This page is specific to

Microsoft Visual Studio 2010/.NET Framework 4

**Designing and Implementing Services**

This section shows you how to define and implement WCF contracts. A service contract specifies what an endpoint communicates to the outside world. At a more concrete level, it is a statement about a set of specific messages organized into basic message exchange patterns (MEPs), such as request/reply, one-way, and duplex. If a service contract is a logically related set of message exchanges, a service operation is a single message exchange. For example, a Hello operation must obviously accept one message (so the caller can announce the greeting) and may or may not return a message (depending upon the courtesy of the operation).

For more information about contracts and other core Windows Communication Foundation (WCF) concepts, see [Fundamental Windows Communication Foundation Concepts](http://msdn.microsoft.com/en-us/library/ms731079.aspx). This topic focuses on understanding service contracts. For more information about how to build clients that use service contracts to connect to services, see [WCF Client Overview](http://msdn.microsoft.com/en-us/library/ms735103.aspx).

**Overview**

This topic provides a high level conceptual orientation to designing and implementing WCF services. Subtopics provide more detailed information about the specifics of design and implementation. Before designing and implementing your WCF application, it is recommended that you:

* Understand what a service contract is, how it works, and how to create one.
* Understand that contracts state minimum requirements that runtime configuration or the hosting environment may not support.

**Service Contracts**

A service contract specifies the following:

* The operations a contract exposes.
* The signature of the operations in terms of messages exchanged.
* The data types of these messages.
* The location of the operations.
* The specific protocols and serialization formats that are used to support successful communication with the service.

For example, a purchase order contract might have a CreateOrder operation that accepts an input of order information types and returns success or failure information, including an order identifier. It might also have a GetOrderStatus operation that accepts an order identifier and returns order status information. A service contract of this sort would specify:

1. That the purchase order contract consisted of CreateOrder and GetOrderStatus operations.
2. That the operations have specified input messages and output messages.
3. The data that these messages can carry.
4. Categorical statements about the communication infrastructure necessary to successfully process the messages. For example, these details include whether and what forms of security are required to establish successful communication.

To convey this kind of information to other applications on many platforms (including non-Microsoft platforms), XML service contracts are publicly expressed in standard XML formats, such as [Web Services Description Language](http://go.microsoft.com/fwlink/?LinkId=94952) (WSDL) and [XML Schema](http://go.microsoft.com/fwlink/?LinkId=94953) (XSD), among others. Developers for many platforms can use this public contract information to create applications that can communicate with the service, both because they understand the language of the specification and because those languages are designed to enable interoperation by describing the public forms, formats, and protocols that the service supports. For more information about how WCF handles this kind of information, see [Metadata](http://msdn.microsoft.com/en-us/library/ms731823.aspx).

Contracts can be expressed many ways, and while WSDL and XSD are excellent languages to describe services in an accessible way, they are difficult languages to use directly and are merely descriptions of a service, not service contract implementations. Therefore, WCF applications use managed attributes, interfaces, and classes both to define the structure of a service and to implement it.

The resulting contract defined in managed types can be *exported* as metadata—WSDL and XSD—when needed by clients or other service implementers. The result is a straightforward programming model that can be described (using public metadata) to any client application. The details of the underlying SOAP messages, the transportation and security-related information, and so on, can be left to WCF, which performs the necessary conversions to and from the service contract type system to the XML type system automatically.

For more information about designing contracts, see [Designing Service Contracts](http://msdn.microsoft.com/en-us/library/ms733070.aspx). For more information about implementing contracts, see [Implementing Service Contracts](http://msdn.microsoft.com/en-us/library/ms733764.aspx).

**Messages Up Front and Center**

Using managed interfaces, classes, and methods to model service operations is straightforward when you are used to remote procedure call (RPC)-style method signatures, in which passing parameters into a method and receiving return values is the normal form of requesting functionality from an object or other type of code. For example, programmers using managed languages such as Visual Basic and C++ COM can apply their knowledge of the RPC-style approach (whether using objects or interfaces) to the creation of WCF service contracts without experiencing the problems inherent in RPC-style distributed object systems. Service orientation provides the benefits of loosely coupled, message-oriented programming while retaining the ease and familiarity of the RPC programming experience.

