CHAPTER 3

Data Representation

IN THIS CHAPTER

- Background
- `XmlSerializer` and `XmlFormatter`
- The XML Fetish
- Using the `XmlFormatter`
- Exception Handling

Background

The Windows Communication Foundation provides a language, called the *Service Model*, for modeling software communications. Software executables can be generated from models expressed in that language using the classes of a lower-level programming framework called the *Channel Layer*.

In the language of the Service Model, a piece of software that responds to communications over a network is a *service*. A service has one or more endpoints to which communications can be directed. An endpoint consists of an *address*, a *binding*, and a *contract*.

The role of the address component is simple. It specifies a unique location for the service on a network in the form of a uniform resource locator.

The binding specifies the protocols for communication between the client and the server. Minimally, the binding must specify a protocol for encoding messages and a protocol for transporting them.

A contract specifies the operations that are available at an endpoint. To define a contract, one merely writes an interface in one's preferred .NET programming language, and adds attributes to it to indicate that the interface is also a Windows Communication Foundation contract. Then, to implement the contract, one simply writes a class that implements the .NET interface that one has defined.

This contract was used to define the operations of a derivatives calculator service in Chapter 2, "The Fundamentals."

```
using System;
using System.Collections.Generic;
```

```
using System.ServiceModel;
using System.Text;

namespace DerivativesCalculator
{
    [ServiceContract]
    public interface IDerivativesCalculator
    {
        [OperationContract]
        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

      void DoNothing();
    }
}
```

One way of writing code for a client of that service that is to use the `CalculateDerivative()` operation would be to write this:

```
string[] symbols = new string[]{"MSFT"};
decimal[] parameters = new decimal[]{3};
string[] functions = new string[]{"TechStockProjections"};

IDerivativesCalculator proxy =
  new ChannelFactory<IDerivativesCalculator>("CalculatorEndpoint").
  CreateChannel();
proxy.CalculateDerivative(symbols, parameters, functions);
((IChannel)proxy).Close();
```

When the statement

```
proxy.CalculateDerivative(symbols, parameters, functions);
```

is executed, the data being provided by the client as input to the operation is added to an instance of the `Message` class of the Windows Communication Foundation's Channel Layer. Within the `Message` class, the data is represented in the form of an XML Information Set (InfoSet). When the data is ready to be transported from the client to the service, the message-encoding protocol specified in the binding will determine the form in which the `Message` object containing the data provided by the client will be represented to the service. The message-encoding protocol could conceivably translate the message into a string of comma-separated values, or into any format whatsoever. However, all the standard bindings specify encoding protocols by which the `Message` object will continue to be represented as an XML InfoSet. Depending on the encoding

protocol of the binding, that XML InfoSet may be encoded either using one of the standard XML text encodings, or the standard MTOM protocol, or using a binary format proprietary to the Windows Communication Foundation.

When a transmission is received by a Windows Communication Foundation service, it is reassembled as a `Message` object by a message-encoding binding element, with the data sent by the client being expressed within the `Message` object as an XML InfoSet, regardless of the format in which it was encoded by the client. The `Message` object will be passed to a component of the Windows Communication Foundation called the *dispatcher*. The dispatcher extracts the client's data items from the XML InfoSet. It invokes the method of the service that implements the operation being used by the client, passing the client's data items to the method as parameters.

`XmlSerializer` and `XmlFormatter`

From the foregoing, it should be apparent that data being sent from the client to the service is serialized to XML within the client, and deserialized from XML within the service. There are two XML serializers that the Windows Communication Foundation can use to accomplish that task.

One of those is a new XML serializer provided with the Windows Communication Foundation. It is the `XmlFormatter` class in the `System.Runtime.Serialization` namespace within the `System.Runtime.Serialization` assembly. That assembly is one of the constituent assemblies of the Windows Communication Foundation. The `XmlFormatter` class is the XML serializer that the Windows Communication Foundation uses by default.

Given a class like the one in Listing 3.1, one could make the class serializable by the `XmlFormatter` by adding `DataContract` and `DataMember` attributes as shown in Listing 3.2. The representation of instances of a class in XML that is implied by the addition of the `DataContract` attribute to the class, and the `DataMember` attributes to its members, is commonly referred to as a *data contract* in the argot of the Windows Communication Foundation.

