Serialization

Moving between the worlds of .NET and XML

Overview

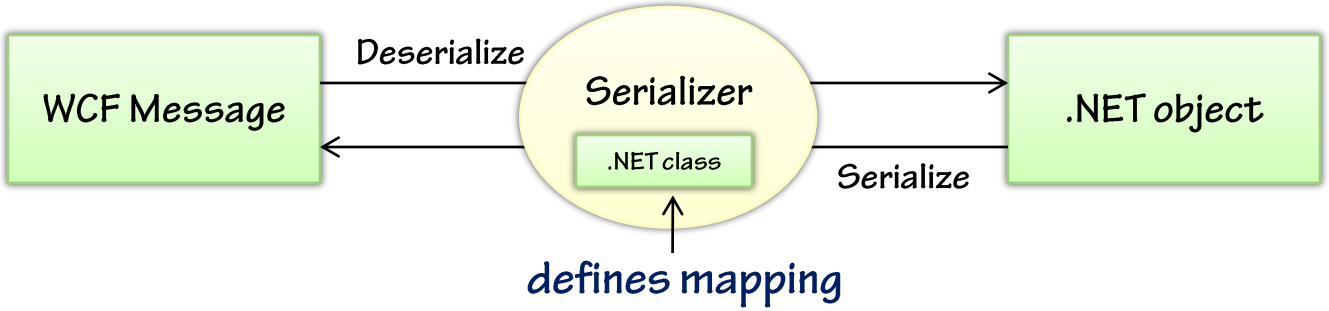
- **WCF serializers**
  - XmlSerializer
  - NetDataContractSerializer
  - DataContractSerializer
- **DataContractSerializer**
  - Mapping details
- **Advanced serialization topics**
  - Known types
  - Collections and generics
  - Serialization events

WCF message processing

- **WCF provides a message-oriented programming framework**
  - All messages modeled by the WCF Message class
  - Moving between raw streams and Message objects is called encoding
- **WCF makes it easy to avoid the Message object altogether**
  - Transforming Message objects into .NET objects is called serialization
  - This process is performed by a serializer at runtime

WCF serializers

- **WCF supports numerous serializers, each with its pros and cons**
  - XmlSerializer (available since .NET 1.0)
  - NetDataContractSerializer
  - DataContractSerializer (the default)
  - DataContractJsonSerializer (new in .NET 3.5)



Specifying a serializer

- **WCF uses DataContractSerializer by default with service contracts**
  - You can tell it to use a specific serializer using special attributes
  - You can apply the attribute to the contract or per method
  - Only XmlSerializer/DataContractSerializer can be specified this way
  - NetDataContractSerializer requires a custom behavior attribute

tells WCF to use XmlSerializer for all operations →

```
[XmlSerializerFormat]
[ServiceContract]
public interface ICustomerService
{
    [OperationContract]
    void AddCustomer(Customer customer);
}
```

Importing/exporting serializer types

- **Each serializer supports a different mapping algorithm**
  - They look for their own attributes that influence mapping
  - Each serializer will produce a slightly different message
- **You can use SvcUtil.exe to move between .NET types and XSD**
  - You can export an XSD that describes what the serializer expects
  - You can import an XSD to produce types for a specific serializer

