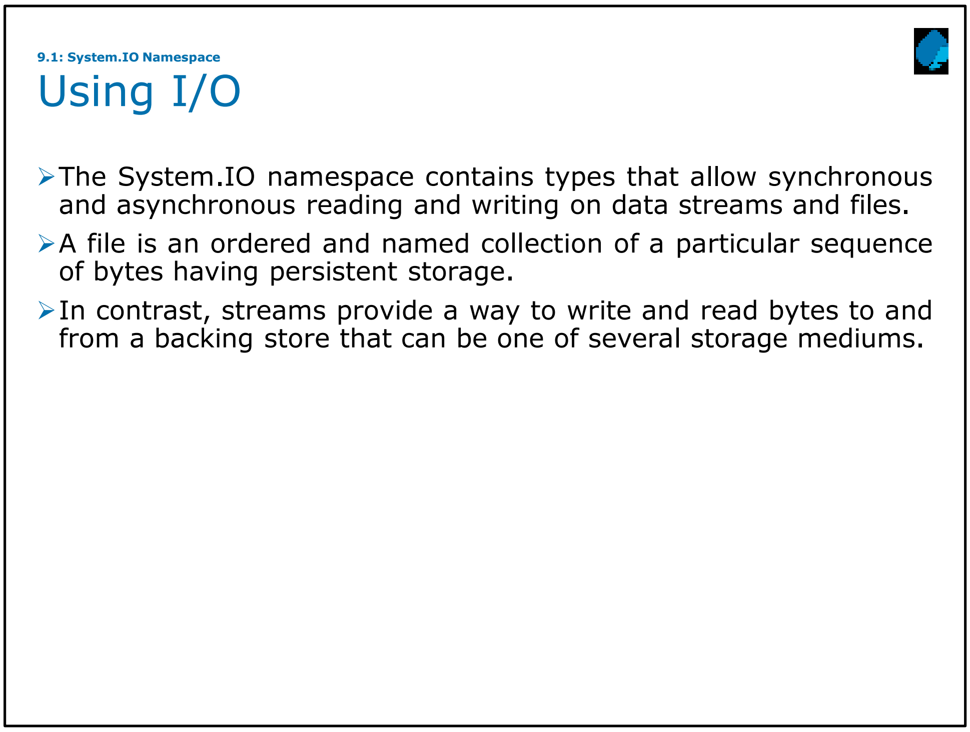
.NET Framework 4.6 and C# 7.0

Lesson 11 : File IO and Serialization

Lesson Objectives

- In this lesson, we will learn about:
  - I/O operations in C#
  - Concept of Serialization
  - The need for Serialization
  - Different ways of Serialization
    - Binary
    - Soap
    - XML
  - JSON Serialization using DataContractJsonSerializer
  - Runtime Serialization



9.1: System.IO Namespace

Using I/O

- The System.IO namespace contains types that allow synchronous and asynchronous reading and writing on data streams and files.
- A file is an ordered and named collection of a particular sequence of bytes having persistent storage.
- In contrast, streams provide a way to write and read bytes to and from a backing store that can be one of several storage mediums.

System.IO Namespace:

The System.IO namespace contains types that allow synchronous and asynchronous reading and writing on data streams and files.
The following distinctions help clarify the differences between a file and a stream.
A file is an ordered and named collection of a particular sequence of bytes having persistent storage. Therefore, with files, one thinks in terms of directory paths, disk storage, and file and directory names.
In contrast, streams provide a way to write and read bytes to and from a backing store that can be one of several storage mediums. Just as there are several backing stores other than disks, there are several kinds of streams other than file streams. For example, there are network, memory, and tape streams.

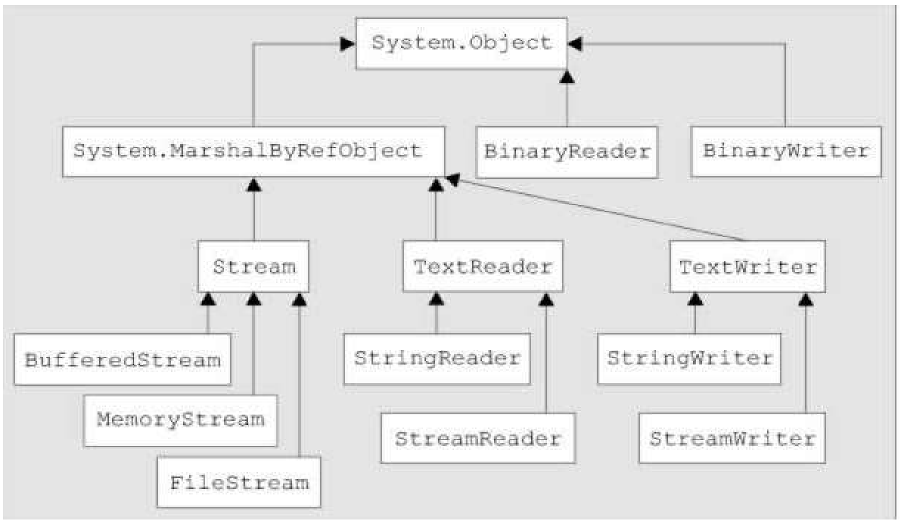
File :A file is an ordered and named collection of a particular sequence of bytes having persistent storage. Therefore, with files, one thinks in terms of directory paths, disk storage, and file and directory names.

Stream :Streams provide a way to write and read bytes to and from a backing store that can be one of several storage mediums. Just as there are several backing stores other than disks, there are several kinds of streams other than file streams. For example, there are network, memory, and tape streams.

All classes that represent streams inherit from the **Stream** class.
The Stream class and its derived classes provide a generic view of data sources and repositories, isolating the programmer from the specific details of the operating system and underlying devices.

9.1: System.IO Namespace

Exploring System.IO Namespace

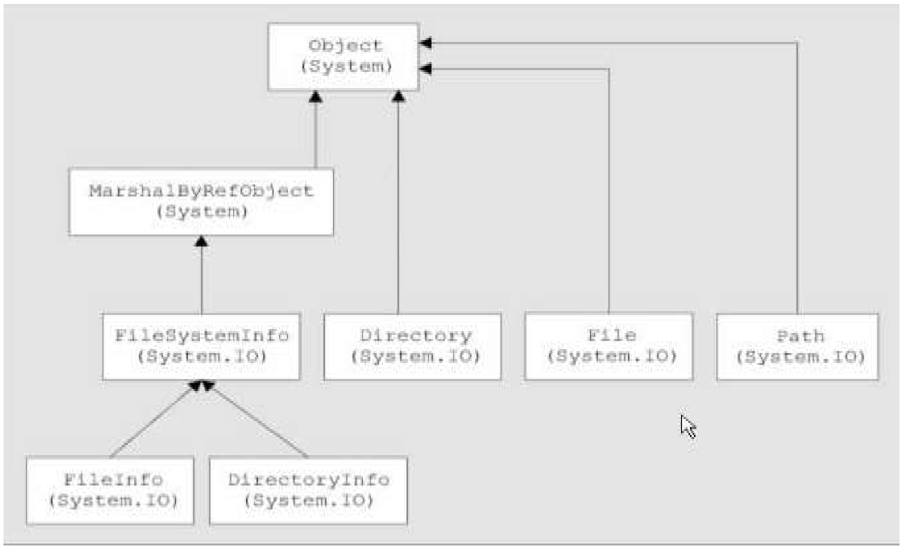


Exploring the System.IO NameSpace:

