XmlSerializerNamespaces dictionary with the prefix-to-namespace mappings, just like we did when the serialized class itself contained the dictionary. This time we supply the dictionary to the XmlSerializer directly, through one of the overloaded version of the Serialize() method. The XmlSerializer will declare all the prefixes in the dictionary at the root element of the serialized object, which means that these prefix declarations are only valid for this particular call of Serialize(). You will have to pass them again to subsequent calls if you want to repeat the declarations, even if the serialized object is of the same type.

NOTE: The prefix declarations will not be repeated if they are already defined in the current scope.

16 Listing 10.12 Declaring namespace prefixes at runtime.

```
public static void SerializeACarWithQualifiedNames( Car aCar, XmlWriter
writer )
{
  XmlSerializer xs =
     new XmlSerializer( typeof( ParkingLot ),
      "urn:christoph-cars");

  XmlSerializerNamespaces namespaces =
    new XmlSerializerNamespaces();

  namespaces.Add( "md", "urn:cars-models" );
  namespaces.Add( "mk", "urn:cars-makes" );

  xs.Serialize( writer, aCar, namespaces );
}
```

The code snippet above demonstrates how to set up the XmlSerializerNamespaces collection to declare a set of prefixes. The XML document fragment below shows the output from the Serialize() call.

17 Listing 10.13 The output of listing 10.12, the namespace prefixes are declared at runtime.

```
D:\christoph\c - #<Car
  xmlns:mk="urn:cars-makes"
  xmlns:md="urn:cars-models"
  mk:Make="Ford">

  <md:Model>Explorer</md:Model>
  <Year>1997</Year>

</Car>
```

NOTE: By default the XmlSerializer creates the namespace declarations xmlns:xsd=http://www.w3.org/2001/XMLSchema and xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance at the root of each serialized element[6] to support data type declarations. If we provide a namespaces collection to the Serialize() method, it will generate declarations for the namespaces in the collection only. We can leverage this if we need to omit the default declarations and by passing an empty collection. If the serialized content requires these namespaces the XmlSerializer declares them locally.

[6] These declarations are omitted for clarity in the example documents in chapter 9 and 10.

DataSet Object

The last class we discuss in the context of the XmlSerializer is the DataSet. We have learned all about how to store and access XML data in a DataSet object in chapter 8. Now it's time to learn how the XmlSerializer serializes and deserializes DataSet objects, because it's different from anything else we have learned so far. Fasten your seat-belt, here we go.

The .NET architects wanted the DataSet class to become the preferred vehicle to transmit data through WebServices. Since efficiency is an important aspect in the transmission of data they designed a special (XML based) format to transfer DataSets data, which only transmits the changes since a DataSet was loaded. The format is called a DiffGram and is discussed in more detail in chapter 8. Since ASP.NET WebServices create and parse SOAP messages with the XmlSerializer, the serializer was now on the hook to serialize DataSet object to the DiffGram format instead of simply serializing the data exposed by the public properties. Furthermore, it needed to support the reverse operation and deserialize DataSets from DiffGrams as well.

The following example serializes a DataSet loaded from a simple XML document. Note that we do not set up anything different from the way we serialized objects throughout the past two chapters to enable serialization to the DiffGram format.

18 Listing 10.14 Serializing a DataSet containing simple contacts file

```
<?xml version="1.0" encoding="utf-8" ?>
<contacts>
    <contact>
        <name>Roger Dolph</name>
        <address>100 Washington Ave, Atlanta, GA</address>
        <phone>123-4568</phone>
   </contact>
</contacts>

static void SerializeContactDataSet()
{
  DataSet   dataset = new DataSet();
  dataset.ReadXml("contacts.xml");
  XmlSerializer ser = new XmlSerializer( typeof( DataSet ) );
  XmlTextWriter writer = new XmlTextWriter( "dataset.xml",
    System.Text.Encoding.UTF8 );
  writer.Formatting = Formatting.Indented;
  ser.Serialize( writer, dataset );
  writer.Close();
}
```

When we examine the output of the XmlSerializer we find two distinct sections. One is the DiffGram we expected to find. The other one is an XML schema describing the data structure and the relationships inside the DataSet in addition to the data in the various tables. The schema information is necessary to accurately deserialize the DataSet later on, since the DiffGram does not include relationship information. Listing 10.15 shows the complete output of the method above.