LISTING 3.1 `DerivativesCalculation` Class

```
public class DerivativesCalculation
{
        private string[] symbols;
        private decimal[] parameters;
        private string[] functions;

        public string[] Symbols
        {
            get
            {
                return this.symbols;
            }
```

LISTING 3.1 Continued

```
            set
            {
                this.symbols = value;
            }
        }

        public decimal[] Parameters
        {
            get
            {
                return this.parameters;
            }

            set
            {
                this.parameters = value;
            }
        }

        public string[] Functions
        {
            get
            {
                return this.functions;
            }

            set
            {
                this.functions = value;
            }

        }
}
```

LISTING 3.2 `DerivativesCalculation` Data Contract

```
[DataContract]
public class DerivativesCalculation
{
        [DataMember]
        private string[] symbols;
        [DataMember]
        private decimal[] parameters;
```

LISTING 3.2 Continued

```
        [DataMember]
        private string[] functions;

        public string[] Symbols
        {
            get
            {
                return this.symbols;
            }

            set
            {
                this.symbols = value;
            }
        }

        public decimal[] Parameters
        {
            get
            {
                return this.parameters;
            }

            set
            {
                this.parameters = value;
            }
        }

        public string[] Functions
        {
            get
            {
                return this.functions;
            }

            set
            {
                this.functions = value;
            }

        }
}
```

Although the `XmlFormatter` is the default XML serializer of the Windows Communication Foundation, the Windows Communication Foundation can also be configured to do XML serialization using the `System.Xml.Serialization.XmlSerializer` class that has always been included in the `System.Xml` assembly of the .NET Framework Class Library. To exercise that option, add the `XmlSerializerFormat` attribute to the definition of a Windows Communication Foundation contract, like so:

```
namespace DerivativesCalculator
{
    [ServiceContract]
    [XmlSerializerFormat]
    public interface IDerivativesCalculator
    {
        [OperationContract]
        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

        void DoNothing();
    }
}
```

The option of using the `XmlSerializer` class for XML serialization can also be selected just for individual operations:

```
namespace DerivativesCalculator
{
    [ServiceContract]
    public interface IDerivativesCalculator
    {
        [OperationContract]
        [XmlSerializerFormat]
        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

        void DoNothing();
    }
}
```

The `XmlSerializer` provides very precise control over how data is to be represented in XML. Its facilities are well documented in the book *.NET Web Services: Architecture and Implementation*, by Keith Ballinger (2003).

The `XmlFormatter`, on the other hand, provides very little control over how data is to be represented in XML. It only allows one to specify the namespaces and names used to refer to data items in the XML, and the order in which the data items are to appear in the XML, as in this case:

```
[DataContract(Namespace="Derivatives",Name="Calculation")]
public class DerivativesCalculation
{
        [DataMember(Namespace="Derivatives",Name="Symbols",Order=0)]
        private string[] symbols;
        [DataMember(Namespace="Derivatives",Name="Parameters",Order=1)]
        private decimal[] parameters;}
        [...]
}
```


By not permitting any control over how data is to be represented in XML, the serialization process becomes highly predictable for the `XmlFormatter` and, thereby, more amenable to optimization. So a practical benefit of the `XmlFormatter`'s design is better performance, approximately 10% better performance.

The XML Fetish

One might wonder whether the gain in performance is worth the loss of control over how data is represented in XML. That is most certainly the case, and understanding why serves to highlight the brilliance of the design of the Windows Communication Foundation.

A practice that is most characteristic of service-oriented programming is commonly referred to as *contract-first development*. Contract-first development is to begin the construction of software by specifying platform-independent representations for the data structures to be exchanged across the external interfaces, and platform-independent protocols for those exchanges.

Contract-first development is a sound practice. It helps one to avoid such unfortunate mistakes as building software that is meant to be interoperable across platforms, but that emits data in formats for which there are only representations on a particular platform, such as the .NET DataSet format.

However, the sound practice of contract-first development has become confused with one particular way of doing contract-first development, by virtue of which people become excessively concerned with XML formats. That one particular way of doing contract-first development is to use an XML editor to compose specifications of data formats in the XML Schema language, taking care to ensure that all complex data types are ultimately defined in terms of XML Schema Datatypes. Now, as a software developer, one's sole interest in contract-first development should be in defining the inputs and outputs of one's software, and in ensuring that, if necessary, those inputs and outputs can be represented in a platform-independent format. Yet practitioners of contract-first development, working in the XML Schema language in an XML editor, tend to become distracted from

those core concerns and start to worry about exactly how the data is to be represented in XML. Consequently, they begin to debate, among other things, the virtues of various ways of encoding XML, and become highly suspicious of anything that might inhibit them from seeing and fiddling with XML. The XML becomes a fetish, falsely imbued with the true virtues of contract-first development, and, as Sigmund Freud wrote, "[s]uch substitutes are with some justice likened to the fetishes in which savages believe that their gods are embodied" (1977, 66).

With the `XmlFormatter`, the Windows Communication Foundation not only restores the focus of software developers to what should be important to them, namely, the specification of inputs and outputs, but also relocates control over the representation of data to where it properly belongs, which is outside of the code, at the system administrator's disposal. Specifically, given the class

```
public class DerivativesCalculation
{
     public string[] Symbols;
     public decimal[] Parameters;
     public string[] Functions;
     public DataTime Date
}
```

all one should care about as a software developer is to be able to say that the class is a data structure that may be an input or an output, and that particular constituents of that structure may be included when it is input or output. The `DataContract` and `DataMember` attributes provided for using the `XmlFormatter` to serialize data allow one to do just that, as in the following:

```
[DataContract]
public class DerivativesCalculation
{
     [DataMember]
     public string[] Symbols;
     [DataMember]
     public decimal[] Parameters;
     [DataMember]
     public string[] Functions;
     public DataTime Date
}
```

It is by configuring the encoding protocol in the binding of an endpoint that one can control exactly how data structures are represented in transmissions.