Many programmers are more comfortable with message-oriented application programming interfaces, such as message queues like Microsoft MSMQ, the [System.Messaging](http://msdn.microsoft.com/en-us/library/system.messaging.aspx) namespaces in the .NET Framework, or sending unstructured XML in HTTP requests, to name a few. For more information about programming at the message level, see [Using Message Contracts](http://msdn.microsoft.com/en-us/library/ms730255.aspx), [Service Channel-Level Programming](http://msdn.microsoft.com/en-us/library/ms789029.aspx), and [Interoperability with POX Applications](http://msdn.microsoft.com/en-us/library/aa738456.aspx).

**Understanding the Hierarchy of Requirements**

A service contract groups the operations; specifies the message exchange pattern, message types, and data types those messages carry; and indicates categories of run-time behavior an implementation must have to support the contract (for example, it may require that messages be encrypted and signed). The service contract itself does not specify precisely how these requirements are met, only that they must be. The type of encryption or the manner in which a message is signed is up to the implementation and configuration of a compliant service.

Notice the way that the contract requires certain things of the service contract implementation and the run-time configuration to add behavior. The set of requirements that must be met to expose a service for use builds on the preceding set of requirements. If a contract makes requirements of the implementation, an implementation can require yet more of the configuration and bindings that enable the service to run. Finally, the host application must also support any requirements that the service configuration and bindings add.

This additive requirement process is important to keep in mind while designing, implementing, configuring, and hosting a Windows Communication Foundation (WCF) service application. For example, the contract can specify that it needs to support a session. If so, then you must configure the binding to support that contractual requirement, or the service implementation will not work. Or if your service requires Windows Integrated Authentication and is hosted in Internet Information Services (IIS), the Web application in which the service resides must have Windows Integrated Authentication turned on and anonymous support turned off. For more information about the features and impact of the different service host application types, see [Hosting Services](http://msdn.microsoft.com/en-us/library/ms730158.aspx).

**Designing Service Contracts**

This topic describes what service contracts are, how they are defined, what operations are available (and the implications for the underlying message exchanges), what data types are used, and other issues that help you design operations that properly satisfy the requirements of your scenario.

## Creating a Service Contract

Service contracts define groups of operations. To create a service contract you must define the operations and specify their grouping. In Windows Communication Foundation (WCF) applications, you define an interface marked with the [ServiceContractAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.servicecontractattribute.aspx) attribute and define the operations by creating methods and marking them with the [OperationContractAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.operationcontractattribute.aspx) attribute. Alternatively you can define operations directly within a class by marking them with the **OperationContractAttribute** attribute. (For a basic example, see [How to: Define a Windows Communication Foundation Service Contract](http://msdn.microsoft.com/en-us/library/ms731835.aspx).)

Any methods that do not have a **OperationContractAttribute** attribute are not service operations and are not exposed for use by clients of WCF services. Like any managed method, they can only be called by objects within their declared access scope.

This topic describes the following decision points when designing a service contract:

* Whether to use classes or interfaces.
* How to specify the data types you want to exchange.
* The types of exchange patterns you can use.
* Whether you can make explicit security requirements part of the contract.
* The restrictions for operation inputs and outputs.

## Classes or Interfaces

Both classes and interfaces represent a grouping of functionality and, therefore, both can be used to define a WCF service contract. However, it is recommended that you use interfaces because they directly model service contracts. Without an implementation, interfaces do no more than define a grouping of methods with certain signatures. Likewise, a service contract without an implementation defines a grouping of operations with certain signatures. Implement a service contract interface and you have implemented a WCF service.

All the benefits of managed interfaces apply to service contract interfaces:

* Service contract interfaces can extend any number of other service contract interfaces.
* A single class can implement any number of service contracts by implementing those service contract interfaces.
* You can modify the implementation of a service contract by changing the interface implementation, while the service contract remains the same.
* You can version your service by implementing the old interface and the new one. Old clients connect to the original version, while newer clients can connect to the newer version.

|  |
| --- |
| **Note:** |
| When inheriting from other service contract interfaces, you cannot override operation properties, such as the name or namespace. If you attempt to do so, you create a new operation in the current service contract. |

For an example of using an interface to create a service contract, see [How to: Create a Service with a Contract Interface](http://msdn.microsoft.com/en-us/library/ms732002.aspx).