Listing 10.15 A serialized DataSet

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<DataSet>
  <xs:schema id="contacts" xmlns=""
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
  <xs:element name="contacts" msdata:IsDataSet="true">
      <xs:complexType>
    <xs:choice maxOccurs="unbounded">
          <xs:element name="contact">
      <xs:complexType>
              <xs:sequence>
                <xs:element name="name" type="xs:string" minOccurs="0"
/>
                <xs:element name="address" type="xs:string"
minOccurs="0" />
                <xs:element name="phone" type="xs:string" minOccurs="0"
/>
      </xs:sequence>
            </xs:complexType>
     </xs:element>
        </xs:choice>
   </xs:complexType>
    </xs:element>
 </xs:schema>
  <diffgr:diffgram xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
xmlns:diffgr="urn:schemas-microsoft-com:xml-diffgram-v1">
  <contacts>
      <contact diffgr:id="contact1" msdata:rowOrder="0"
diffgr:hasChanges="inserted">
    <name>Roger Dolph</name>
        <address>100 Washington Ave, Atlanta, GA</address>
    <phone>123-4568</phone>
      </contact>
  </contacts>
  </diffgr:diffgram>
</DataSet>
```

IXmlSerializable Interface

Looking at this complex format begs the question how the XmlSerializer generates a DiffGram when it serializes a DataSet. When the XmlSerializer turns a DataSet into XML it will not retrieve any values from any property. Instead it lets the object control its own serialization. The DataSet implements the unsupported interface System.Xml.Serializarion.IXmlSerializable shown in listing 10.17.

19 Listing 10.17 The IXmlSerializable interface.

```
public interface IXmlSerializable
{
  XmlSchema GetSchema();
  void ReadXml(XmlReader reader);
  void WriteXml(XmlWriter writer);
}
```

Whenever Serialize() detects that an object implements this interface it will pass the XmlWriter to the object and does not attempt to serialize the object. Likewise, Deserialize() passes the XmlReader instance to the object and lets the object parse the XML stream.

> BIG, FAT WARNING: IXmlSerializable is documented as "for internal use only". Microsoft is free to change this interface at any time. Keep this in mind if you should ever decide to implement this interface yourself.

Sometimes implementing IXmlSerializable may be the only way to serialize a class, so let's take a quick look at the interface, method-by-method:

- GetSchema() is called when the constructor is creating a type mapping for the class. You need to return an XmlSchema object that describes the XML format created by WriteXml().
- ReadXml(XmlReader reader) is called from the Deserialize() method right after it was able to identify a type that implements IXmlSerializable in the XML stream. It constructs the object and calls ReadXml() on it. The XmlReader is positioned at the first child node of the object's root. You have to read the XML stream and set your object's fields yourself. The details about reading XML streams with classes derived from XmlReader are explained in chapter 2.
- WriteXml(XmlWriter writer) is called from the Serialize() method when the XmlSerializer detects that an object that implements IXmlSerializable. The writer is positioned after the root node for the object when it is passed to the WriteXml() method.

With those three methods you can completely control how serialization and deserialization of a class takes place. You can find an example how to implement IXmlSerializable in our web site, if you feel the urge to explore the power of this interface in more depth. Again: implement this interface only as a last resort solution. Since you cannot assume that Microsoft supports this interface in a later version of the .NET Framework, you must deploy your solution only in environments where you can ensure the installed version of the .NET runtime supports this interface.

SerializableAttribute

We can control serialization of class at a number of levels. First of all, we can express whether or not we intend a class to be serialized when we develop it. At a finer-grained level of control, we can code classes to take control over their serialization if the runtime's serialization mechanism is not appropriate and finally, we can override serialization for any given class externally, by delegating the whole process to a different class. We will learn about each of these options in this chapter.

There is only one true requirement for classes the SoapFormatter can process: A class needs permission from its author to be serialized. This permission is expressed by attaching the SerializableAttribute from the System namespace a class. Any time we add the Serializable attribute to a class like this:

```
[System.Serializable]
public class SerializableClass
{
```

```
}
```

we let other developers know that we gave our OK to them serializing our class. The serialization formatters check each object they serializes for this attribute and throw a SerializationException if the attribute is not present. If the attribute is present the formatter will persist the state of every(!) field, public and private alike, to the specified output stream. The fields of a [Serializable] object have to reference objects that are themselves [Serializable], or else we must explicitly exclude them from the generated SOAP message (we will learn how to do that in the following section). All classes higher up in the inheritance hierarchy, i.e. classes we derive from, have to have serialization clearance as well. Otherwise we could easily circumvent the class authors' intent and make a non-serializable class serializable by simply deriving from it.

Many classes in the .NET Framework libraries are marked with this attribute, in fact, all primitive types in the System namespace are. Check the .NET Framework documentation to see which ones are and which ones are not. The notable exception on the long list of serializable classes, are the classes in the System.Xml namespace, XmlDocument and XmlNode for example. While they provide excellent support for serialization with the XmlSerializer, we cannot serialize them with a serialization formatter. We have to code serialization for these classes explicitly. How this works we will see in section 12.1.3.

The NonSerializedAttribute

Some classes do not want to serialize all their fields either a) because they reference non-serializable objects, b) because we want to prevent sensitive information, like passwords or encryption keys, to show up in the output, or c) because it does not make sense. Serializing delegates and events, for example, is usually not interesting because they are tied to the application executing when the object is serialized. For all these cases the System namespace provides the NonSerializedAttribute. We can attach it to the fields we want to exclude from the serializer output as shown in the following code fragment:

```
[Serializable]
public class SerializableClass
{
  [NonSerialized]
  public string _MySuperSecretPassword;