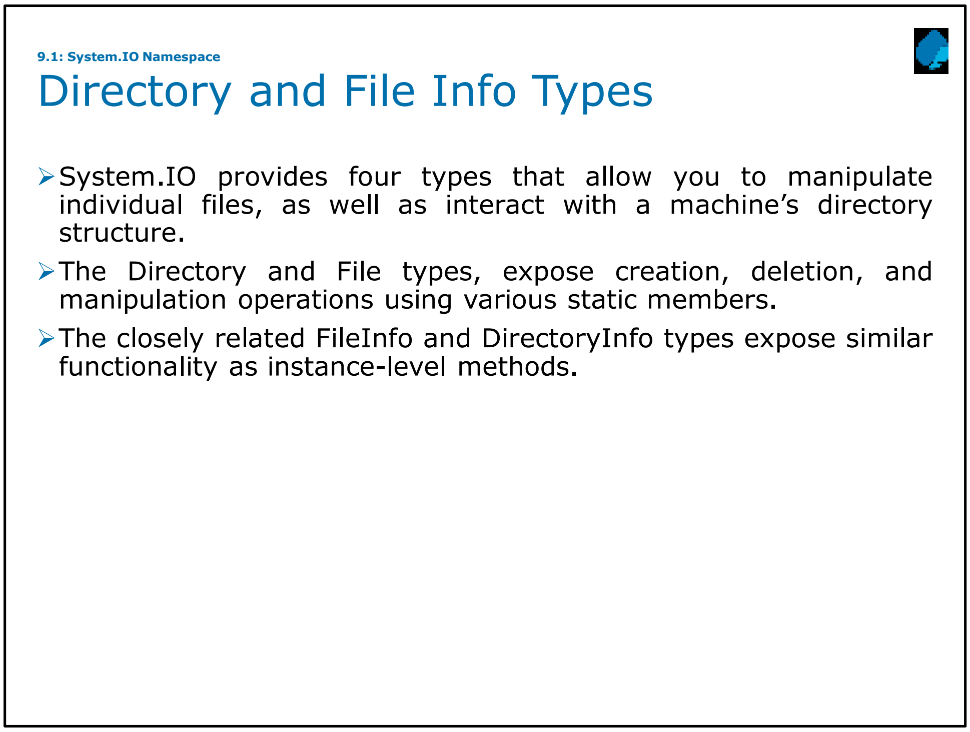
The System.IO namespace is the region of the base class libraries devoted to file-based (and memory-based) input and output services. Following Table shows core types of System.IO namespace.

Class	Description
BinaryReader:	Reads primitive data types as binary values in a specific encoding.
BinaryWriter	Writes primitive types in binary to a stream and supports writing strings in a specific encoding.
BufferedStream	Buffers reads and writes to another stream. This class cannot be inherited.
Directory	Exposes static methods for creating, moving, and enumerating through directories and subdirectories.
DirectoryInfo	Exposes instance methods for creating, moving, and enumerating through directories and subdirectories.

Exploring System.IO Namespace



The FileInfo and DirectoryInfo types expose similar functionality provided by File and Directory classes as instance-level methods.



9.1: System.IO Namespace

Directory and File Info Types

- System.IO provides four types that allow you to manipulate individual files, as well as interact with a machine’s directory structure.
- The Directory and File types, expose creation, deletion, and manipulation operations using various static members.
- The closely related FileInfo and DirectoryInfo types expose similar functionality as instance-level methods.

The Directory (Info) and File (Info) Types:

System.IO provides four types that allow you to manipulate individual files, as well as interact with a machine's directory structure. The first two types Directory and File, expose creation, deletion, and manipulation operations using various static members. The closely related FileInfo and DirectoryInfo types expose similar functionality as instance-level methods.

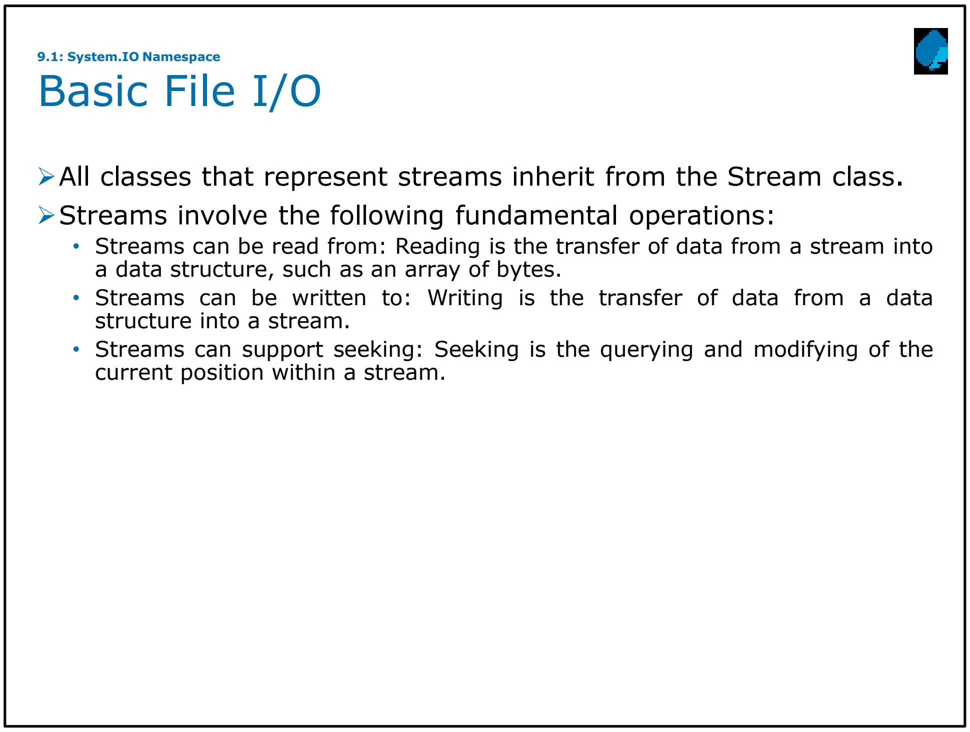
Basic File IO:

The abstract base class Stream supports reading and writing bytes. Stream integrates asynchronous support. Its default implementations define synchronous reads and writes in terms of their corresponding asynchronous methods, and vice versa.
All classes that represent streams inherit from the Stream class. The Stream class and its derived classes provide a generic view of data sources and repositories, isolating the programmer from the specific details of the operating system and underlying devices.
Streams involve these fundamental operations:
Streams can be read from. Reading is the transfer of data from a stream into a data structure, such as an array of bytes.
Streams can be written to. Writing is the transfer of data from a data structure into a stream.
Streams can support seeking. Seeking is the querying and modifying of the current position within a stream.

- Hidden Slide

I/O Classes Derived from System.Object:

BinaryReader and BinaryWriter read and write encoded strings and primitive data types from and to Streams.
File provides static methods for the creation, copying, deletion, moving, and opening of files, and aids in the creation of FileStream objects. The FileInfo class provides instance methods.
Directory provides static methods for creating, moving, and enumerating through directories and subdirectories. The DirectoryInfo class provides instance methods.
Path provides methods and properties for processing directory strings in a cross-platform manner.
File, Path, and Directory are sealed (in Microsoft Visual Basic, NotInheritable) classes. You can create new instances of these classes, but they can have no derived classes.



9.1: System.IO Namespace

Basic File I/O

- All classes that represent streams inherit from the Stream class.
- Streams involve the following fundamental operations:
  - Streams can be read from: Reading is the transfer of data from a stream into a data structure, such as an array of bytes.
  - Streams can be written to: Writing is the transfer of data from a data structure into a stream.
  - Streams can support seeking: Seeking is the querying and modifying of the current position within a stream.

Depending on the underlying data source or repository, streams might support only some of these capabilities.
For example, NetworkStreams do not support seeking. The CanRead, CanWrite, and CanSeek properties of Stream and its derived classes determine the operations that various streams support.

Classes Derived from System.IO.Stream:
FileStream supports random access to files through its Seek method. FileStream opens files synchronously by default, but supports asynchronous operation as well. File contains static methods, and FileInfo contains instance methods.
A BufferedStream is a Stream that adds buffering to another Stream such as a NetworkStream. (FileStream already has buffering internally, and a MemoryStream does not need buffering). A BufferedStream object can be composed around some types of streams in order to improve read and write performance. A buffer is a block of bytes in memory used to cache data, thereby reducing the number of calls to the operating system.
System.IO.TextReader and Its Derived Classes
TextReader is the abstract base class for StreamReader and StringReader objects. While the implementations of the abstract Stream class are designed for byte input and output, the implementations of TextReader are designed for Unicode character output.
StreamReader reads characters from Streams, using Encoding to convert characters to and from bytes. StreamReader has a constructor that attempts to ascertain what the correct Encoding for a given Stream is, based on the presence of an Encoding-specific preamble, such as a byte order mark.

Cont..