Now there are two scenarios to consider. In the first scenario, the organization that has adopted the Windows Communication Foundation is building a service. In the other scenario, the organization that has adopted the Windows Communication Foundation is building a client.

In the first of these scenarios, the Windows Communication Foundation developers define the data structures to be exchanged with their services using the `DataContract` and `DataMember` attributes. Then they generate representations of those structures in the XML Schema language using the Service Metadata Tool, introduced in the preceding chapter. They provide those XML Schema language representations to developers wanting to use their services. The designers of the Windows Communication Foundation have expended considerable effort to ensure that the structure of the XML into which the `XmlFormatter` serializes data should be readily consumable by the tools various vendors provide to assist in deserializing data that is in XML. Thus, anyone wanting to use the services provided by the Windows Communication Foundation developers in this scenario should be able to do so, despite the Windows Communication Foundation developers never having necessarily looked at, or manipulated, any XML in the process of providing the services.

In the second scenario, the Windows Communication Foundation developers use the Service Metadata Tool to generate code for using a software service that may or may not have been developed using the Windows Communication Foundation. If the XML representations of the inputs and outputs of that service deviate from the way in which the `XmlFormatter` represents data in XML, then the code generated by the Service Metadata Tool will include the switch for using the `XmlSerializer` instead of the `XmlFormatter` for serializing data to XML. That code should allow the Windows Communication Foundation developers to use the service, and, once again, they will not have had to look at or manipulate any XML in order to do so.

Should the fetish for XML prove too overwhelming, and one is compelled to look at the XML Schema language that defines how a class will be represented within XML, the Windows Communication Foundation does make provision for that. Executing the Service Metadata Tool in this way,

```
svcutil /datacontractonly SomeAssembly.dll
```

where *`SomeAssembly.dll`* is the name of some assembly in which a data contract is defined for a class, will yield the XML Schema language specifying the format of the XML into which instances of the class will be serialized.

The question being considered is whether the gain in performance yielded by the `XmlFormatter` is adequate compensation for its providing so little control over how data is represented in XML. The answer that should be apparent from the foregoing is that control over how data is represented in XML is generally of no use to software developers, so, yes, any gain in performance in exchange for that control is certainly welcome.

Using the `XmlFormatter`

To become familiar with the `XmlFormatter`, open the Visual Studio solution associated with this chapter that you downloaded from www.samspublishing.com. The solution contains a single project, called *Serialization*, for building a console application. All the code is in a single module, `Program.cs`, the content of which is shown in Listing 3.3. Note that there is a `using` statement for the `System.Runtime.Serialization` namespace, and also that the project references the `System.Runtime.Serialization` assembly.

LISTING 3.3 Program.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Runtime.Serialization;
using System.ServiceModel;
using System.Text;

namespace Serialization
{
    [DataContract(Name="Calculation")]
    public class ServiceViewOfData: IUnknownSerializationData
    {
        [DataMember(Name = "Symbols")]
        private string[] symbols;
        [DataMember(Name = "Parameters")]
        private decimal[] parameters;
        [DataMember(Name = "Functions")]
        private string[] functions;
        [DataMember(Name="Value")]
        private decimal value;

        private UnknownSerializationData unknownData;

        public string[] Symbols
        {
            get
            {
                return this.symbols;
            }

            set
            {
                this.symbols = value;
            }
        }

        public decimal[] Parameters
        {
            get
            {
                return this.parameters;
            }
```

LISTING 3.3 Continued

```
        set
        {
            this.parameters = value;
        }
    }

    public string[] Functions
    {
        get
        {
            return this.functions;
        }

        set
        {
            this.functions = value;
        }

    }

    public decimal Value
    {
        get
        {
            return this.value;
        }

        set
        {
            this.value = value;
        }
    }

    #region IUnknownSerializationData Members

    public UnknownSerializationData UnknownData
    {
        get
        {
            return this.unknownData;
        }
```

LISTING 3.3 Continued

```
            set
            {
                this.unknownData = value;
            }
        }

        #endregion
    }

    [DataContract]
    public class Data
    {
        [DataMember]
        public string Value;
    }