You can, however, use a class to define a service contract and implement that contract at the same time. The advantage of creating your services by applying **ServiceContractAttribute** and **OperationContractAttribute** directly to the class and the methods on the class, respectively, is speed and simplicity. The disadvantages are that managed classes do not support multiple inheritance, and as a result they can only implement one service contract at a time. In addition, any modification to the class or method signatures modifies the public contract for that service, which can prevent unmodified clients from using your service. For more information, see [Implementing Service Contracts](http://msdn.microsoft.com/en-us/library/ms733764.aspx).

For an example that uses a class to create a service contract and implements it at the same time, see [How to: Create a Windows Communication Foundation Contract with a Class](http://msdn.microsoft.com/en-us/library/ms730210.aspx).

At this point, you should understand the difference between defining your service contract by using an interface and by using a class. The next step is deciding what data can be passed back and forth between a service and its clients.

## Parameters and Return Values

Each operation has a return value and a parameter, even if these are **void**. However, unlike a local method, in which you can pass references to objects from one object to another, service operations do not pass references to objects. Instead, they pass copies of the objects.

This is significant because each type used in a parameter or return value must be serializable; that is, it must be possible to convert an object of that type into a stream of bytes and from a stream of bytes into an object.

Primitive types are serializable by default, as are many types in the .NET Framework.

|  |
| --- |
| **Description: noteNote:** |
| The value of the parameter names in the operation signature are part of the contract and are case sensitive. If you want to use the same parameter name locally but modify the name in the published metadata, see the [System.ServiceModel.MessageParameterAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.messageparameterattribute.aspx). |

#### Data Contracts

Service-oriented applications like Windows Communication Foundation (WCF) applications are designed to interoperate with the widest possible number of client applications on both Microsoft and non-Microsoft platforms. For the widest possible interoperability, it is recommended that you mark your types with the [DataContractAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datacontractattribute.aspx) and [DataMemberAttribute](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.datamemberattribute.aspx) attributes to create a data contract, which is the portion of the service contract that describes the data that your service operations exchange.

Data contracts are opt-in style contracts: No type or data member is serialized unless you explicitly apply the data contract attribute. Data contracts are unrelated to the access scope of the managed code: Private data members can be serialized and sent elsewhere to be accessed publicly. (For a basic example of a data contract, see [How to: Create a Basic Data Contract for a Class or Structure](http://msdn.microsoft.com/en-us/library/ms733811.aspx).) WCF handles the definition of the underlying SOAP messages that enable the operation's functionality as well as the serialization of your data types into and out of the body of the messages. As long as your data types are serializable, you do not need to think about the underlying message exchange infrastructure when designing your operations.

Although the typical WCF application uses the **DataContractAttribute** and **DataMemberAttribute** attributes to create data contracts for operations, you can use other serialization mechanisms. The standard [ISerializable](http://msdn.microsoft.com/en-us/library/system.runtime.serialization.iserializable.aspx), [SerializableAttribute](http://msdn.microsoft.com/en-us/library/system.serializableattribute.aspx) and [IXmlSerializable](http://msdn.microsoft.com/en-us/library/system.xml.serialization.ixmlserializable.aspx) mechanisms all work to handle the serialization of your data types into the underlying SOAP messages that carry them from one application to another. You can employ more serialization strategies if your data types require special support. For more information about the choices for serialization of data types in WCF applications, see [Specifying Data Transfer in Service Contracts](http://msdn.microsoft.com/en-us/library/ms732038.aspx).

It is important to note that the CLR names in the definition of a Service Contract and its operations are significant and should not be confused. To avoid confusion of types used to define a Service Contract use the [ObfuscationAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.obfuscationattribute.aspx) and [ObfuscateAssemblyAttribute](http://msdn.microsoft.com/en-us/library/system.reflection.obfuscateassemblyattribute.aspx) attributes.