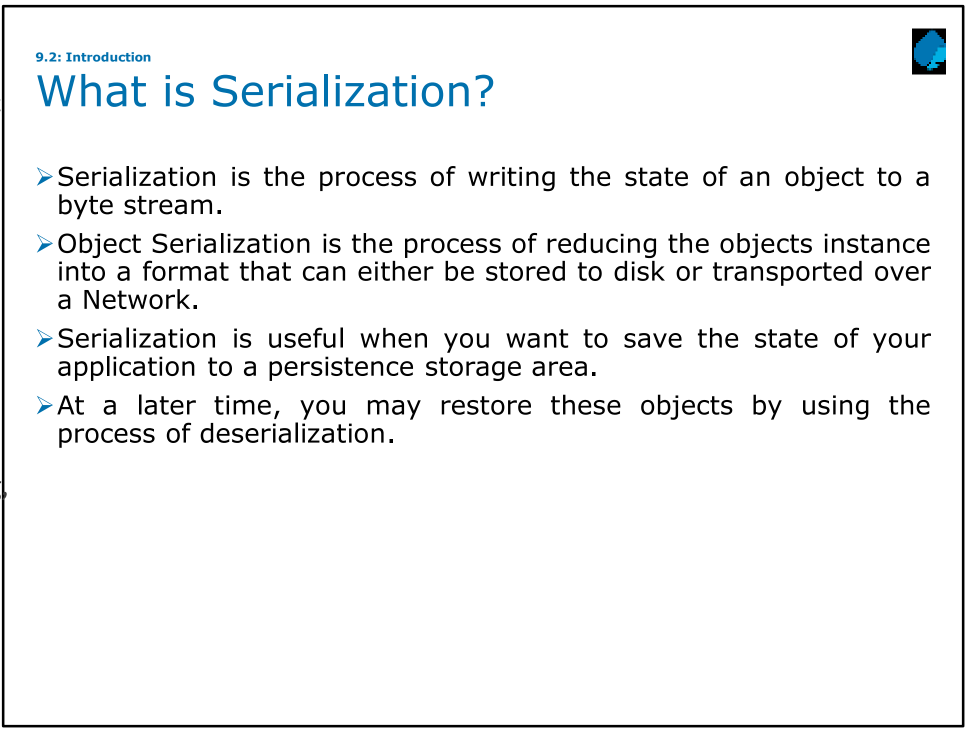
System.IO.TextReader and Its Derived Classes (Cont..)
StringReader reads characters from Strings. StringReader allows you to treat Strings with the same API, so your output can be either a Stream in any encoding or a String.
System.IO.TextWriter and Its Derived Classes:
TextWriter is the abstract base class for StreamWriter and StringWriter objects. While the implementations of the abstract Stream class are designed for byte input and output, the implementations of TextWriter are designed for Unicode character input.
StreamWriter writes characters to Streams, using Encoding to convert characters to bytes.
StringWriter writes characters to Strings. StringWriter allows you to treat Strings with the same API, so your output can be either a Stream in any encoding or a String.

Demo

- FileStream Class
- Reader and Writer Classes

Add the notes here.

The basic idea of serialization is that an object writes its current state, usually indicated by the value of its member variables, to Temporary (either memory or network streams) or persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from the storage. Serialization handles all the details of object pointers and circular object references that are used when you serialize an object.



9.2: Introduction

What is Serialization?

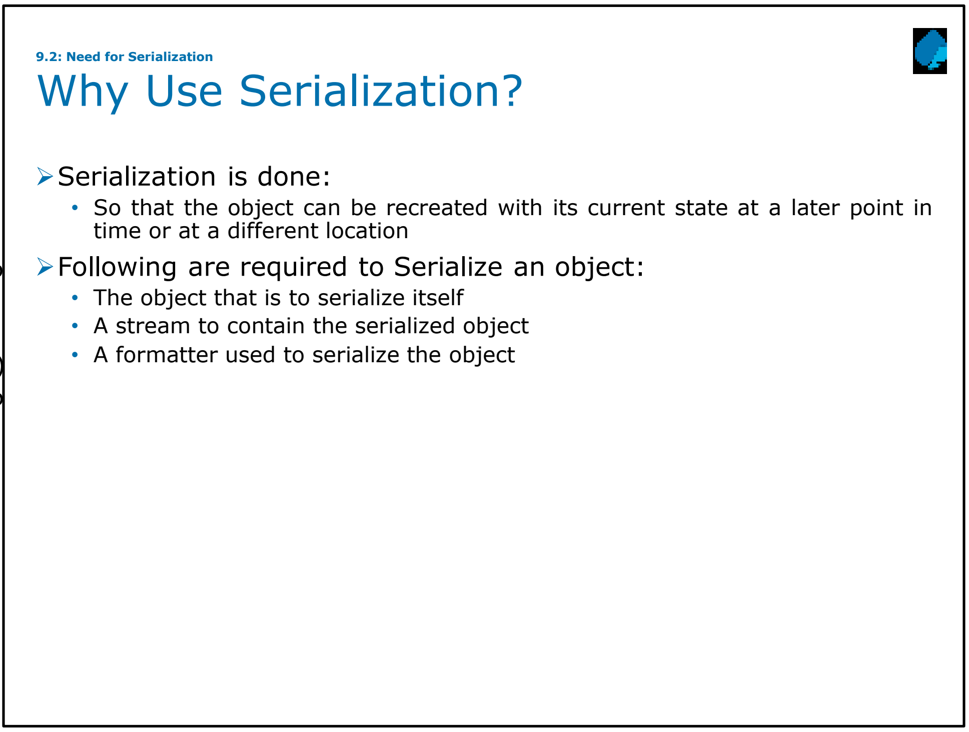
- Serialization is the process of writing the state of an object to a byte stream.
- Object Serialization is the process of reducing the objects instance into a format that can either be stored to disk or transported over a Network.
- Serialization is useful when you want to save the state of your application to a persistence storage area.
- At a later time, you may restore these objects by using the process of deserialization.

What is Serialization?

Serialization is the process of taking objects and converting their state information into a form that can be stored or transported. The basic idea of serialization is that an object writes its current state, usually indicated by the value of its member variables, to temporary (either memory or network streams) or persistent storage. Later, the object can be re-created by reading, or deserializing, the object's state from storage. Serialization handles all the details of object pointers and circular object references that are used when you serialize an object.

The serialized stream might be encoded using XML, SOAP, or a compact binary representation. The Formatter object that is used determines the format. The formatter is actually a pluggable component of a channel and a custom formatter can be plugged in to replace the standard XML or binary formatters supplied by remoting. Pluggable formatters allow the developer to serialize objects in the two supplied formats (binary and SOAP) or create their own.

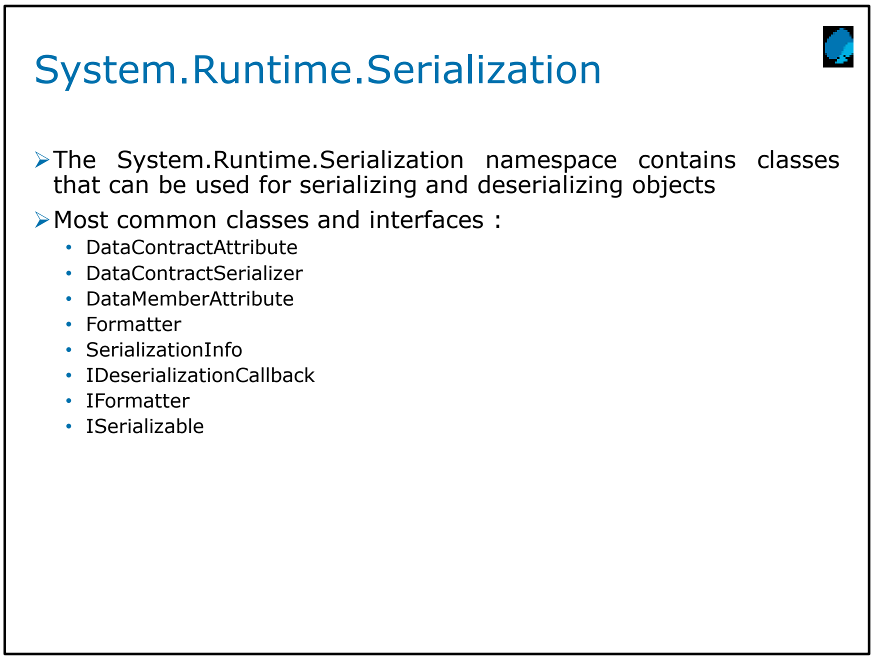
The System.Runtime. Formatters contains Two additional namescpaces (*.Binary and *.Soap) that provide two default formatters.



9.2: Need for Serialization

Why Use Serialization?