    [DataContract]
    public class DerivedData : Data
    {
    }

    [DataContract(Name = "Calculation")]
    public class ClientViewOfData : IUnknownSerializationData
    {
        [DataMember(Name = "Symbols")]
        public string[] Symbols;
        [DataMember(Name = "Parameters")]
        public decimal[] Parameters;
        [DataMember(Name="Functions")]
        public string[] Functions;
        [DataMember(Name="Value")]
        public decimal Value;
        [DataMember(Name = "Reference")]
        public Guid Reference;

        private UnknownSerializationData unknownData;


        public UnknownSerializationData UnknownData
        {
            get
            {
                return this.unknownData;
            }
```

LISTING 3.3 Continued

```
            set
            {
                this.unknownData = value;
            }
        }


    }


    [ServiceContract(Name = "DerivativesCalculator")]
    [KnownType(typeof(DerivedData))]
    public interface IServiceViewOfService
    {
        [OperationContract(Name="FromXSDTypes")]
        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

        [OperationContract(Name="FromDataSet")]
        DataSet CalculateDerivative(DataSet input);

        [OperationContract(Name = "FromDataContract")]
        ServiceViewOfData CalculateDerivative(ServiceViewOfData input);

        [OperationContract(Name = "AlsoFromDataContract")]
        Data DoSomething(Data input);

    }

    [ServiceContract(Name="DerivativesCalculator")]
    [KnownType(typeof(DerivedData))]
    public interface IClientViewOfService
    {
        [OperationContract(Name = "FromXSDTypes")]
        decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions);

        [OperationContract(Name = "FromDataSet")]
        DataSet CalculateDerivative(DataSet input);
```

LISTING 3.3 Continued

```
        [OperationContract(Name = "FromDataContract")]
        ClientViewOfData CalculateDerivative(ClientViewOfData input);

        [OperationContract(Name = "AlsoFromDataContract")]
        Data DoSomething(Data input);
    }

    public class DerivativesCalculator : IServiceViewOfService
    {
        #region IDerivativesCalculator Members

        public decimal CalculateDerivative(
            string[] symbols,
            decimal[] parameters,
            string[] functions)
        {
            return (decimal)(System.DateTime.Now.Millisecond);
        }


        public DataSet CalculateDerivative(DataSet input)
        {
            if (input.Tables.Count > 0)
            {
                if (input.Tables[0].Rows.Count > 0)
                {
                    input.Tables[0].Rows[0]["Value"] =
                        (decimal)(System.DateTime.Now.Millisecond);
                }
            }
            return input;
        }

        public ServiceViewOfData CalculateDerivative(ServiceViewOfData input)
        {
            input.Value = this.CalculateDerivative(
                input.Symbols,
                input.Parameters,
                input.Functions);
            return input;
        }

        public Data DoSomething(Data input)
        {
```

LISTING 3.3 Continued

```
            return input;
        }

        #endregion
    }

    public class Program
    {
        public static void Main(string[] args)
        {
            using (ServiceHost host = new ServiceHost(
                typeof(DerivativesCalculator),
                new Uri[] {
                    new Uri("http://localhost:8000/Derivatives") }))
            {
                host.AddServiceEndpoint(
                    typeof(IServiceViewOfService),
                    new BasicHttpBinding(),
                    "Calculator");
                host.Open();

                Console.WriteLine("The service is available.");

                string address =
                    "http://localhost:8000/Derivatives/Calculator";

                ChannelFactory<IClientViewOfService> factory =
                    new ChannelFactory<IClientViewOfService>(
                        new BasicHttpBinding(),
                        new EndpointAddress(
                            new Uri(address)));
                IClientViewOfService proxy =
                    factory.CreateChannel();

                decimal result = proxy.CalculateDerivative(
                    new string[] { "MSFT" },
                    new decimal[] { 3 },
                    new string[] { "TechStockProjection" });
                Console.WriteLine(
                  "Value using XSD types is {0}.",
                  result);

                DataTable table = new DataTable("InputTable");
                table.Columns.Add("Symbol", typeof(string));
```

LISTING 3.3 Continued

```
table.Columns.Add("Parameter", typeof(decimal));
table.Columns.Add("Function", typeof(string));
table.Columns.Add("Value", typeof(decimal));

table.Rows.Add("MSFT", 3, "TechStockProjection",0.00);

DataSet input = new DataSet("Input");
input.Tables.Add(table);

DataSet output = proxy.CalculateDerivative(input);
if (output != null)
{
    if (output.Tables.Count > 0)
    {
        if (output.Tables[0].Rows.Count > 0)
        {
            Console.WriteLine(
                "Value using a DataSet is {0}.",
                output.Tables[0].Rows[0]["Value"]);
        }
    }
}

ClientViewOfData calculation =
    new ClientViewOfData();
calculation.Symbols =
    new string[] { "MSFT" };
calculation.Parameters =
    new decimal[] { 3 };
calculation.Functions =
    new string[] { "TechStockProjection" };
calculation.Reference =
    Guid.NewGuid();

Console.WriteLine(
    "Reference is {0}.", calculation.Reference);