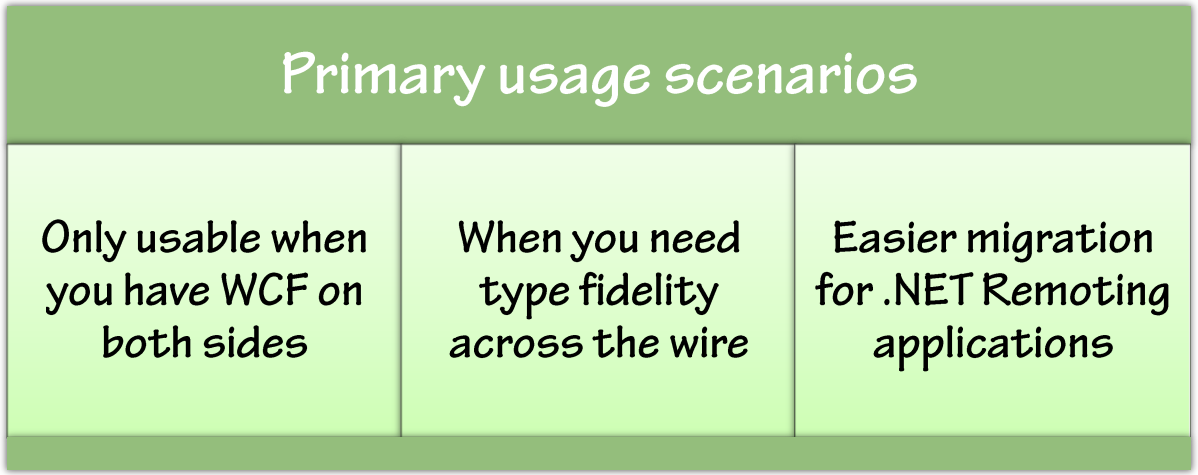
XmlSerializer

- **WCF continues to support XmlSerializer from .NET 1.0**
  - Found in System.Xml.Serialization
  - Use by default with ASP.NET Web services (ASMX)
  - Supports nearly the full-range of XSD constructs
  - Generates read/write code, hence public visibility required

Primary usage scenarios		
Backwards compatibility with ASMX	When integrating with non-WCF services	When using contract-first design techniques

NetDataContractSerializer

- **NetDataContractSerializer is analogous to .NET Remoting formatters**
  - It implements IFormatter and it's compatible with [Serializable] types
  - Serializes .NET type information into the message
  - You get the same type (same version even) when deserializing
  - Uses reflection during serialization, hence privates are serializable



Not recommended for service-oriented designs

DataContractSerializer

- **DataContractSerializer is the new default serializer for WCF**
  - Does not serialize type information into the message
  - But doesn't support the full-range of XSD constructs either
  - Constrains message types to an "interop-safe" subset of XSD
  - Uses reflection to serialize, hence privates are serializable

Primary usage scenarios	
Use by default unless you have a good reason not to	Typically the best choice for code-first designs

Supported types

- **What can DataContractSerializer serialize? (in order of precedence)**
  - CLR primitive types (int, double, string, etc)
  - Byte array, DateTime, TimeSpan, GUID, Uri
  - XmlQualifiedName, XmlElement and XmlNode array
  - Enums (without any annotations)
  - Types marked with [DataContract] or [CollectionDataContract]
  - Types that implement IXmlSerializable
  - Arrays/collection classes including generics
  - Types marked with [Serializable] or implementing ISerializable

If you need to serialize a type that isn't supported,
use a data contract surrogate (advanced)

[DataContract] basics

- **DataContractSerializer looks for [DataContract] attributes**
  - Mapping attributes found in System.Runtime.Serialization
- **Explicit opt-in model**
  - The type must be annotated with [DataContract]
  - Only members annotated with [DataMember] are included

makes this serializable

included in message

```
[DataContract]
public class Invoice {
    [DataMember]
    public string CustomerId;
    [DataMember]
    public DateTime InvoiceDate;
    public string SomePrivateData;
    ...
}
```

not included in message

Default mapping

- **Default mapping for [DataContract] and [Serializable] types**
  - Class name becomes element (& complex type) name
  - Field/property names become local element names
  - Local elements are mapped in alphabetical order
  - XML namespace derived from the type's .NET namespace

```
[DataContract]
public class Invoice {
    [DataMember]
    public string InvoiceId;
    [DataMember]
    public string CustomerName;
    ...
}
```

default mapping

```
<Invoice xmlns="..." >
   <CustomerName>Acme, Inc</CustomerName>
   <InvoiceId>1234-5678</InvoiceId>
   ...
</Invoice>
```