- Serialization is done:
  - So that the object can be recreated with its current state at a later point in time or at a different location
- Following are required to Serialize an object:
  - The object that is to serialize itself
  - A stream to contain the serialized object
  - A formatter used to serialize the object

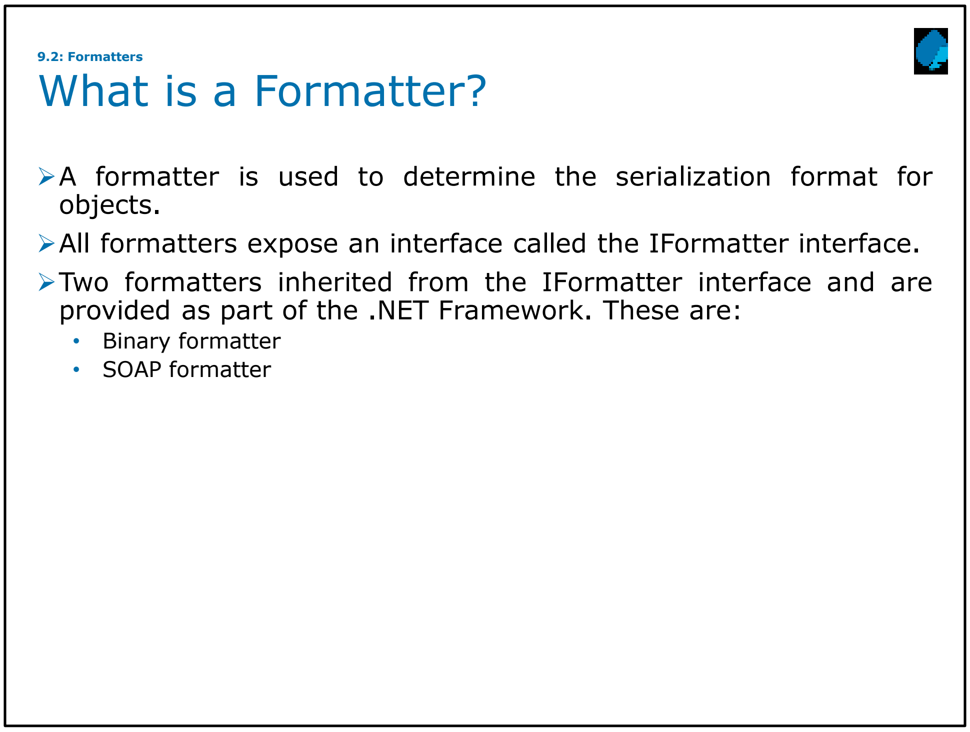


System.Runtime.Serialization

- The System.Runtime.Serialization namespace contains classes that can be used for serializing and deserializing objects
- Most common classes and interfaces :
  - DataContractAttribute
  - DataContractSerializer
  - DataMemberAttribute
  - Formatter
  - SerializationInfo
  - IDeserializationCallback
  - IFormatter
  - ISerializable

DataContractAttribute - Specifies that the type defines or implements a data contract and is serializable by a serializer, such as the DataContractSerializer. To make their type serializable, type authors must define a data contract for their type.
DataContractSerializer - Serializes and deserializes an instance of a type into an XML stream or document using a supplied data contract. This class cannot be inherited.
DataMemberAttribute - When applied to the member of a type, specifies that the member is part of a data contract and is serializable by the DataContractSerializer.
Formatter - Provides base functionality for the common language runtime serialization formatters.
SerializationInfo - Stores all the data needed to serialize or deserialize an object. This class cannot be inherited.
IDeserializationCallback - Indicates that a class is to be notified when deserialization of the entire object graph has been completed. Note that this interface is not called when deserializing with the XmlSerializer
IFormatter - Provides functionality for formatting serialized objects.
ISerializable - Allows an object to control its own serialization and

deserialization.



9.2: Formatters

What is a Formatter?

- A formatter is used to determine the serialization format for objects.
- All formatters expose an interface called the IFormatter interface.
- Two formatters inherited from the IFormatter interface and are provided as part of the .NET Framework. These are:
  - Binary formatter
  - SOAP formatter

BinaryFormatter type seriliazes your object to a stream using a compact binary format.

SoapFormatter type represents your object as a SOAP message.

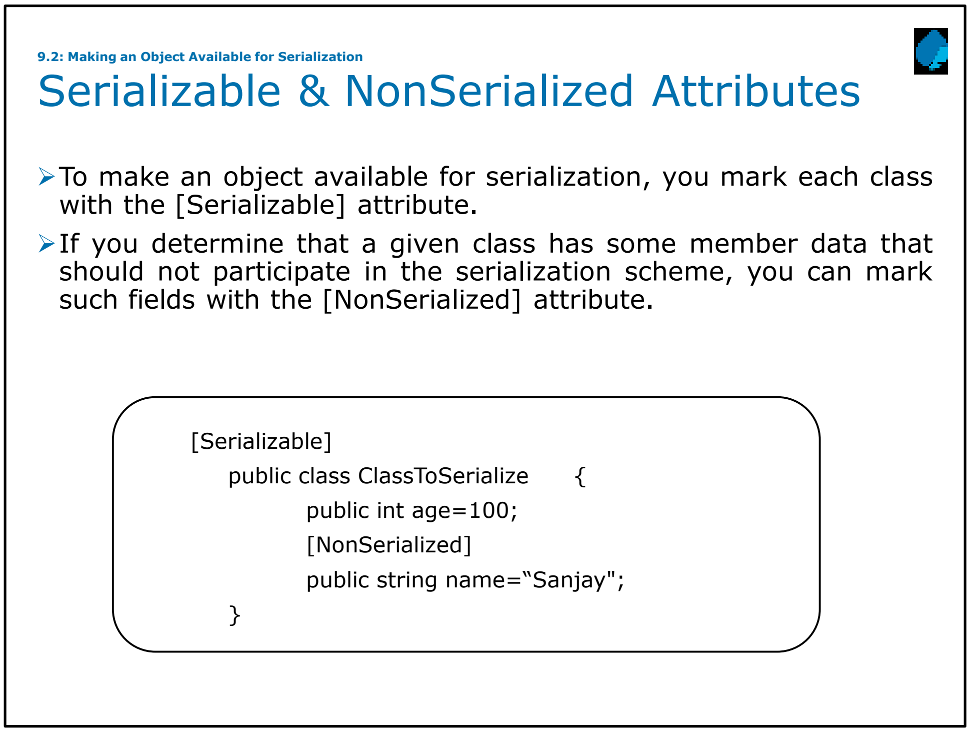
The Binary Formatter:

The Binary formatter provides binary encoding for compact serialization either for storage or for socket-based network streams. The BinaryFormatter class is generally not appropriate when data is meant to be passed through a firewall.

The SOAP Formatter:

The SOAP formatter provides formatting that can be used to enable objects to be serialized using the SOAP protocol. The Soap Formatter class is primarily used for serialization through firewalls or among diverse systems.

Explain the Usage of both the attributes specified in the example on the slide.



9.2: Making an Object Available for Serialization

Serializable & NonSerialized Attributes

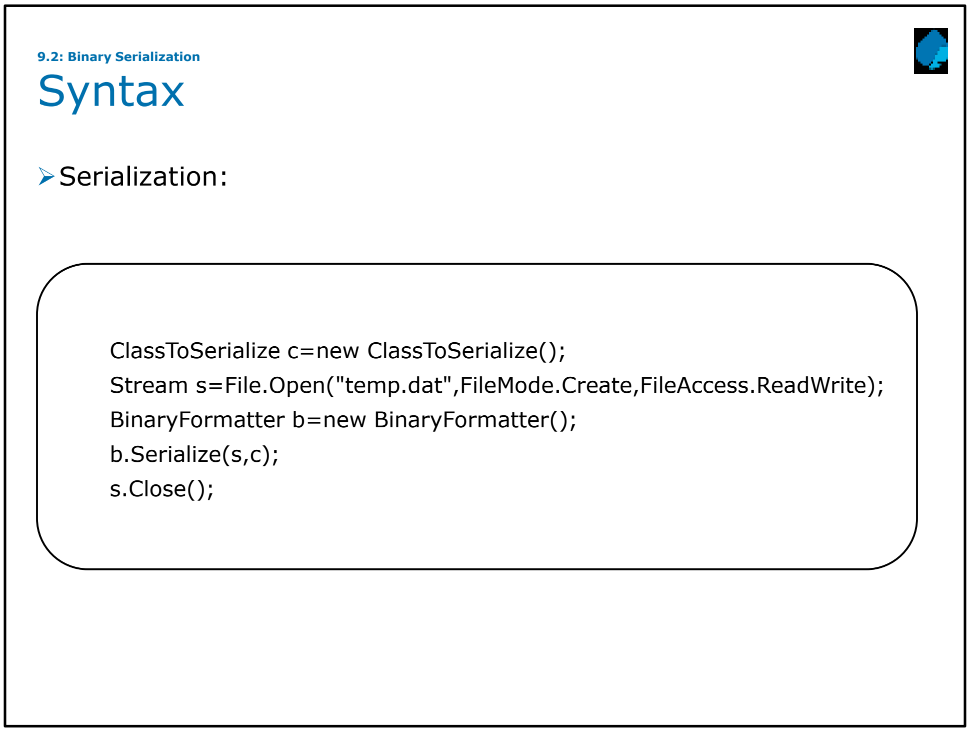
- To make an object available for serialization, you mark each class with the [Serializable] attribute.
- If you determine that a given class has some member data that should not participate in the serialization scheme, you can mark such fields with the [NonSerialized] attribute.

```
[Serializable]
    public class ClassToSerialize     {
            public int age=100;
            [NonSerialized]
            public string name="Sanjay";
    }
```