#### Mapping Parameters and Return Values to Message Exchanges

Service operations are supported by an underlying exchange of SOAP messages that transfer application data back and forth, in addition to the data required by the application to support certain standard security, transaction, and session-related features. Because this is the case, the signature of a service operation dictates a certain underlying message exchange pattern (MEP) that can support the data transfer and the features an operation requires. You can specify three patterns in the WCF programming model: request/reply, one-way, and duplex message patterns.

##### Request/Reply

A request/reply pattern is one in which a request sender (a client application) receives a reply with which the request is correlated. This is the default MEP because it supports both an operation in which one or more parameters are passed to the operation and a return and one or more out values that the operation passes back to the caller. For example, the following C# code example shows a basic service operation that takes one string and returns a string.

C#

[OperationContractAttribute]

string Hello(string greeting);

The following is the equivalent Visual Basic code.

Visual Basic

<OperationContractAttribute()>

Function Hello (ByVal greeting As String) As String

This operation signature dictates the form of underlying message exchange. If no correlation existed, WCF cannot determine for which operation the return value is intended.

Note that unless you specify a different underlying message pattern, even service operations that return **void** (**Nothing** in Visual Basic) are request/reply message exchanges. The result for your operation is that unless a client invokes the operation asynchronously, the client stops processing until the return message is received, even though that message is empty in the normal case. The following C# code example shows an operation that does not return until the client has received an empty message in response.

C#

[OperationContractAttribute]

void Hello(string greeting);

The following is the equivalent Visual Basic code.

Visual Basic

<OperationContractAttribute()>

Sub Hello (ByVal greeting As String)

The preceding example can slow client performance and responsiveness if the operation takes a long time to perform, but there are advantages to request/reply operations even when they return **void**. The most obvious one is that SOAP faults can be returned in the response message, which indicates that some service-related error condition has occurred, whether in communication or processing. SOAP faults that are specified in a service contract are passed to the client application as a [FaultException](http://msdn.microsoft.com/en-us/library/ms576199.aspx) object, where the type parameter is the type specified in the service contract. This makes notifying clients about error conditions in WCF services easy. For more information about exceptions, SOAP faults, and error handling, see [Specifying and Handling Faults in Contracts and Services](http://msdn.microsoft.com/en-us/library/ms733721.aspx). To see an example of a request/reply service and client, see [How to: Create a Request-Reply Contract](http://msdn.microsoft.com/en-us/library/ms732381.aspx). For more information about issues with the request-reply pattern, see [Request-Reply Services](http://msdn.microsoft.com/en-us/library/ms730913.aspx).

##### One-way

If the client of a WCF service application should not wait for the operation to complete and does not process SOAP faults, the operation can specify a one-way message pattern. A one-way operation is one in which a client invokes an operation and continues processing after WCF writes the message to the network. Typically this means that unless the data being sent in the outbound message is extremely large the client continues running almost immediately (unless there is an error sending the data). This type of message exchange pattern supports event-like behavior from a client to a service application.

A message exchange in which one message is sent and none are received cannot support a service operation that specifies a return value other than **void**; in this case an [InvalidOperationException](http://msdn.microsoft.com/en-us/library/system.invalidoperationexception.aspx) exception is thrown.

No return message also means that there can be no SOAP fault returned to indicate any errors in processing or communication. (Communicating error information when operations are one-way operations requires a duplex message exchange pattern.)

To specify a one-way message exchange for an operation that returns **void**, set the [IsOneWay](http://msdn.microsoft.com/en-us/library/system.servicemodel.operationcontractattribute.isoneway.aspx) property to **true**, as in the following C# code example.

C#

[OperationContractAttribute(IsOneWay=true)]

void Hello(string greeting);

The following is the equivalent Visual Basic code.