Customizing mapping details

[DataContract]	• Name • Namespace	Controls the XSD type/root element
[DataMember]	• Name • IsRequired • Order • EmitDefaultValue	Controls the local element definition

Mapping properties vs. fields

- **With [Serializable] types, only fields are included in the message**
  - Properties are always ignored
- **With [DataContract], you can map fields or properties**
  - Simply annotate property with [DataMember]
  - Property must have set & get methods (used during serialization)

include property in message

```
[DataContract]
public class Invoice
{
    private string customerId;
    [DataMember]
    public string CustomerId
    {
        get { return customerId; }
        set { CustomerId = value; }
    }
    ...
```

don't include underlying field (or it will show up twice)

Public vs. private

- **Unlike XmlSerializer, DCS uses reflection to implement serialization**
  - This gives it the ability to serialize private/protected/internal members
  - Only members marked with [DataMember] are included

include private field →

```
[DataContract]
public class Invoice
{
    [DataMember]
    private string customerId;
    public string CustomerId
    {
        get { return customerId; }
        set { CustomerId = value; }
    }
    ...
```

Optional vs. nillable

- **Both reference and value types show up as optional in the schema**
  - Set IsRequired=true to make members required
- **Reference types show up in the schema as nillable by default**
  - While value types are not nillable by default
  - You can make value types nillable by using Nullable<T>
- **During serialization, default values are emitted by default**
  - Set EmitDefaultValue=false to drop the element from the message
  - During deserialization, default values are used for missing elements

Various aspects of this behavior are different from XmlSerializer

Optional vs. nillable

```
[DataContract]
public class Invoice
{
    [DataMember]
    public string InvoiceId;
    [DataMember]
    public string CustomerName;
    [DataMember]
    public DateTime InvoiceDate;
    [DataMember]
    public int Age;
    ...
}
```

```
Invoice invoice = new Invoice();
... // serialize invoice here
```

EmitDefaultValue=true (default)

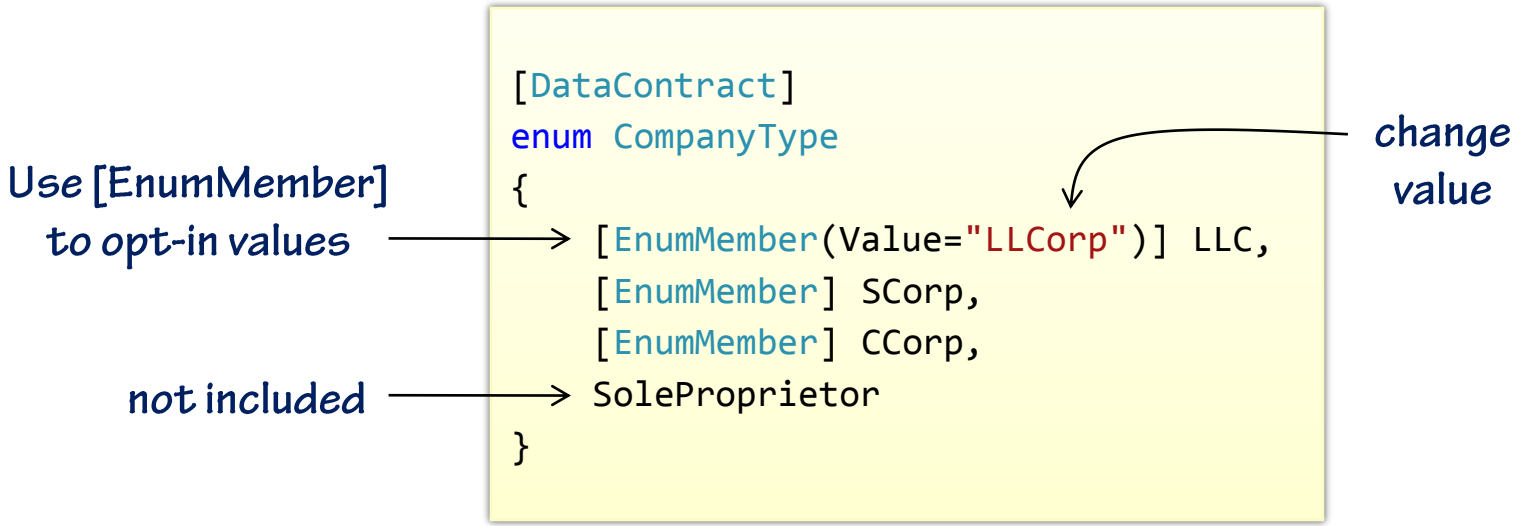
```
<Invoice xmlns="..." xmlns:i="...">
  <Age>0</Age>
  <CustomerName i:nil="true"/>
  <InvoiceDate>0001-01-01T00:00:00</InvoiceDate>
  <InvoiceId i:nil="true"/>
</Invoice>
```