ClientViewOfData calculationResult =
    proxy.CalculateDerivative(calculation);
Console.WriteLine(
  "Value using a Data Contract is {0}.",
  calculationResult.Value);
```

LISTING 3.3 Continued

```
                Console.WriteLine(
                    "Reference is {0}.", calculationResult.Reference);

                DerivedData derivedData = new DerivedData();
                Data outputData = proxy.DoSomething(derivedData);

                MemoryStream stream = new MemoryStream();
                XmlFormatter formatter = new XmlFormatter();
                formatter.Serialize(stream, calculation);
                Console.WriteLine(
                  UnicodeEncoding.UTF8.GetChars(stream.GetBuffer()));

                Console.WriteLine("Done.");

                ((IChannel)proxy).Close();

                host.Close();

                Console.ReadKey();

            }
        }
    }
}
```

In Listing 3.3, the static `Main()` method of the class called `Program` both provides a host for a Windows Communication Foundation service and uses that service as a client. The hosting of the service is accomplished with this code:

```
using (ServiceHost host = new ServiceHost(
  typeof(DerivativesCalculator),
  new Uri[] {
    new Uri("http://localhost:8000/Derivatives") }))
{
    host.AddServiceEndpoint(
      typeof(IServiceViewOfService),
      new BasicHttpBinding(),
      "Calculator");
    host.Open();
    [...]
    host.Close();
    [...]
}
```

In this code, an endpoint, with its address, binding, and contract, is added to the service programmatically, rather than through the host application's configuration, as was done in the preceding chapter. Similarly, the client code directly incorporates information about the service's endpoint, rather than referring to endpoint information in the application's configuration:

```
string address =
  "http://localhost:8000/Derivatives/Calculator";
ChannelFactory<IClientViewOfService> factory =
  new ChannelFactory<IClientViewOfService>(
    new BasicHttpBinding(),
    new EndpointAddress(
    new Uri(address)));
IClientViewOfService proxy =
  factory.CreateChannel();
```

This imperative style of programming with the Windows Communication Foundation is used here for two reasons. The first is simply to show that this style is an option. The second, and more important, reason is to make the code that is discussed here complete in itself, with no reference to outside elements such as configuration files.

The contract of the service is defined in this way:

```
[ServiceContract(Name = "DerivativesCalculator")]
[...]
public interface IServiceViewOfService
{
  [...]
}
```

The client has a separate definition of the contract of the service, but the definition used by the client and the definition used by the service are semantically identical:

```
[ServiceContract(Name="DerivativesCalculator")]
[...]
public interface IClientViewOfService
{
  [...]
}
```

In both the client's version of the contract and the service's version, there is this definition of an operation:

```
[OperationContract(Name="FromXSDTypes")]
decimal CalculateDerivative(
  string[] symbols,
  decimal[] parameters,
  string[] functions);
```

When the client uses that operation, like this,

```
decimal result = proxy.CalculateDerivative(
  new string[] { "MSFT" },
  new decimal[] { 3 },
  new string[] { "TechStockProjection" });
Console.WriteLine(
  "Value using XSD types is {0}.",
  result);
```

its attempt to do so works, which can be verified by running the application built from the solution. In this case, the inputs and outputs of the operation are .NET types that correspond quite obviously to XML Schema Datatypes. The `XmlFormatter` automatically serializes the inputs and outputs into XML.

The next operation defined in both the client's version of the contract and the service's version is this one:

```
[OperationContract(Name = "FromDataSet")]
DataSet CalculateDerivative(DataSet input);
```

The client uses that operation with this code:

```
DataTable table = new DataTable("InputTable");
table.Columns.Add("Symbol", typeof(string));
table.Columns.Add("Parameter", typeof(decimal));
table.Columns.Add("Function", typeof(string));
table.Columns.Add("Value", typeof(decimal));

table.Rows.Add("MSFT", 3, "TechStockProjection",0.00);

DataSet input = new DataSet("Input");
input.Tables.Add(table);

DataSet output = proxy.CalculateDerivative(input);
[...]
Console.WriteLine(
  "Value using a DataSet is {0}.",
 output.Tables[0].Rows[0]["Value"]);
```

In this case, the input and output of the operation are both .NET DataSet objects, and the .NET DataSet type does not obviously correspond to any XML Schema Datatype. Nevertheless, the `XmlFormatter` automatically serializes the input and output to XML, as it will for any .NET type that implements `ISerializable`. Of course, passing .NET DataSets around is a very bad idea if one can anticipate a non-.NET client needing to participate, and it is never wise to rule that out as a possibility.

In the service's version of the contract, this operation is included:

```
[OperationContract(Name = "FromDataContract")]
ServiceViewOfData CalculateDerivative(ServiceViewOfData input);
```

The input and output of this operation is an instance of the `ServiceViewOfData` class, which is defined like so:

```
 [DataContract(Name="Calculation")]
public class ServiceViewOfData: IUnknownSerializationData
{
  [DataMember(Name = "Symbols")]
  private string[] symbols;
  [DataMember(Name = "Parameters")]
  private decimal[] parameters;
  [DataMember(Name = "Functions")]
  private string[] functions;
  [DataMember(Name="Value")]
  private decimal value;
  [...]
}
```

The client's version of the contract defines the corresponding operation in this way:

```
[OperationContract(Name = "FromDataContract")]
ClientViewOfData CalculateDerivative(ClientViewOfData input);
```

Here, the input and output are of the `ClientViewOfData` type, which is defined in this way:

```
[DataContract(Name = "Calculation")]
public class ClientViewOfData : IUnknownSerializationData
{
  [DataMember(Name = "Symbols")]
  public string[] Symbols;
  [DataMember(Name = "Parameters")]
  public decimal[] Parameters;
  [DataMember(Name="Functions")]
  public string[] Functions;
  [DataMember(Name="Value")]
  public decimal Value;
  [DataMember(Name = "Reference")]
  public Guid Reference;
}
```

The service's `ServiceViewOfData` class and the client's `ClientViewOfData` class are used to define data contracts that are compatible with one another. The data contracts are

compatible because they have the same namespace and name, and because the members that have the same names in each version of the contract also have the same types. Because of the compatibility of the data contracts used in the client's version of the operation and the service's version, those operations that the client and the service define in different ways are also compatible with one another.

The client's version of the data contract includes a member that the service's version of the data contract omits: the member named Reference. However, the service's version implements the Windows Communication Foundation's IUnknownSerializationData interface, thus:


```
[DataContract(Name="Calculation")]
public class ServiceViewOfData: IUnknownSerializationData
{
  [...]
  private UnknownSerializationData unknownData;
  [...]
  public UnknownSerializationData UnknownData
  {
    get
    {
      return this.unknownData;
    }

    set
    {
      this.unknownData = value;
    }
  }
}
```

By implementing the IUnknownSerializationData interface, it sets aside some memory that the XmlFormatter can use for storing and retrieving the values of members that other versions of the same contract might include. In this case, that memory is named by the variable called unknownData. Thus, when a more advanced version of the same data contract is passed to service, with members that the service's version of the data contract does not include, the XmlFormatter is able to pass the values of those members through the service. In particular, when the client calls the service using this code,

```
ClientViewOfData calculation =
  new ClientViewOfData();
calculation.Symbols =
  new string[] { "MSFT" };
calculation.Parameters =
  new decimal[] { 3 };
calculation.Functions =
  new string[] { "TechStockProjection" };
```

```
calculation.Reference =
  Guid.NewGuid();

Console.WriteLine(
  "Reference is {0}.", calculation.Reference);

ClientViewOfData calculationResult =
  proxy.CalculateDerivative(calculation);
Console.WriteLine(
  "Value using a Data Contract is {0}.",
  calculationResult.Value);

Console.WriteLine(
  "Reference is {0}.", calculationResult.Reference);
```

not only is the `XmlFormatter` able to serialize the custom type, `ClientViewOfData`, to XML for transmission to the service, but the member called `Reference` that is in the client's version of the data contract, but not in the service's version, passes through the service without being lost.

Two things should be evident from this case. First, the `DataContract` and `DataMember` attributes make it very easy to provide for the serialization of one's custom data types by the Windows Communication Foundation's `XmlFormatter`. Second, implementing the Windows Communication Foundation's `IUnknownSerializationData` interface is always a good idea, because it allows different versions of the same data contract to evolve independently of one another, yet still be usable together.

The last operation defined for the service is this one, which is defined in the same way both in the code used by the service and in the code used by the client:

```
[OperationContract(Name = "AlsoFromDataContract")]
Data DoSomething(Data input);
```

The input and output of the operation are of the custom `Data` type, which is made serializable by the `XmlFormatter` through the use of the `DataContract` and `DataMember` attributes, thus:

```
[DataContract]
public class Data
{
  [DataMember]
  public string Value;
}
```

The client uses that operation with this code:

```
DerivedData derivedData = new DerivedData();
Data outputData = proxy.DoSomething(derivedData);
```

That code passes an instance of the DerivedData type to the service, a type which is derived from the Data class, in this way:

```
[DataContract]
public class DerivedData : Data
{
}
```

What will happen in this case is that the XmlFormatter, in deserializing the data received from the client on behalf of the service, will encounter the XML into which an instance of the DerivedData class had been serialized, when it will be expecting the XML into which an instance of the Data class has been serialized. That will cause the XmlFormatter to throw an exception. However, both the service's version of the endpoint's contract and the client's version have a KnownType attribute that refers to the DerivedData class:

```
[ServiceContract(Name = "DerivativesCalculator")]
[KnownType(typeof(DerivedData))]
public interface IServiceViewOfService
{
  [...]
}
[...]
[ServiceContract(Name="DerivativesCalculator")]
[KnownType(typeof(DerivedData))]
public interface IClientViewOfService
{
  [...]
}
```

That attribute prepares the XmlFormatter to accept the XML for a DerivedData object whenever it is expecting the XML for an instance of any type from which the DerivedData class derives. So, by virtue of that attribute being added to the definition of the service's contract, when the XmlFormatter encounters XML for a DerivedData object when it is expecting XML for a Data object, it is able to deserialize that XML into an instance of the DerivedData class. It follows that if one was to define an operation in this way,

```
[OperationContract]
void DoSomething(object[] inputArray);
```

then one should add to the service contract a KnownType attribute for each of the types that might actually be included in the input parameter array.

That the KnownType attribute has to be added in order for the DerivedData objects to be successfully deserialized shows that one should avoid using inheritance as a way of versioning data contracts. If a base type is expected, but a derived type is received, serialization of the derived type will fail unless the code is modified with the addition of the KnownType attribute to anticipate the derived type.

The Windows Communication Foundation uses the `XmlFormatter` invisibly. So these remaining lines of client code in the sample simply show that it can be used separately from the rest of the Windows Communication Foundation for serializing data to XML:

```
MemoryStream stream = new MemoryStream();
XmlFormatter formatter = new XmlFormatter();
formatter.Serialize(stream, calculation);
Console.WriteLine(
  UnicodeEncoding.UTF8.GetChars(stream.GetBuffer()));
```

Exception Handling

Data contracts also assist in being able to notify clients of exceptions that may occur in a service. Too see how that works, follow these steps:

1. Add this class to the `Program.cs` module of the Serialization project referred to previously:

   ```
   public class SomeError
   {
     public string Content;
   }
   ```

2. Create a data contract from that class using the `DataContract` and `DataMember` attributes:

   ```
   [DataContract]
   public class SomeError
   {
     [DataMember]
     public string Content;
   }
   ```

   This yields a data contract that specifies the format of a simple error message that the service might send to the client.

3. Add an operation to the `IServiceViewOfService` interface that defines the service's version of the service's contract:

   ```
   [OperationContract(Name="Faulty")]
   decimal DivideByZero(decimal input);
   ```

4. Add a fault contract to the operation, informing clients that they should anticipate that, instead of returning the expected result, the service may return an error message of the form defined by the `SomeError` data contract:

   ```
   [OperationContract(Name="Faulty")]
   [FaultContract(typeof(SomeError))]
   decimal DivideByZero(decimal input);
   ```

5. Add an implementation of the `DivideByZero()` method to the `DerivativesCalculator` class, which is the service type that implements the `DerivativesCalculator` service contract defined by the `IServiceViewOfService` interface:

```
public class DerivativesCalculator : IServiceViewOfService
{
  [...]
  public decimal DivideByZero(decimal input)
  {
    try
    {
      decimal denominator = 0;
      return input / denominator;
    }
    catch (Exception exception)
    {
      SomeError error = new SomeError();
      error.Content = exception.Message;
      throw new FaultException<SomeError>(error);
    }
  }
}
```

   By virtue of this code, when the service traps an exception in the `DivideByZero()` method, it creates an instance of the `SomeError` class to convey selected information about the exception to the caller. That information is then sent to the caller using the Windows Communication Foundation's generic, `FaultException<T>`.

6. Because of the `FaultContract` attribute on the `DivideByZero()` method, if the metadata for the service was to be downloaded, and client code generated from it using the Service Metadata Tool, the client's version of the contract would automatically include the definition of the `DivideByZero()` method, and its associated fault contract. However, in this case, simply add the method and the fault contract to the client's version of the contract, which is in the `IClientViewOfService` interface:

```
[ServiceContract(Name="DerivativesCalculator")]
[KnownType(typeof(DerivedData))]
public interface IClientViewOfService
{
  [...]
  [OperationContract(Name = "Faulty")]
  [FaultContract(typeof(SomeError))]
  decimal DivideByZero(decimal input);
}
```

7. Now have the client use the `Faulty` operation by adding code to the static `Main()` method of the `Program` class, as shown in Listing 3.4.

LISTING 3.4 Anticipating a Fault

```
public class Program
{
  public static void Main(string[] args)
  {
    using (ServiceHost host = new ServiceHost(
      typeof(DerivativesCalculator),
        new Uri[] {
          new Uri("http://localhost:8000/Derivatives") }))
    {
      host.AddServiceEndpoint(
        typeof(IServiceViewOfService),
        new BasicHttpBinding(),
        "Calculator");
      host.Open();

      Console.WriteLine("The service is available.");

      string address =
        "http://localhost:8000/Derivatives/Calculator";

      ChannelFactory<IClientViewOfService> factory =
        new ChannelFactory<IClientViewOfService>(
          new BasicHttpBinding(),
            new EndpointAddress(
              new Uri(address)));
      IClientViewOfService proxy =
        factory.CreateChannel();

      [...]

      try
      {
        Decimal quotient = proxy.DivideByZero(9);
      }
      catch (FaultException<SomeError> error)
      {
        Console.WriteLine("Error: {0}.", error.Detail.Content);
      }
```

LISTING 3.4 Continued

```
      [...]