Visual Basic

<OperationContractAttribute(IsOneWay := True)>

Sub Hello (ByVal greeting As String)

This method is identical to the preceding request/reply example, but setting the **IsOneWay** property to **true** means that although the method is identical, the service operation does not send a return message and clients return immediately once the outbound message has been handed to the channel layer. For an example, see [How to: Create a One-Way Contract](http://msdn.microsoft.com/en-us/library/ms733035.aspx). For more information about the one-way pattern, see [One-Way Services](http://msdn.microsoft.com/en-us/library/ms730149.aspx).

##### Duplex

A duplex pattern is characterized by the ability of both the service and the client to send messages to each other independently whether using one-way or request/reply messaging. This form of two-way communication is useful for services that must communicate directly to the client or for providing an asynchronous experience to either side of a message exchange, including event-like behavior.

The duplex pattern is slightly more complex than the request/reply or one-way patterns because of the additional mechanism for communicating with the client.

To design a duplex contract, you must also design a callback contract and assign the type of that callback contract to the [CallbackContract](http://msdn.microsoft.com/en-us/library/system.servicemodel.servicecontractattribute.callbackcontract.aspx) property of the **ServiceContractAttribute** attribute that marks your service contract.

To implement a duplex pattern, you must create a second interface that contains the method declarations that are called on the client.

For an example of creating a service, and a client that accesses that service, see [How to: Create a Duplex Contract](http://msdn.microsoft.com/en-us/library/ms731184.aspx) and [How to: Access Services with a Duplex Contract](http://msdn.microsoft.com/en-us/library/ms731935.aspx). For a working sample, see [Duplex](http://msdn.microsoft.com/en-us/library/ms752216.aspx). For more information about issues using duplex contracts, see [Duplex Services](http://msdn.microsoft.com/en-us/library/ms731064.aspx).

|  |
| --- |
| **Description: CautionCaution:** |
| When a service receives a duplex message, it looks at the **ReplyTo** element in that incoming message to determine where to send the reply. If the channel that is used to receive the message is not secured, then an untrusted client could send a malicious message with a target machine's **ReplyTo**, leading to a denial of service (DOS) of that target machine. |

##### Out and Ref Parameters

In most cases, you can use in parameters (**ByVal** in Visual Basic) and out and ref parameters (**ByRef** in Visual Basic). Because both out and ref parameters indicate that data is returned from an operation, an operation signature such as the following specifies that a request/reply operation is required even though the operation signature returns **void**.

C#

[ServiceContractAttribute]

public interface IMyContract

{

[OperationContractAttribute]

public void PopulateData(ref CustomDataType data);

}

The following is the equivalent Visual Basic code.

[Visual Basic]

<ServiceContractAttribute()> \_

Public Interface IMyContract

<OperationContractAttribute()> \_

Public Sub PopulateData(ByRef data As CustomDataType)

End Interface

The only exceptions are those cases in which your signature has a particular structure. For example, you can use the [NetMsmqBinding](http://msdn.microsoft.com/en-us/library/system.servicemodel.netmsmqbinding.aspx) binding to communicate with clients only if the method used to declare an operation returns **void**; there can be no output value, whether it is a return value, ref, or out parameter.

In addition, using out or ref parameters requires that the operation have an underlying response message to carry back the modified object. If your operation is a one-way operation, an **InvalidOperationException** exception is thrown at runtime.

### Specify Message Protection Level on the Contract

When designing your contract, you must also decide the message protection level of services that implement your contract. This is necessary only if message security is applied to the binding in the contract's endpoint. If the binding has security turned off (that is, if the system-provided binding sets the [System.ServiceModel.SecurityMode](http://msdn.microsoft.com/en-us/library/system.servicemodel.securitymode.aspx) to the value [System.ServiceModel.SecurityMode.None](http://msdn.microsoft.com/en-us/library/system.servicemodel.securitymode.aspx)) then you do not have to decide on the message protection level for the contract. In most cases, system-provided bindings with message-level security applied provide a sufficient protection level and you do not have to consider the protection level for each operation or for each message.