EmitDefaultValue=false

```
<Invoice xmlns="..." xmlns:i="..."/>
```

Enumerations

- **Enums are serializable by default without any annotations**
  - All enum values are included in the schema type
- **You can exclude specific values or change their names if desired**
  - Annotate enum with [DataContract]
  - Annotate each value you wish to include with [EnumMember]



Composing data contracts

- **[DataContract] types may be composed from other serializable types**
  - All types used must be marked with [DataContract] or [Serializable]

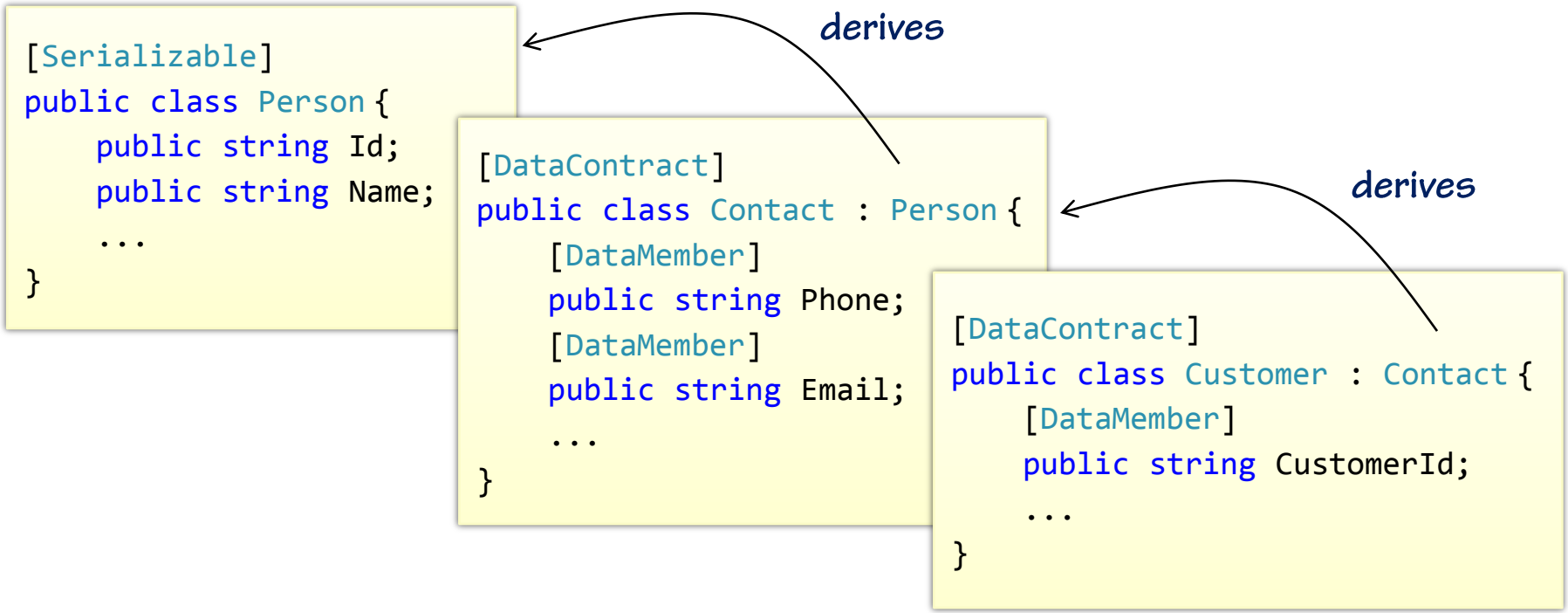
```
[Serializable]
public class Address
{
    public string Street;
    public string City;
    public string State;
    public string Zip;
}
```

```
[DataContract]
enum CompanyType
{
    [EnumMember] LLC,
    [EnumMember] SCorp,
    [EnumMember] CCorp,
}
```

```
[DataContract]
public class Company
{
    [DataMember] public string Name;
    [DataMember] public CompanyType Type;
    [DataMember] public Address Address;
}
```