  // …
}
```

When the SoapFormatter serializes an instance of the SerializableClass class, there will be no reference to the _MySuperSecretPassword member. Accordingly, NonSerialized fields are initialized to null in objects created by deserialization.

Custom Serialization

While excluding individual fields helps us getting around some problems, there will be cases where we cannot exclude non-serializable fields from the output because they store vital information. For these (and some other) cases the serialization framework provides another

option to customize object serialization: Classes can take control of the data they want to serialize by implementing the ISerializable interface. This option requires writing actual code to handle serialization and deserialization instead of changing the behavior simply by attaching attributes to a class. In turn it gives us a great deal of flexibility in the way we can save and restore objects.

Serializing objects with ISerializable

After a serialization formatter checked the object it is about to serialize for the SerializableAttribute it always queries for the ISerializable interface to find out whether or not the object can take over its own serialization. If the object implements this interface, the formatter calls the interface's only method: GetObjectData() to pass a SerializationInfo container that the object can fill with the data it wants to serialize.

1.6 Table 12.1 The ISerializable interface defines one single method: GetObjectData(). The formatter calls this method to retrieve the data to serialize. Implementing ISerializable implies implementing a deserialization constructor with the same signature as GetObejectData.

Method	Description
void GetObjectData(SerializationInfo *info*, StreamingContext *context*)	Fills the SerializationInfo *info* with the data to serialize. The StreamingContext contains information to tune the serialized data to the intended deserialization scenario.

The SerializationInfo

A SerializationInfo object stores name-value pairs, similar to a dictionary. Additionally, it provides some properties (table 12.4) to control the type and assembly information written to the output stream. This information will be used later on to locate the correct type and its assembly when the object is deserialized. By default these properties are initialized to the full type name and the fully qualified assembly name, but we can change and cause deserialization to happen with an instance of a different class. Changing these properties is useful if the serialized object is only a proxy object, because it can serialize itself as if it was the actual object.

We can store any object in the SerializationInfo container by calling one of the many overloaded versions of the AddValue() method. Ideally, we fill the container with enough information to re-create the serialized object later, but there is nothing in the framework to enforce this. Note that the SerializationInfo object will serialize objects passed to AddValue() method the same way any other objects are serialized, i.e. they also require the SerializableAttribute and can handle their own serialization by implementing the ISerializable interface. The SerializationInfo is just a data container; it has no influence to what format the data inside is persisted to. The format is solely determined by the serialization formatter. The BinaryFormatter produces a binary format and the SoapFormatter produces SOAP messages with the data stored in the SerializationInfo. Once GetObjectData() returns the formatter iterates over the items in the SerializationInfo object and serializes the value of each name-value pair into an element named according to the pair's name.

1.7 Table 12.4 The properties of the SerializationInfo class identify the serialized type and its assembly.

Property	Access	Type	Description
AssemblyName	(read/write)	string	The assembly name of the type being (de-)serialized
FullTypeName	(read/write)	string	The full name, i.e. class name and namespaces, of the type being (de-)serialized.
MemberCount	(read only)	int	The count of members available in this instance

StreamingContextStates

Besides the SerializationInfo, GetObjectData() also receives a context object that provides a hint about the destination of the serialized object in the form of a StreamingContextStates value. With this piece of information an object can choose the representation that is most efficient for the environment where it will be deserialized. For example, if the object will be re-created in another process currently running on the same machine, the object can pass system-wide handles, to files for example, verbatim thus avoiding extra overhead re-opening the file. If the object is intended to be deserialized on another machine, where the handle is not valid, the object can store a UNC path the receiving application can access instead of the handle. Table 12.5 shows the possible values of the StreamingContextStates enumeration.

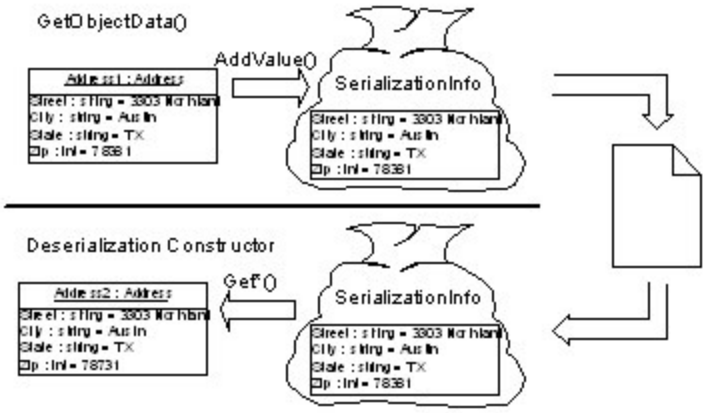
1.8 Table 12.3 The values of the StreamingContextStates enumeration identify the source or the destination of a serialized object. The values can be combined.

StreamingContextStates Value	Description
All	The serialized data has to be valid in all contexts.
Clone	The object graph is cloned and stays within the same process. The cloned graph has access to the same handles and unmanaged resources as the original graph.
CrossAppDomain	The source or the destination is in a different AppDomain.
CrossMachine	The source or destination is on a different computer. The serialized data must not be machine specific.
CrossProcess	The source or the destination is a different process on the same computer. Machine specific data is permitted.
File	The source or the destination is a file. The serialized data must not be transient, process or machine specific.
Other	The source or the destination is unknown.
Persistence	The source or the destination is a persisted store, e.g a database, files, or another form of storage. The serialized data must not be transient, process or machine specific.
Remoting	The source or the destination is accessed through remoting, but the location is unknown. The serialized data must not be transient, process or machine specific.

Deserializing Objects with ISerializable

When you looked at the definition of the ISerializable interface in the previous paragraph you might have scratched your head and wondered why it does not define a method to customize deserialization. What's all the flexibility worth if we do not have the deserialization counterpart to GetObjectData()? If it's not defined in the interface how does it work? These are valid questions. Their answer lies only in the semantics of an interface definition. There very much is a counterpart to the GetObjectData() method, but the .NET team chose to require it in the form of a constructor, rather than adding a SetObjectData() method to the interface. An interface cannot express this requirement, because it can only define method signatures, not class constructors. Implementating this functionality in a constructor helps avoiding issues related to multiple, possibly even concurrent, calls to an interface method. The downside of this design is that we cannot rely on the compiler to detect a missing deserialization constructor. Instead, we have to guard deserialization operations with an exception handler block to make sure ill designed objects will not crash our applications.

You may feel somewhat uneasy to expose a constructor that allows direct access to all members of the class. After all, this bypasses all encapsulation and control mechanisms you carefully set up through overloaded constructors. One step to protect ourselves from illegitimate



use of this constructor is to always declare the deserialization constructor protected instead of public. At least protected access to the constructor prevents explicit use of this constructor to instantiate objects. The serialization framework is not affected by this access restriction because it calls the constructor through the reflection API.

The signature of the deserialization constructor is identical to the signature of the GetObjectData() method, it receives a SerializationInfo object and a StreamingContext object. The SerializationInfo object contains the same name-value pairs we added in GetObjectData().

This time the StreamingContext object, provides information about the origin of the serialized data, i.e. whether it came from a live object running on the same machine or if it was received over a network or from a file. Once again we can leverage this information to optimize the overall serialization process like we have already seen when we already seen when we discussed implementing GetObjectData().

Figure 12.1 The SerializationInfo object serves as a container for the data we want to serialize. We fill the container in the GetObjectData) method of the ISerializable interface. When the formatter deserializes the object it hands us the container to retrieve the data we put into it.

The following example class in listing 12.1 below shows how we can implement the ISerializable interface in order to persist a non-serializable SqlConnection object. We wrap the SqlConnection object with a SerializableSqlConnection class that implements ISerializable. The first choice to make a non-serializable SqlConnection would be to derive a new class that implements ISerializable, but unfortunately the SqlConnection connection class is sealed and can not be extended. When we wrap a class we have to implement pass-through methods for each public property and method of the class. We will learn a better technique than wrapping a class in section 12.3, but for now we take a look how we can implement ISerializable.

Our GetObjectData() implementation persists enough information to create a new SqlConnection object that connects to the same database as the serialized object. It calls the AddValue() method to add the connection string and the connection state to a SerializationInfo object. The formatter in use will write the two values to output stream. Later on, when we deserialize an object from the persisted data, the formatter will populate a SerializationInfo object with the name-value pairs the original object persisted in GetObjectData(). It passes the new SerializationInfo to the deserialization constructor. The constructor can retrieve the values by their name through a number of type-safe Get methods exposed by the SerializationInfo class. Table 12.6 shows the complete list of these Get methods.

20 Listing 12.1 This class serializes a SqlConnection object, which is not marked [Serializable]