The protection level is a value that specifies whether the messages (or message parts) that support a service are signed, signed and encrypted, or sent without signatures or encryption. The protection level can be set at various scopes: At the service level, for a particular operation, for a message within that operation, or a message part. Values set at one scope become the default value for smaller scopes unless explicitly overridden. If a binding configuration cannot provide the required minimum protection level for the contract, an exception is thrown. And when no protection level values are explicitly set on the contract, the binding configuration controls the protection level for all messages if the binding has message security. This is the default behavior.

|  |
| --- |
| **Description: ImportantNote:** |
| Deciding whether to explicitly set various scopes of a contract to less than the full protection level of [System.Net.Security.ProtectionLevel.EncryptAndSign](http://msdn.microsoft.com/en-us/library/system.net.security.protectionlevel.aspx) is generally a decision that trades some degree of security for increased performance. In these cases, your decisions must revolve around your operations and the value of the data they exchange. For more information, see [Securing Services](http://msdn.microsoft.com/en-us/library/ms734769.aspx). |

For example, the following code example does not set either the [ProtectionLevel](http://msdn.microsoft.com/en-us/library/system.servicemodel.servicecontractattribute.protectionlevel.aspx) or the [ProtectionLevel](http://msdn.microsoft.com/en-us/library/system.servicemodel.operationcontractattribute.protectionlevel.aspx) property on the contract.

C#

[ServiceContract]

public interface ISampleService

{

[OperationContractAttribute]

public string GetString();

[OperationContractAttribute]

public int GetInt();

}

The following is the equivalent Visual Basic code.

[Visual Basic]

<ServiceContractAttribute()> \_

Public Interface ISampleService

<OperationContractAttribute()> \_

Public Function GetString()As String

<OperationContractAttribute()> \_

Public Function GetData() As Integer

End Interface

When interacting with an ISampleService implementation in an endpoint with a default [WSHttpBinding](http://msdn.microsoft.com/en-us/library/system.servicemodel.wshttpbinding.aspx) (the default **System.ServiceModel.SecurityMode**, which is [Message](http://msdn.microsoft.com/en-us/library/system.servicemodel.securitymode.aspx)), all messages are encrypted and signed because this is the default protection level. However, when an ISampleService service is used with a default [BasicHttpBinding](http://msdn.microsoft.com/en-us/library/system.servicemodel.basichttpbinding.aspx) (the default **SecurityMode**, which is **None**), all messages are sent as text because there is no security for this binding and so the protection level is ignored (that is, the messages are neither encrypted nor signed). If the **SecurityMode** was changed to **Message**, then these messages would be encrypted and signed (because that would now be the binding's default protection level).

If you want to explicitly specify or adjust the protection requirements for your contract, set the **ProtectionLevel** property (or any of the **ProtectionLevel** properties at a smaller scope) to the level your service contract requires. In this case, using an explicit setting requires the binding to support that setting at a minimum for the scope used. For example, the following code example specifies one **ProtectionLevel** value explicitly, for the GetGuid operation.

[C#]

[ServiceContract]

public interface IExplicitProtectionLevelSampleService

{

[OperationContractAttribute]

public string GetString();

[OperationContractAttribute(ProtectionLevel=ProtectionLevel.None)]

public int GetInt();

[OperationContractAttribute(ProtectionLevel=ProtectionLevel.EncryptAndSign)]

public int GetGuid();

}

The following is the equivalent Visual Basic code.

[Visual Basic]

<ServiceContract()> \_

Public Interface IExplicitProtectionLevelSampleService

<OperationContract()> \_

Public Function GetString() As String

End Function

<OperationContract(ProtectionLevel := ProtectionLevel.None)> \_

Public Function GetInt() As Integer

End Function

<OperationContractAttribute(ProtectionLevel := ProtectionLevel.EncryptAndSign)> \_

Public Function GetGuid() As Integer

End Function

End Interface

A service that implements this IExplicitProtectionLevelSampleService contract and has an endpoint that uses the default **WSHttpBinding** (the default **System.ServiceModel.SecurityMode**, which is **Message**) has the following behavior:

* The GetString operation messages are encrypted and signed.
* The GetInt operation messages are sent as unencrypted and unsigned (that is, plain) text.
* The GetGuid operation [System.Guid](http://msdn.microsoft.com/en-us/library/system.guid.aspx) is returned in a message that is encrypted and signed.