Serializable class hierarchies

- **If a [DataContract] has a base class, the base must also be serializable**
  - All types within the inheritance hierarchy must be serializable
  - They must be marked with [Serializable] or [DataContract]



Known types

- **In C#, you can substitute a derived type for a base type at runtime**
- **With WCF services, it's not quite that easy**
  - WCF must "know" about all possible substitutions ahead of time
  - You specify possible substitutions using [KnownType]

```
[KnownType(typeof(Contact))]
[KnownType(typeof(Customer))]
[DataContract]
public class Person {
    [DataContract]
    public string Id;
    ...
}
```

```
[KnownType("GetMyKnownTypes")]
[Serializable]
public class Person {
    ...
    static IEnumerable<Type> GetMyKnownTypes() {
        List<Type> knownTypes = new List<Type>();
        knownTypes.Add(typeof(Contact));
        knownTypes.Add(typeof(Customer));
        return knownTypes;
    }
}
```

Service known types

- **You can also annotate the service contract with known types**
  - Use [ServiceKnownType] on the contract or on specific operations
  - Useful when you can't annotate the base type
  - Only makes substitutions possible on the annotated contract

```
[ServiceContract]
public interface IContactService {
    [ServiceKnownType(typeof(Contact))]
    [ServiceKnownType(typeof(Customer))]
    [OperationContract]
    void AddContact(Person p);
    ...
}
```

annotate specific operations

annotate the entire contract

```
[ServiceKnownType(typeof(Contact))]
[ServiceKnownType(typeof(Customer))]
[ServiceContract]
public interface IContactService {
    [OperationContract]
    void AddContact(Person p);
    ...
}
```

Configuring known types

- **You can also specify known types in the application configuration file**
  - Within <system.runtime.serialization> under <dataContractSerializer>
  - Add each <knownType> within the <declaredTypes> element

known types for Person →

```
<configuration>
  <system.runtime.serialization>
    <dataContractSerializer>
      <declaredTypes>
        <add type="Person,MyAssembly,Version=1.0.0.0,...">
          <knownType type="Contact,MyAssembly,Version=1.0.0.0,..."/>
          <knownType type="Customer,MyAssembly,Version=1.0.0.0,..."/>
        </add>
      </declaredTypes>
    </dataContractSerializer>
  </system.runtime.serialization>
</configuration>
```

allows you to add/modify/remove known types after deployment

DataSets

- **DataSets are statistically one of the most commonly used data types**
  - The DataSet is simply a generic table structure for holding rows of data
- **Using DataSets in your data contracts can be problematic**
  - The XSD will end up with a wildcard (meaning anything can go there)
  - The other side will have to use an XML API to process
- **However, DataSets can be deserialized on the other side if it's WCF**