Serializable and NonSerialized Attributes:

To make an object available for serialization, you mark each class with the [Serializable] attribute. If you determine that a given class has some member data that should not participate in the serialization scheme, you can mark such fields with the [NonSerialized] attribute. This can be helpful if you have member variables(or Properties) in a serializable class that do not need to be "remembered"(e.g. constants, transient data, and so on).
The ClassToSerialize class is marked Serializable and a name data member is set as NonSerialized data member by setting the [NonSerialized] attribute to this member. If you do not make a class serializable a exception is thrown when you try to serialize the object of that class.

Additional notes for instructor



9.2: Binary Serialization

Syntax

➢ Serialization:

```
ClassToSerialize c=new ClassToSerialize();
Stream s=File.Open("temp.dat",FileMode.Create,FileAccess.ReadWrite);
BinaryFormatter b=new BinaryFormatter();
b.Serialize(s,c);
s.Close();
```

Binary Serialization:

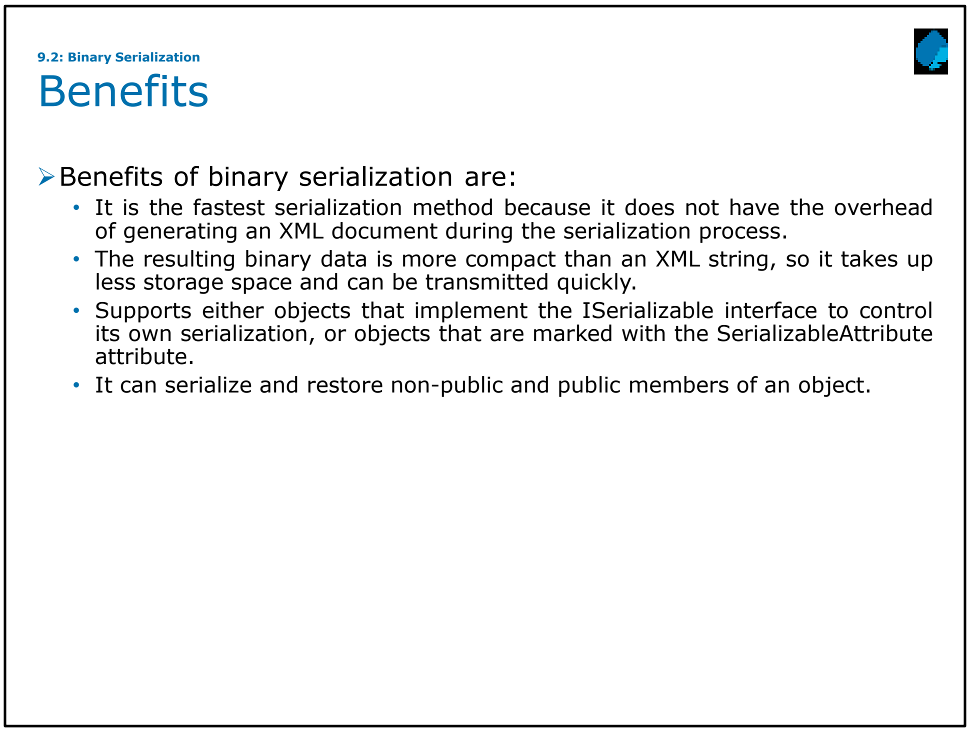
The above code demonstrated how Binary serialization method is used to serialize an object.
The general steps for serializing are :
Create an instance of File that will store serialized object.
Create a stream from the file object.
Create an instance of BinaryFormatter.
Call serialize method of the instance passing it stream and object to serialize.

Deserialization: Syntax

➢ Deserialization:

```
Stream s=File.Open("temp.dat",FileMode.Open,FileAccess.Read);
 BinaryFormatter b=new BinaryFormatter();
 c=(ClassToSerialize)b.Deserialize(s);
 Console.WriteLine(c.age);
 Console.WriteLine(c.name);
 s.Close();
```

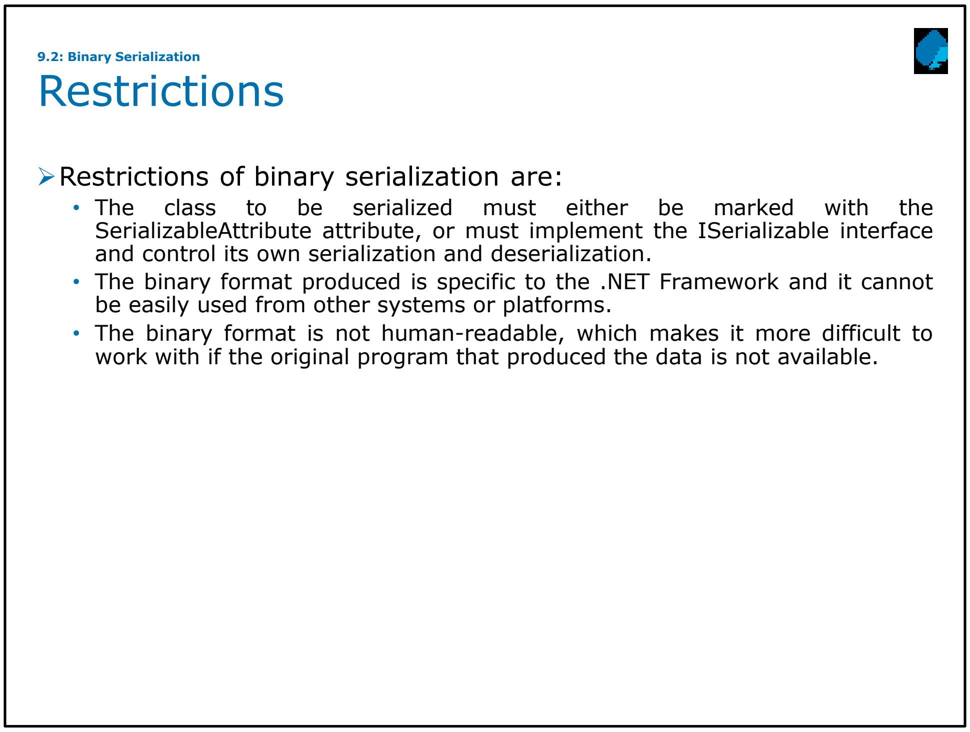
Binary Deserialization:
The above code demonstrated how Binary Deserialization method is used to deserialize an object.
The steps for de-serializing the object are similar. The only change is that you need to call deserialize method of BinaryFormatter object.



Benefits

- Benefits of binary serialization are:
  - It is the fastest serialization method because it does not have the overhead of generating an XML document during the serialization process.
  - The resulting binary data is more compact than an XML string, so it takes up less storage space and can be transmitted quickly.
  - Supports either objects that implement the ISerializable interface to control its own serialization, or objects that are marked with the SerializableAttribute attribute.
  - It can serialize and restore non-public and public members of an object.

The above are the various benefits of using Binary Serialization.