```
using System;
using System.Runtime.Serialization; // for ISerializable
using System.Data.SqlClient; // for SqlConnection
using System.Data; // for ConnectionState

namespace Christoph.Simple
{
  [Serializable]                                          | #1
  public class SerializableSqlConnection : ISerializable  |
  {
    private SqlConnection _DbConnection;                        | #2
    protected SerializableSqlConnection (
      SerializationInfo info, StreamingContext context)         | #3
    {                                                           |
      try                                                       |
      {                                                         |
        string connectionString = info.GetString( "DbString" );|
        _DbConnection = new SqlConnection( connectionString );  |
        if( ConnectionState.Open                                |
          != (ConnectionState)info.GetValue("DbState",          |
            typeof(ConnectionState)) )                          |
        {                                                       |
          _DbConnection.Open();                                 |
        }                                                       |
      }                                                         |
      catch( SerializationException ex )
      {
        Console.WriteLine("exception ex {0}", ex.Message );
        Console.WriteLine("exception ex {0}", ex.StackTrace );
      }
    } // SerializableSqlConnection

    public void GetObjectData(
      SerializationInfo info, StreamingContext context )            |#4
    {                                                               |
      if( null != _DbConnection )                                   |
      {                                                             |
        info.AddValue( "DbString", _DbConnection.ConnectionString );|
        info.AddValue( "DbState", _DbConnection.State );            |
      }                                                             |
    }  // GetObjectData                                            |