    }
  }
}
```

Because receiving an error message from an attempt to use the operation should be anticipated, as the `FaultContract` for the operation indicates, the client code is written to handle that possibility. That is accomplished using the Windows Communication Foundation's `FaultException<T>` generic, which was also used in the code for the service to convey information about an exception to the client. The `Detail` property of the `FaultException<T>` generic provides access to an instance of `T`, which, in this case, is an instance of the `SomeError` class that the client can interrogate for information about the error that occurred.

This approach to handling exceptions provided by the Windows Communication Foundation has multiple virtues. It allows the developers of services to easily define the structure of the error messages that they want to transmit to client programmers. It also allows them to advertise to client programmers which operations of their services might return particular error messages instead of the results they would otherwise expect. The service programmers are able to easily formulate and transmit error messages to clients, and client programmers have a simple syntax, almost exactly like ordinary exception-handling syntax, for receiving and examining error messages. Most important, service programmers get to decide exactly what information about errors that occur in their services they want to have conveyed to clients.

However, the design of the Windows Communication Foundation does anticipate the utility, solely in the process of debugging a service, of being able to return to a client complete information about any unanticipated exceptions that might occur within a service. That can be accomplished using the `ReturnUnknownExceptionsAsFaults` behavior.

Behaviors are mechanisms internal to Windows Communication Foundation services or clients. They may be controlled by programmers or administrators.

Those behaviors that programmers are expected to want to control are manipulated using attributes. For example, if a programmer knows that a service is thread-safe, the programmer can permit the Windows Communication Foundation to allow multiple threads to access the service concurrently by adding the `ServiceBehavior` attribute to the service's service type class, and setting the value of the `ConcurrencyMode` parameter of that attribute to the `Multiple` value of the `ConcurrencyMode` enumeration:

```
[ServiceBehavior(ConcurrencyMode=ConcurrencyMode.Multiple)]
public class DerivativesCalculator : IServiceViewOfService
```

Behaviors that administrators are expected to want to control can be manipulated in the configuration of a service or client. The `ReturnUnknownExceptionsAsFaults` behavior is

one of those. This configuration of a service will result in any unhandled exceptions being transmitted to the client:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
    <system.serviceModel>
        <services>
          <service type=
"DerivativesCalculator.DerivativesCalculatorServiceType,
DerivativesCalculatorService"
            behaviorConfiguration="DerivativesCalculatorBehavior">
                <endpoint
                    address=""
                    binding="basicHttpBinding"
                    contract=
"DerivativesCalculator.IDerivativesCalculator,DerivativesCalculatorService"
                />
            </service>
        </services>
        <behaviors>
            <behavior name="DerivativesCalculatorBehavior"
             returnUnknownExceptionsAsFaults="true"/>
        </behaviors>
    </system.serviceModel>
</configuration>
```

To reiterate, this configuration may be very useful for diagnosis in the process of debugging a service, but it is dangerous in debugging, because transmitting all the information about an exception to a client may expose information about the service that could be used to compromise it.

Summary

Data being sent from a Windows Communication Foundation client to a service is serialized to XML within the client, and data received from clients by Windows Communication Foundation services is deserialized from XML within the service. There are two XML serializers that the Windows Communication Foundation can use to accomplish the serialization to XML and deserialization from XML. One is the `XmlSerializer` class that has been a part of the .NET Framework class library from the outset. The other is the `XmlFormatter` class that is new with the Windows Communication Foundation. Whereas the `XmlSerializer` provides precise control over how data is represented as XML, the `XmlFormatter` provides very little control over that in order to make the serialization process very predictable, and, thereby, easier to optimize. As a result, the `XmlFormatter` outperforms the `XmlSerializer`.

Allowing the `XmlFormatter` to serialize one's custom types is very simple. One merely adds a `DataContract` attribute to the definition of the type, and `DataMember` attributes to each of the type's members that are to be serialized.

Implementing the `IUnknownSerializationData` interface in any type that is to be serialized using the `XmlFormatter` is wise. It allows for different versions of the same way of representing the data in XML to evolve independently of one another, yet still be usable together.

References

Ballinger, Keith. 2003. *.NET Web Services: Architecture and Implementation*. Reading, MA: Addison-Wesley.

Freud, Sigmund. 1977. *Three Essays on Sexuality*. In *On Sexuality: Three Essays on Sexuality and Other Works*, ed. Angela Richards. The Pelican Freud Library, ed. James Strachey, no. 7. London, UK: Penguin.

This page intentionally left blank