Restrictions

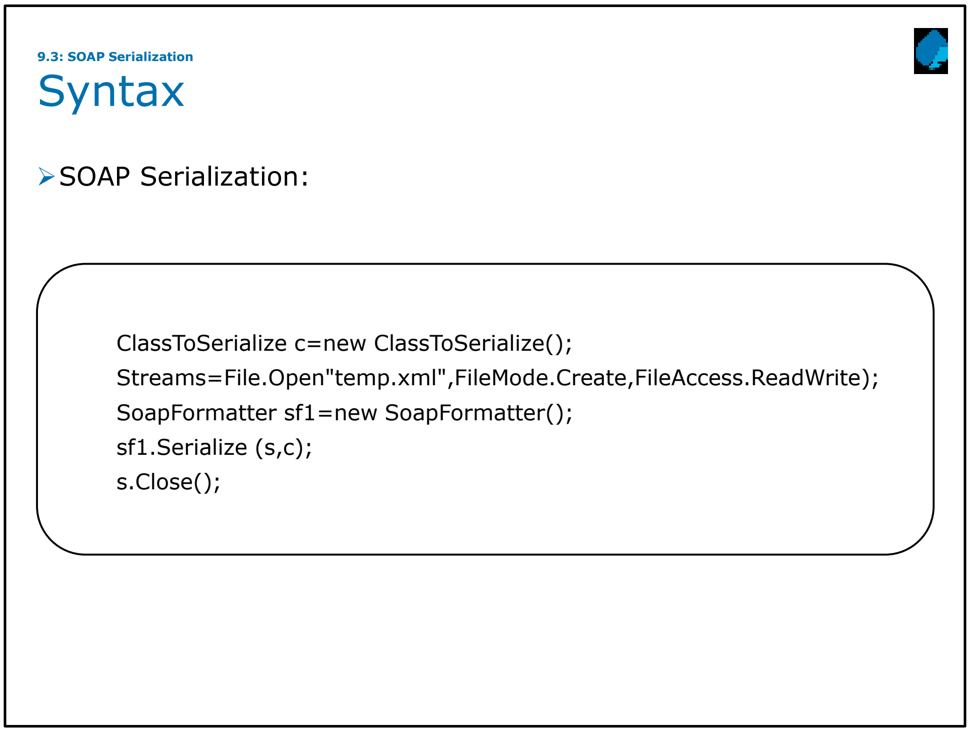
- Restrictions of binary serialization are:
  - The class to be serialized must either be marked with the SerializableAttribute attribute, or must implement the ISerializable interface and control its own serialization and deserialization.
  - The binary format produced is specific to the .NET Framework and it cannot be easily used from other systems or platforms.
  - The binary format is not human-readable, which makes it more difficult to work with if the original program that produced the data is not available.

The above slide specifies the restrictions for Binary serialization.

Demo

- Demo on Serialization and DeSerialization using Binary Formatter

Add the notes here.



Syntax

- SOAP Serialization:

```
ClassToSerialize c=new ClassToSerialize();
Streams=File.Open"temp.xml",FileMode.Create,FileAccess.ReadWrite);
SoapFormatter sf1=new SoapFormatter();
sf1.Serialize (s,c);
s.Close();
```

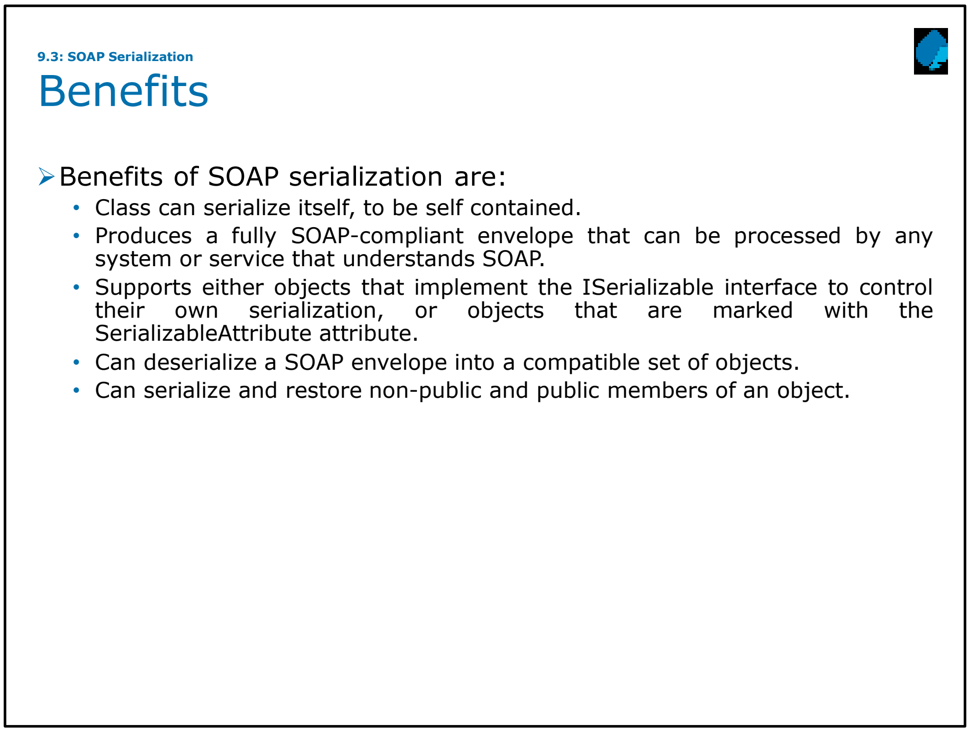
The above code demonstrated how do you serialize a object using SOAP serialization method.

SOAP Deserialization: Syntax

- SOAP Deserialization:

```
ClassToSerialize c;
Stream s=File.Open("temp.xml",FileMode.Open,FileAccess.Read);
c=(ClassToSerialize)(new SoapFormatter().Deserialize(s));
//c=(ClassToSerialize)sf.Deserialize(s);
Console.WriteLine(c.age);
Console.WriteLine(c.name);
s.Close();
```

The above code demonstrated how do you Deserialize a object using SOAP serialization method.



9.3: SOAP Serialization

Benefits

- Benefits of SOAP serialization are:
  - Class can serialize itself, to be self contained.
  - Produces a fully SOAP-compliant envelope that can be processed by any system or service that understands SOAP.
  - Supports either objects that implement the ISerializable interface to control their own serialization, or objects that are marked with the SerializableAttribute attribute.
  - Can deserialize a SOAP envelope into a compatible set of objects.
  - Can serialize and restore non-public and public members of an object.

The above are the various benefits of using SOAP Serialization.

Restrictions

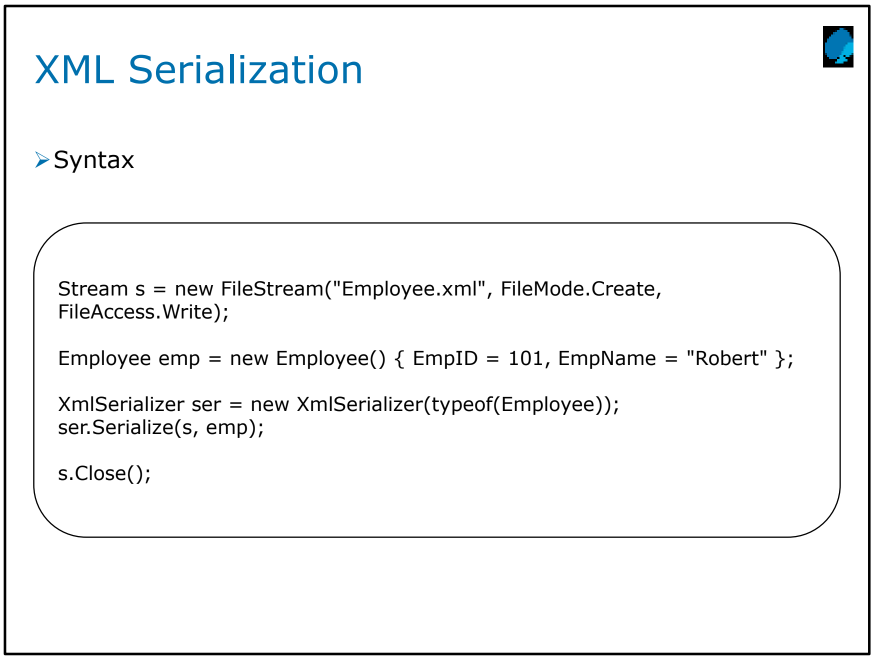
➢ Restrictions of SOAP serialization are:
- The class to be serialized must either be marked with the SerializableAttribute attribute, or must implement the ISerializable interface and control its own serialization and deserialization.
- Only understands SOAP. It cannot work with arbitrary XML schemas.

The above are the various restrictions of using SOAP Serialization.

Demo

- Demo on SOAP Serialization

Add the notes here.