   // more useful code omitted …
 } // class PersistableSqlConnection
} // namespace
```

(annotation) <#1 Mark the class serializable and declare that it handles its own serialization.>
(annotation) <#2 The SqlConnection class is not serializable.>
(annotation) <#3 The deserialization constructor is called to restore all the serialized members. The constructor retrieves the stored values from SerializationInfo object to create a new SqlConnection object. If the original connection was open at the time it was serialized, the new connection is opened. We can declare the constructor protected to guard against explicit use.>
(annotation) <#4 The GetObjectData() method stores enough information to restore the SqlConnection.>

There is one more detail we have to know about in order to correctly implement a deserialization constructor. We must not execute any methods on any objects we retrieve from the SerializationInfo container. The .NET Framework does not guarantee that these objects are fully constructed and initialized when it is calling the deserialization constructor. In the example above, we can call Open() on the connection object only because we instantiated it ourselves, we did not retrieve it from the SerializationInfo.

1.9 Table 12.6 The SerializationInfo exposes methods to add and retrieve name-value pairs to describe an object.

Method	Description
public void AddValue(string name, XXX value);	Adds a name-value pair to the SerializationInfo. Several overloads are available to add all types to the SerializationInfo
public bool GetBoolean(string name);	Retrieves a Boolean value from the SerializationInfo.
public byte GetByte(string name);	Retrieves an 8-bit unsigned integer value from the SerializationInfo.
public char GetChar(string name);	Retrieves a Unicode character value from the SerializationInfo.
public DateTime GetDateTime(string name);	Retrieves a DateTime value from the SerializationInfo.
public decimal GetDecimal(string name);	Retrieves a Decimal value from the SerializationInfo.
public double GetDouble(string name);	Retrieves a double-precision value from the SerializationInfo.
public SerializationInfoEnumerator GetEnumerator(string name);	Returns an SerializationInfoEnumerator to iterate over the name-value pairs in the SerializationInfo.
public short GetInt16(string name);	Retrieves a 16-bit signed integer value from the SerializationInfo.
public int GetInt32(string name);	Retrieves a 32-bit signed integer value from the SerializationInfo.
public long GetInt64(string name);	Retrieves a 64-bit signed integer value from the SerializationInfo.
public sbyte GetSByte(string name);	Retrieves an 8-bit signed integer value from the SerializationInfo.
public float GetSingle(string name);	Retrieves a single-precision value from the SerializationInfo.
public string GetString(string name);	Retrieves a String value from the SerializationInfo.
public ushort GetUInt16(string name);	Retrieves a 16-bit unsigned integer value from the SerializationInfo.
public uint GetUInt32(	Retrieves a 32-bit unsigned integer value from the SerializationInfo.

string name);	
public ulong GetUInt64(string name);	Retrieves a 64-bit unsigned integer value from the SerializationInfo.
public object GetValue(string name);	Retrieves a value of any type from the SerializationInfo.
public void SetType(Type type);	Sets the Type to appear in the serialized output. This type is instantiated when the object is deserialized.

ISerializable

In the previous section I mentioned how we can derive from a non-serializable class and implement ISerializable. If we do that we must be aware that most likely there are reasons why the class is not serializable in the first place. We have to be very careful how we reconstruct the object. Extending a class that already implements ISerializable, however, creates a different scenario that deserves some special attention. It is perfectly compliant with the inheritance rules of the .NET type system to extend a class that implements ISerializable without providing a new implementation of GetObjectData() or the deserialization constructor. However, the serialization formatters in the .NET Framework will always delegate control over the serialization process to the object when they detect an ISerializable implementation, even if the implementation belongs to a base class higher up in the inheritance hierarchy. Since an implementation of GetObjectData() in a base class can not know how to serialize any members defined in derived classes it will obviously not serialize them. We will get some strange results if we miss to implement GetObjectData() and/or the deserialization constructor. Unfortunately the compiler can not even warn us if we missed implementing either one of these methods because the code is syntactically valid.

Also, when we inherit from a class that already implements ISerializable the base class' implementation is no longer called automatically. Instead, our new implementation of GetObjectData() is in control of the serialization process and with control comes responsibility: Our GetObjectData() method has the responsibility to serialize the entire class hierarchy. The easiest way to fulfill this responsibility is to call the base class' implementation of GetObjectData(), as seen in the following listing.

21 Listing 12.2 A class derived from a class implement ISerializable has to implement ISerializable as well.

```
[Serializable]
public class BaseClass : ISerializable
{
      public BaseClass (){}

      public BaseClass (SerializationInfo info,          |
        StreamingContext context)                        | #1
      {                                                  |
            // …                                         |
      }                                                  |
      public void GetObjectData( SerializationInfo info, |
```

```
        StreamingContext context )                          |
      {                                                     |
            // …                                            |
      }                                                     |
}