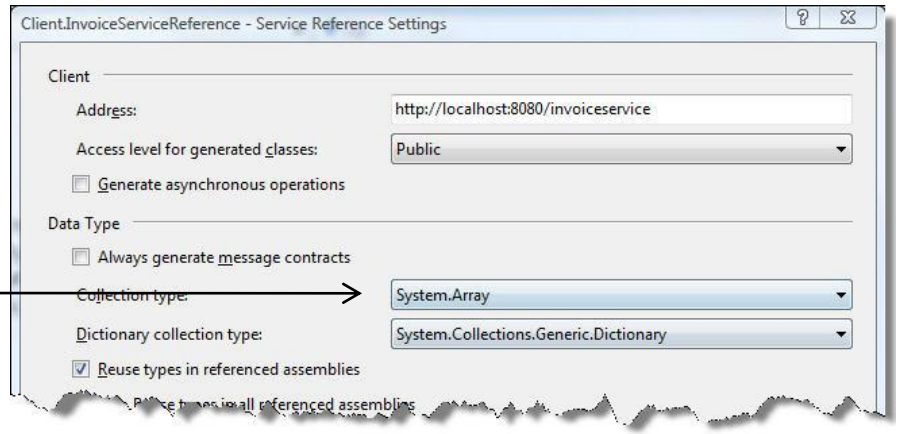
```
<Employees>
  <xs:schema id="NewDataSet" xmlns:xs="..." xmlns:msdata="... ">
    <xs:element name="NewDataSet" msdata:IsDataSet="true" ...>
    ...
  </xs:schema>
  <diffgr:diffgram xmlns:diffgr="..." xmlns:msdata="..."/>
</Employees>
```

here's the magic

Arrays and collections

- **Instead of DataSet, you're usually better off using a typed array**
  - Consumers can easily map to array types in their environment
- **Both arrays & collections map to the same XSD format (sequence)**
  - Array, ArrayList, and collections (IEnumerable, IList, ICollection)
  - Generic collections (IEnumerable<T>, IList<T>, ICollection<T>)
  - Iterators (.NET 3.5 only)
- **When generating code from XSD, you can choose the array type**

Specify what type to use for sequences



You can also use /collectionType (SvcUtil.exe)

Custom collections

- **You can use custom collection types in your data contracts**
  - Annotate the class with [CollectionDataContract]

```
[CollectionDataContract(Name="employees", ItemName="employee")]
public class EmployeeCollection : Collection<string>
{ ... }
```

control container and item names

provide an Add method

```
[CollectionDataContract(Name="MyFancyCollectionOf{0}")]
public class MyFancyCollection<T>: IEnumerable<T> {
    public void Add(T item) { ... }
    public IEnumerator<T> GetEnumerator() { ... }
    ...
}
```

Serializer events

- **Constructors are not called during deserialization**
  - However, property set/get methods are called
- **Callback methods are provided at each serialization stage**
  - Mark the callback method with serialization stage attribute
  - Method must accept a single argument of type StreamingContext

```
[DataContract]
public class Company {
    [OnDeserializing] void OnDeserializing(StreamingContext ctx) { ... }
    [OnDeserialized] void OnDeserialized(StreamingContext ctx) { ... }
    [OnSerializing] void OnSerializing(StreamingContext ctx) { ... }
    [OnSerialized] void OnSerialized(StreamingContext ctx) { ... }
    ...
}
```

serialization stage attributes

Summary

- **WCF provides a message-oriented programming model**
  - But messages can be mapped to .NET objects for ease-of-use
- **WCF provides numerous serializers**
  - DataContractSerializer is the default serializer for WCF
  - NetDataContractSerializer is provided for when need type fidelity
  - XmlSerializer is support for backwards compatibility with ASMX
- **DataContractSerializer supports advanced serialization features**
  - Surrogates, substitution, arrays/collections, generics, etc
- **Use SvcUtil.exe to map between .NET and XSD types**

References

- **Serialization in Windows Communication Foundation, Skonnard**
  - http://msdn.microsoft.com/msdnmag/issues/06/08/ServiceStation/
- **WCF Messaging Fundamentals, Skonnard**
  - http://msdn.microsoft.com/msdnmag/issues/07/04/ServiceStation/
- **Pluralsight's WCF Wiki**
  - http://pluralsight.com/wiki/default.aspx/Aaron/WindowsCommunicationFoundationWiki.html