XML Serialization

➢Syntax

```
Stream s = new FileStream("Employee.xml", FileMode.Create,
FileAccess.Write);

Employee emp = new Employee() { EmpID = 101, EmpName = "Robert" };

XmlSerializer ser = new XmlSerializer(typeof(Employee));
ser.Serialize(s, emp);

s.Close();
```

XML serialization serializes the public fields and properties of an object, or parameters and return values of methods, into a XML stream that conforms to a specific XML Schema definition (XSD) document. XML serialization results in strongly typed classes with properties and fields that are converted to XML.

System.Xml.Serialization contains the classes necessary for serializing and deserializing XML.

Apply attributes to classes and class members to control the way the XmlSerializer serializes or deserializes an instance of the class.

XML Deserialization

- Syntax

```
Stream s = new FileStream("Employee.xml", FileMode.Open,
FileAccess.Read);

XmlSerializer ser = new XmlSerializer(typeof(Employee));

Employee anotherEmp = (Employee)ser.Deserialize(s);

s.Close();
```

XML Serialization Benefits

- Benefits of XML serialization are:
  - XmlSerializer class gives complete and flexible control when you serialize an object as XML
  - If you are creating an XML Web Service, you can apply attributes that control serialization to classes and members to ensure that the XML output conforms to a specific schema
  - You have no constraints on the applications you develop, as long as the XML stream that is generated conforms to a given schema
  - Supports either objects that implement the ISerializable interface to control their own serialization, or objects that are marked with the SerializableAttribute attribute

XML Serialization Restriction

- Restrictions of XML serialization are:
  - The class to be serialized must either be marked with the SerializableAttribute attribute, or must implement the ISerializable interface and control its own serialization and deserialization
  - Only public properties and fields can be serialized. Properties must have public accessors (get and set methods). If you must serialize non-public data, use the DataContractSerializer class rather than XML serialization
  - A class must have a default constructor to be serialized by XmlSerializer

Demo

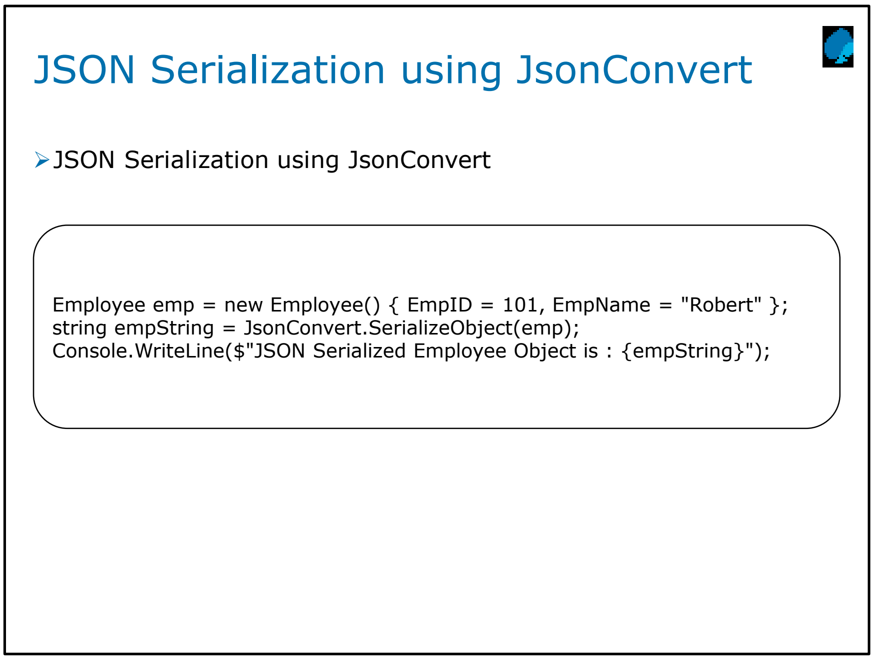
- Demo on XML Serialization

What is JSON?

- JSON stands for JavaScript Object Notation
- JSON is lightweight format for storing and transporting data
- JSON is often used when data is sent from a server to a web page
- JSON is “self-describing” and easy to understand
- JSON data is written as name/value pairs. A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value
- JSON objects are written inside curly braces

JSON Serialization

- JSON text and .NET object conversion can be done by using JsonSerializer
- The JsonSerializer converts .NET objects into their JSON equivalent and back again by mapping the .NET object property names to the JSON property names and copies the values
- JSON Serialization/Deserialization can be implemented by three ways :
  - Using JsonConvert
  - Using JsonSerializer
  - Using DataContractJsonSerializer



JSON Serialization using JsonConvert

➢JSON Serialization using JsonConvert

```
Employee emp = new Employee() { EmpID = 101, EmpName = "Robert" };
string empString = JsonConvert.SerializeObject(emp);
Console.WriteLine($"JSON Serialized Employee Object is : {empString}");
```

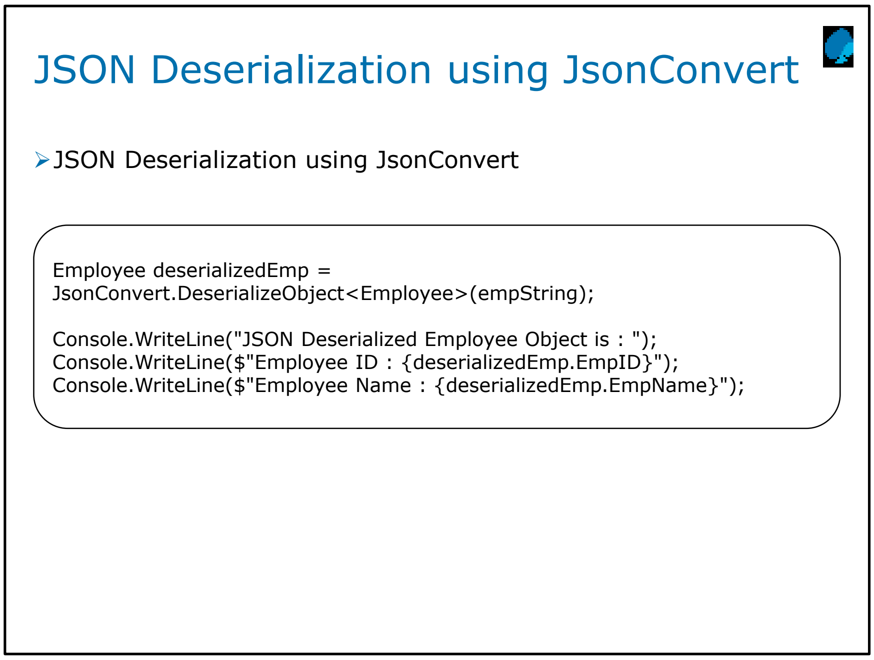
Json.NET is a third-party library, helps conversion between JSON text and .NET object using JsonSerializer.

Let’s start learning how to install and implement:

In Visual Studio, go to Tools Menu -> Choose Library Package Manager -> Package Manager Console. It opens a command window where we need to put the following command to install Newtonsoft.Json.

Install-Package Newtonsoft.Json
OR
In Visual Studio, Tools menu -> Manage Nuget Package Manager Solution and type “JSON.NET” to search it online.

In Serialization, it converts a custom .Net object to a Json string. In the following code, it creates an instance of BlogSiteclass and assigns some values to its properties. Then it calls static method SerializeObject() of JsonConvert class by passing object(BlogSites). It returns JSON data in string format.