[Serializable]
public class DerivedClass :  BaseClass, ISerializable    | #2
{
      public DerivedClass (){}

      public DerivedClass (SerializationInfo info,          |
       StreamingContext context)                            | #3
      {                                                     |
            base( info, context );                          |  | #4
            // …                                            |
      }                                                     |
      public new void GetObjectData( SerializationInfo info,|
        StreamingContext context )                          |
      {                                                     |
            base.GetObjectData( info, context );            |  | #4
            // …                                            |
      }                                                     |
}
```

(annotation) <#1 Base class implements GetObjectData() and a deserialization constructor.>
(annotation) <#2 The derived class has to be marked serializable and declare ISerializable.>
(annotation) <#3 The derived class has to provide implementations for GetObjectData() and the deserialization constructor.>
(annotation) <#4 We delegate serialization and deserialization of the base classes to the base class implementations.>

Finishing Deserialization

If a class requires additional action to complete the initialization of an instance after all fields are assigned, it can implement the IDeserializationCallback interface. The interface defines the OnDeserialization()method, which the formatter calls after all referenced objects are ready to go.

Note that all [Serializable] classes can choose to implement the IDeserializationCallback interface. It is not restricted to classes that also implement ISerializable, for example we can also implement the interface to initialize any fields marked with the [NonSerialized] attribute after the formatter populated all other class fields.

Versioning

Object serialization in the .NET Framework does not provide a built-in versioning scheme. This can quickly become a problem when our applications attempt to deserialize an outdated version of a class like in the following scenario. Imagine we added more fields to a new version of a class. When we shut down our system to upgrade to the new version, all application objects serialize their state to persistant storage. Then we install the new version and boot up the system. All objects serialized before the switch-over are now deserialized, but no values are available for the newly added fields. The serialization formatter throws an exception when it tries to find a value for a new field and our system will not even boot up. Now what? We cannot let that happen. We have to protect our classes (and our jobs) by adding and make them resistant to

version changes. The best (and only) way to accomplish robust versioning functionality in the .NET Framework is to implement the ISerializable interface when we add fields to new versions and manually control deserialization to handle missing values in the deserialization constructor.

SurrogateSelectors

Section 12.2.2 demonstrated techniques how we can serialize objects that are not marked serializable. We can either derive a new class that implements the ISerializable interface or, if the class does not allow us to derive from it, we can wrap it in a new class that will handle serialization. Both techniques have their drawbacks. Maybe we cannot replace a class with a different class everywhere it is referenced and wrapping an object does not only introduce a completely unrelated type it also is pretty cumbersome.

Implementing SerializationSurrogates

The .NET Framework provides one more hook to customize object serialization that allows full customization if the whole serialization process and is completely transparent to the serialized objects. This solution requires a little bit more coding than implementing the ISerializable interface, but it will allow us to control serialization (and deserialization) for every class in the system, regardless if it is marked Serializable. The trick is to reroute the whole serialization process to a different class, the serialization surrogate. This class has to implement the ISerializationSurrogate interface, which is similar in nature to the ISerializable interface. Table 12.7 shows the methods defined by the ISerializationSurrogate interface and their semantics.

1.10 Table 12.7 A serialization surrogate takes over serialization for all instances of a certain class. The surrogate has to implement the ISerializationSurrogate interface with these methods.

Method	Description
void GetObjectData(object *obj*, SerializationInfo *info*, StreamingContext *context*);	Fills the SerializationInfo *info* with the data to serialize the object *obj*. The StreamingContext contains information to tune the serialized data to the intended deserialization scenario.
object SetObjectData(object *obj*, SerializationInfo *info*, StreamingContext *context*, ISurrogateSelector *selector*);	Retrieves data to deserialize the object *obj* from the SerializationInfo *info* . The StreamingContext contains information to identify source context of the serialized data. The *selector* parameter supplies the SurrogateSelector chain registered with this formatter.

The GetObjectData() and SetObjectData() methods perform the same functionality as GetObjectData() on the ISerializable interface and the deserialization constructor. The difference is that these methods are not implemented on the object that is serialized or deserialized. The parameters of both methods are identical to their counterparts in the ISerializable implementations. The SerializationInfo object serves as a container for the serialized information and the StreamingContext provides details about the context in which the operation executes. We already discussed these two classes in more detail in the previous section, now we can focus implementing ISerializationSurrogate.