For more information about protection levels and how to use them, see [Understanding Protection Level](http://msdn.microsoft.com/en-us/library/aa347692.aspx). For more information about security, see [Securing Services](http://msdn.microsoft.com/en-us/library/ms734769.aspx).

##### Other Operation Signature Requirements

Some application features require a particular kind of operation signature. For example, the **NetMsmqBinding** binding supports durable services and clients, in which an application can restart in the middle of communication and pick up where it left off without missing any messages. (For more information, see [Queues in Windows Communication Foundation](http://msdn.microsoft.com/en-us/library/ms731089.aspx).) However, durable operations must take only one in parameter and have no return value.

Another example is the use of [Stream](http://msdn.microsoft.com/en-us/library/system.io.stream.aspx) types in operations. Because the **Stream** parameter includes the entire message body, if an input or an output (that is, ref parameter, out parameter, or return value) is of type **Stream**, then it must be the only input or output specified in your operation. In addition, the parameter or return type must be either **Stream**, [System.ServiceModel.Channels.Message](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.message.aspx), or **System.Xml.Serialization.IXmlSerializable**. For more information about streams, see [Large Data and Streaming](http://msdn.microsoft.com/en-us/library/ms733742.aspx).

##### Names, Namespaces, and Obfuscation

The names and namespaces of the .NET types in the definition of contracts and operations are significant when contracts are converted into WSDL and when contract messages are created and sent. Therefore, it is strongly recommended that service contract names and namespaces are explicitly set using the **Name** and **Namespace** properties of all supporting contract attributes such as the **ServiceContractAttribute**, **OperationContractAttribute**, **DataContractAttribute**, **DataMemberAttribute**, and other contract attributes.

One result of this is that if the names and namespaces are not explicitly set, the use of IL obfuscation on the assembly alters the contract type names and namespaces and results in modified WSDL and wire exchanges that typically fail. If you do not set the contract names and namespaces explicitly but do intend to use obfuscation, use the **ObfuscationAttribute** and **ObfuscateAssemblyAttribute** attributes to prevent the modification of the contract type names and namespaces.

**Implementing Service Contracts**

A service is a class that exposes functionality available to clients at one or more endpoints. To create a service, write a class that implements a Windows Communication Foundation (WCF) contract. You can do this in one of two ways. You can define the contract separately as an interface and then create a class that implements that interface. Alternatively, you can create the class and contract directly by placing the [ServiceContractAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.servicecontractattribute.aspx) attribute on the class itself and the [OperationContractAttribute](http://msdn.microsoft.com/en-us/library/system.servicemodel.operationcontractattribute.aspx) attribute on the methods available to the clients of the service.

## Creating a service class

The following is an example of a service that implements an **IMath** contract that has been defined separately.

C#

// Define the IMath contract.

[ServiceContract]

public interface IMath

{

[OperationContract]

double Add(double A, double B);

[OperationContract]

double Multiply (double A, double B);

}

// Implement the IMath contract in the MathService class.

public class MathService : IMath

{

public double Add (double A, double B) { return A + B; }

public double Multiply (double A, double B) { return A \* B; }

}

Alternatively, a service can expose a contract directly. The following is an example of a service class that defines and implements a **MathService** contract.

C#

// Define the MathService contract directly on the service class.

[ServiceContract]  
class MathService

{

[OperationContract]  
 public double Add(double A, double B) { return A + B; }

[OperationContract]

private double Multiply (double A, double B) { return A \* B; }

}

Note that the preceding services expose different contracts because the contract names are different. In the first case, the exposed contract is named "**IMath**" while in the second case the contract is named "**MathService**".

You can set a few things at the service and operation implementation levels, such as concurrency and instancing. For more information, see [Designing and Implementing Services](http://msdn.microsoft.com/en-us/library/ms729746.aspx).

After implementing a service contract, you must create one or more endpoints for the service. For more information, see [Endpoint Creation Overview](http://msdn.microsoft.com/en-us/library/ms735096.aspx). For more information about how to run a service, see [Hosting Services](http://msdn.microsoft.com/en-us/library/ms730158.aspx).