Because ISerializationSurrogate is not implemented on the serialized object directly the SetObjectData() and GetObjectData() methods only have access to public fields and properties. In many cases the public fields provide enough information to recreate objects later on. However,

SetObjectData() will receive a fresh, completely un-initialized object that was created without(!) running any constructors. Likewise, no variable-initializers were executed when this object is created, all fields are set to null when the object is passed to SetObjectData(). You can probably imagine the catastrophic effects an uninitialized private field can have on the behavior of an object.

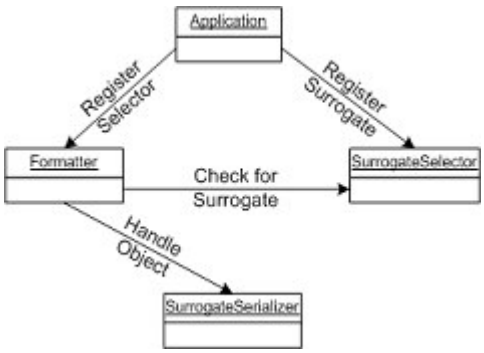
> WARNING: We must exercise extreme caution and thoroughly test to ensure deserialized objects are fully functional if we implement surrogates for 3rd party classes that do not persist all fields.

Now, there is a way to read and set private fields, but it is only available in fully trusted environments. Just as the runtime formatters, we can access private members through reflection. In order to do so our applications require all security privileges related to reflection. Also the penalty for accessing fields through reflection is very stiff, easily greater than 100x compared to assignments through writable properties, for example. However, by setting each member during deserialization we can guarantee an object will behave just like the one serialized.

Registering Serialization Surrogates

We have to register the surrogate with a formatter if we want the formatter to channel serialization for certain classes through our surrogates rather than checking the object for ISerializable or handling the class itself. However, the registration is managed by a surrogate selector, not by the formatter itself. The selector is a special container to register surrogates by type and serialization context. Yes, you can register different surrogates for different scenarios, e.g. one for long term object persistence in files and one to transmit objects across process boundaries on the same machine. This gives us the same flexibility for optimizing the serialization process as we have with the ISerializable interface. Furthermore, it allows very fine grained control over the scenarios in which we want to override a class' built-in serialization because we do not have to register a surrogate for all possible contexts. The scenarios are also identified by a StreamingContext object with the semantics we discussed in section 12.2 and table 12.3.

To register one or more selectors, the formatter exposes a property named SurrogateSelector. With surrogate selectors registered, the formatter iterates over all selectors to find a surrogate for the object type it has to process in the given context. If it can locate an appropriate surrogate it delegates serialization to the surrogate. Figure 12.1 illustrates this interaction between the



formatter, the surrogate selector and the serialization surrogate.

Figure 12.1 Several objects collaborate when we delegate serialization to a surrogate serializer: An application

registers a surrogate for a certain class with a surrogate selector. Every time a serialization formatter reads or writes an object, it checks with the surrogate selector if any surrogates are available for the given class in the current context. If the surrogate can provide a surrogate, all serialization activity as delegated to the surrogate.

A surrogate selector object has to implement the ISurrogateSelector interface (table 12.8) to interact with the runtime serialization formatters of the .NET Framework. The interface defines the GetSurrogate() method to retrieve serialization surrogates by object type and serialization scenario.

1.11 Table 12.8 Serialization formatters communicate with surrogate selectors over the ISurrogateSelector interface. The interface allows

Method	Description
public virtual void **ChainSelector(** ISurrogateSelector *selector* **);**	Adds a selector object to the chain of selector objects
public virtual ISurrogateSelector **GetNextSelector();**	Returns the next surrogate selector in the chain of selector objects.
public virtual ISerializationSurrogate **GetSurrogate(** Type *type*, StreamingContext *context*, **out** ISurrogateSelector *selector* **);**	Locate a surrogate for *type* and *context* in the chain of selector objects. The *selector* parameter references the surrogate selector containing the matching surrogate when the method returns.

The interface defines two additional methods, one to allow chaining multiple selector objects and one to enable the serialization formatter to traverse the chain. The interface does not define any methods to add surrogates to the selector's selection, but since the interface only defines the interaction between the serialization formatter and the surrogate selector a firm definition of this method is not required.

Now that you know how the ISurrogateSelector interface works, I can tell you rarely have to implement it because the .NET Framework already supplies a default implementation ready for us to use with the SurrogateSelector class. In addition to the methods mandated by the ISurrogateSelector interface this class also exposes the two methods shown in table 12.9 to add and remove surrogate objects.

1.12 Table 12.9 The SurrogateSelector class exposes methods to manage the contained serialization surrogates

Method	Description
public virtual void **AddSurrogate(** Type *type*, StreamingContext *context*, ISerializationSurrogate *surrogate* **);**	Adds the *surrogate* to use for an object of type *type* in the context specified by *context* to the slection.
public virtual void **RemoveSurrogate(** Type *type*, StreamingContext *context* **);**	Removes the surrogate for objects of type *type* and the context specified by *context* from the selection.

SerializationSurrogate

For the remainder of this chapter we will develop an example to demonstrate how we can register a serialization surrogate with a formatter. First, we need a class that the surrogate is going to serialize. The class is shown in the following listing (12.3).

22 Listing 12.3: A class without the Serializable attribute.

```
public sealed class NonSerializable
{
  public NonSerializable ()
  {
    Console.WriteLine( "NonSerializable ctor" );
  }

  private string _privateString;
  public string _publicString = "aPublicString";
}
```

There is nothing special about this class, actually, it's pretty useless. Nevertheless it will help us to understand how surrogates work. The best way to serialize this class is through a surrogate because it is not marked with the Serializable attribute and it is also declared sealed, therefore we cannot derive from it to make it serializable. You will come across many sealed classes when you get going with programming on the .NET platform. Knowing how to serialize them, even when they are not marked serializable, is very useful.

Next, we are going to write the surrogate. The surrogate class has to implement the ISerializationSurrogate methods: GetObjectData() and SetObjectData() for the formatter to delegate serialization and deserialization to the surrogate.

23 Listing 12.4: A serialization surrogate for the NonSerializable class. The surrogate accesses private data members of the NonSerializable objects through reflection.

```
using System;
using System.Reflection; // to access private data members
using System.Runtime.Serialization; // for ISerializationSurrogate and
                                    // related classes
// No [Serializable] required
public class NonSerializableSurrogate : ISerializationSurrogate
{
  public void GetObjectData(object obj,
    SerializationInfo info,
    StreamingContext context)
  {
    NonSerializable nsObj = obj as NonSerializable;          |#1
    if( nsObj != null )
    {
      info.AddValue( "PublicMember", nsObj._publicString );
      info.AddValue( "PrivateMember",
        nsObj.GetType().GetField("_privateString",           |#2
          BindingFlags.Instance                              |
            | BindingFlags.NonPublic ).GetValue( nsObj ) );  |
    }
  } // GetObjectData
```

```
  public object SetObjectData(object obj,
    SerializationInfo info,
    StreamingContext context,
    ISurrogateSelector selector)
  {
    NonSerializable nsObj = obj as NonSerializable;          |#3
    if( nsObj != null )
    {
      nsObj._publicString = info.GetString( "PublicMember" );
      nsObj.GetType().GetField("_privateString",             |#4
        BindingFlags.Instance                                |
          | BindingFlags.NonPublic ).SetValue(nsObj,         |
            info.GetString( "PrivateMember" ) );             |
    }
    return obj;                                              |#5
  }
}
```

(annotation) <#1 Safety check that we are really handling the correct object type.>
(annotation) <#2 Retrieve the value of the private data field through type refelection.>
(annotation) <#3 Another safety check to make sure we are deserializing the correct object type.>
(annotation) <#4 Set the value of the private field through the reflection API.>
(annotation) <#5 Return the object after all the values are set.>

This GetObjectData() implementation looks very much like the GetObjectData() method in the example for the ISerializable interface. The only difference is that the serialized object is passed in as a parameter. The SerializationInfo object serves once more as the container for all the data we want to serialize. We call the AddValue() method to add the objects we want to serialize to the container. The SerializationInfo object will do everything else: check if any surrogates are registered for the added objects, check the objects for ISerializable and finally hand everything off to the formatter object. The formatter classes then handle all the gritty details about how the objects within the SerializationInfo are persisted and recreated upon deserialization. When it is time to deserialize the object we can retrieve all the stored values through the various Get* methods from the SerializationInfo object passed to SetObjectData(). Retrieving objects by calling GetValue() will also ensure that all retrieved objects are properly deserialized as well.

Our surrogate class above accesses the private field of the NonSerializable class through reflection. Sometimes this might be our only solution to properly handle 3rd party classes, but in general we should design serializable classes and surrogates to avoid reflection in favor of properties for example. Accessing fields through reflection bears a huge overhead compared to direct access or access through properties. Reading a private field through reflection, for example, can be more than 150x slower than reading it through a property.

The last step before we can serialize and deserialize objects with our great, new surrogate is to create a serialization formatter and register the surrogate with it. The follwing example creates a custom formatter class that can always serialize objects of the NonSerializable class. The SoapFormatterWithSurrogate wraps the SoapFormatter class, and performs the registration of the surrogate and the surrogate selector in the constructor. First, we register the surrogate with a SurrogateSelector and specify the scenarios in which the formatter should delegate all action to this surrogate. Our example calls AddSurrogate() method with a StreamingContext object

initialized with StreamingContextStates.All to register the surrogate for all serialization scenarios because the NonSerializable class can never be serialized on its own. Finally we pass the SurrogateSelector to the constructor of the SoapFormatter object and the formatter is ready to go. Every time a NonSerializable object is serialized (and deserialized) with a SoapFormatterWithSurrogate, the surrogate will automatically handle persisting and restoring the data.

24 Listing 12.5: Registering a SerializerSurrogate.

```
using System.Runtime.Serializtion;
public class SoapFormatterWithSurrogate
{
  private SoapFormatter _Formatter;
  public SoapFormatterWithSurrogate()
  {
    SurrogateSelector selector = new SurrogateSelector();
    selector.AddSurrogate( typeof( NonSerializable ),
      new StreamingContext( StreamingContextStates.All ),
        new NonSerializableSurrogate() );

    _Formatter =
      new SoapFormatter( selector,
        new StreamingContext( StreamingContextStates.All ) );
  }
  public void SerializeWithSurrogate(Stream destination,
    NonSerializable obj)
  {
    _Formatter.Serialize( destination, obj );
  }
  public NonSerializable DeserializeWithSurrogate(Stream source)
  {
    return (NonSerializable)_Formatter.Deserialize( source );
  }
}
```

Summary

Object serialization in the .NET Framework is valuable tool, not just in XML enabled .NET applications. This chapter demonstrated how we can develop serializable classes and how we can apply different techniques to override serialization behavior built into a class. The serialization format was not important in this chapter, because what learned is independent of the format. Yet it was important to understand the different aspects of serializable objects before we focus on serializing objects with the SoapFormatter. The SoapFormatter serializes objects into SOAP messages or parses SOAP messages and extracts serialized objects from the message.