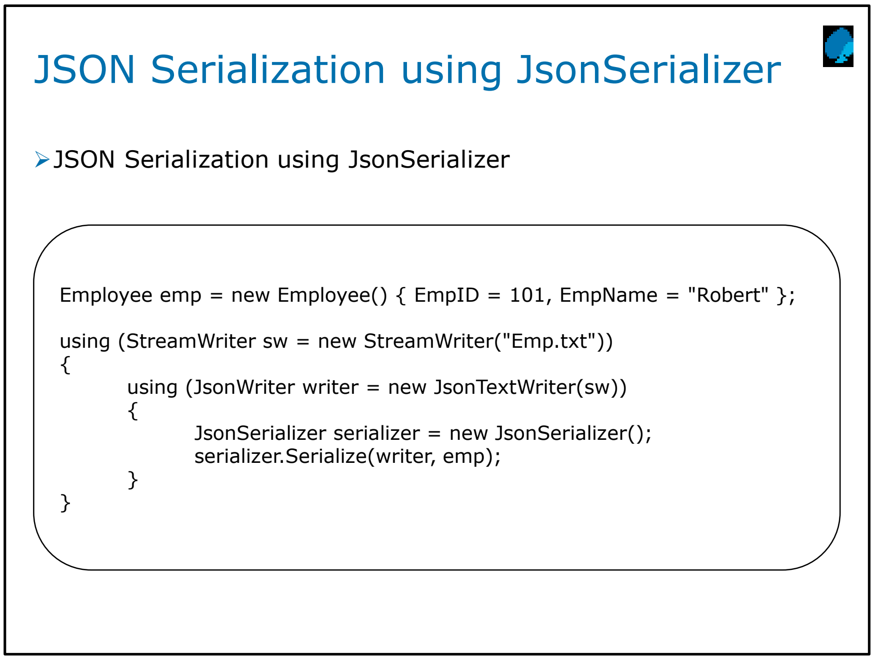


JSON Deserialization using JsonConvert

- JSON Deserialization using JsonConvert

```
Employee deserializedEmp =
JsonConvert.DeserializeObject<Employee>(empString);

Console.WriteLine("JSON Deserialized Employee Object is : ");
Console.WriteLine($"Employee ID : {deserializedEmp.EmpID}");
Console.WriteLine($"Employee Name : {deserializedEmp.EmpName}");
```

In Deserialization, it does the opposite of Serialization which means it converts JSON string to custom .Net object. In the following code, it calls static method DeserializeObject() of JsonConvert class by passing JSON data. It returns custom object from JSON data.



JSON Serialization using JsonSerializer

- JSON Serialization using JsonSerializer

```
Employee emp = new Employee() { EmpID = 101, EmpName = "Robert" };

using (StreamWriter sw = new StreamWriter("Emp.txt"))
{
        using (JsonWriter writer = new JsonTextWriter(sw))
        {
                JsonSerializer serializer = new JsonSerializer();
                serializer.Serialize(writer, emp);
        }
}
```

Json.NET is a third-party library, helps conversion between JSON text and .NET object using JsonSerializer.

Let’s start learning how to install and implement:

In Visual Studio, go to Tools Menu -> Choose Library Package Manager -> Package Manager Console. It opens a command window where we need to put the following command to install Newtonsoft.Json.

Install-Package Newtonsoft.Json
OR
In Visual Studio, Tools menu -> Manage Nuget Package Manager Solution and type “JSON.NET” to search it online.

For more control over how an object is serialized, the JsonSerializer can be used directly. The JsonSerializer is able to read and write JSON text directly to a stream via JsonTextReader and JsonTextWriter

JSON Deserialization using JsonSerializer

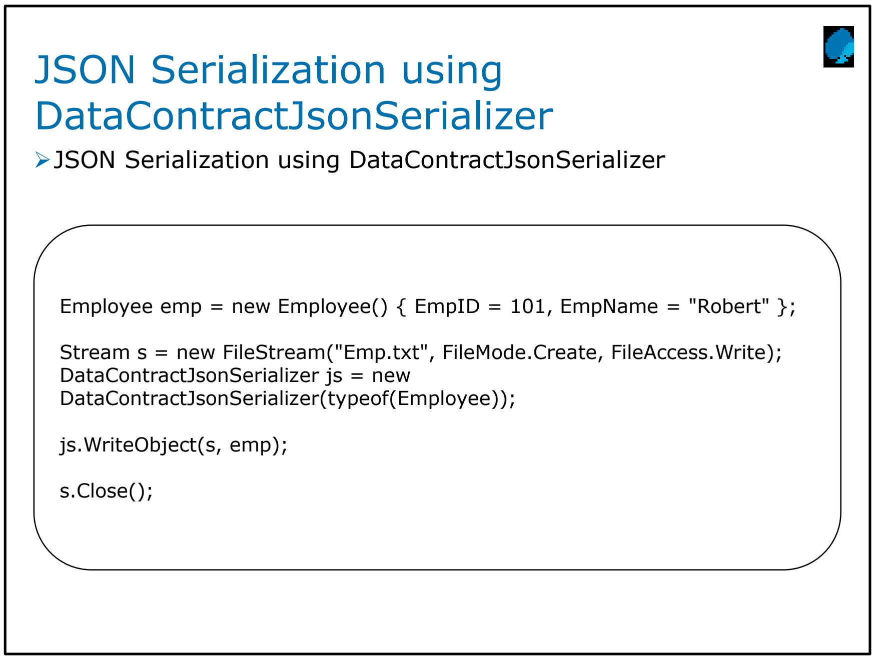
- JSON Deserialization using JsonSerializer

```
using (StreamReader sr = new StreamReader("Emp.txt"))
{
        using (JsonReader reader = new JsonTextReader(sr))
        {
                JsonSerializer serializer = new JsonSerializer();
                Employee desEmp = serializer.Deserialize<Employee>(reader);

                Console.WriteLine($"Employee ID : {desEmp.EmpID}");
                Console.WriteLine($"Employee Name : {desEmp.EmpName}");
        }
}
```

DataContract and DataMember

- 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
- DataContractAttribute class specifies that the type defines or implements a data contract and is serializable by a serializer, such as the DataContractSerializer
- DataMemberAttribute when applied to the member of a type, specifies that the member is part of a data contract and is serializable by the DataContractSerializer

```
[DataContract]
class Employee
{
        [DataMember]
        public int EmpID { get; set; }

        [DataMember]
        public string EmpName { get; set; }
}
```



JSON Serialization using DataContractJsonSerializer

- JSON Serialization using DataContractJsonSerializer

```
Employee emp = new Employee() { EmpID = 101, EmpName = "Robert" };

Stream s = new FileStream("Emp.txt", FileMode.Create, FileAccess.Write);
DataContractJsonSerializer js = new
DataContractJsonSerializer(typeof(Employee));

js.WriteObject(s, emp);

s.Close();
```

DataContractJsonSerializer class helps to serialize and deserialize JSON. It is present in namespace System.Runtime.Serialization.Json which is available in assembly System.Runtime.Serialization.dll. Using the class we can serialize an object into JSON data and deserialize JSON data into an object.
Normally, JSON serialization and deserialization is handled automatically by Windows Communication Foundation (WCF) when you use data contract types in service operations that are exposed over AJAX-enabled endpoints. However, in some cases you may need to work with JSON data directly

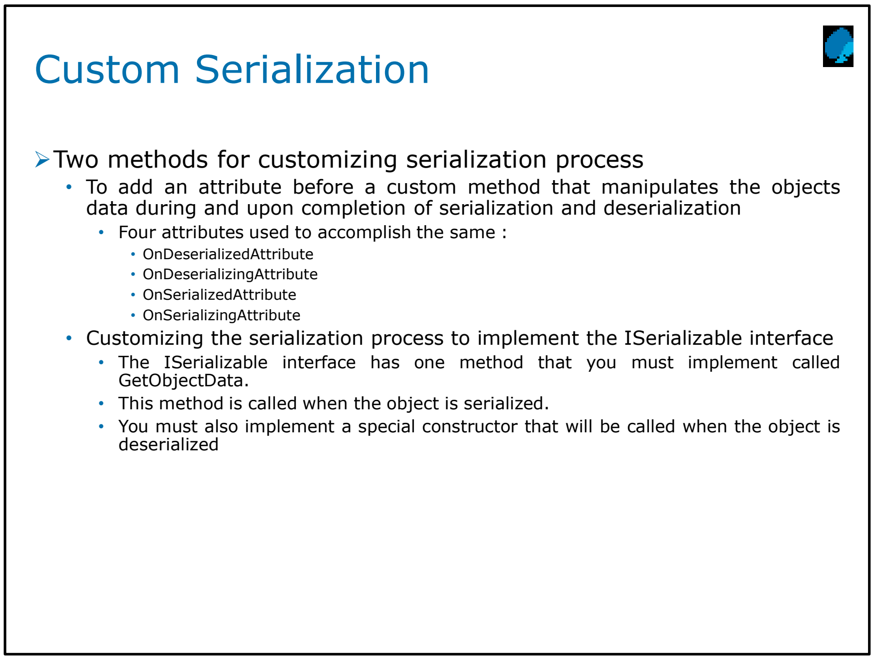
JSON Deserialization using DataContractJsonSerializer

- JSON Deserialization using DataContractJsonSerializer

```
Stream s = new FileStream("Emp.txt", FileMode.Open, FileAccess.Read);
Employee desEmp = js.ReadObject(s) as Employee;
s.Close();

Console.WriteLine($"Employee ID : {desEmp.EmpID}");
Console.WriteLine($"Employee Name : {desEmp.EmpName}");
```

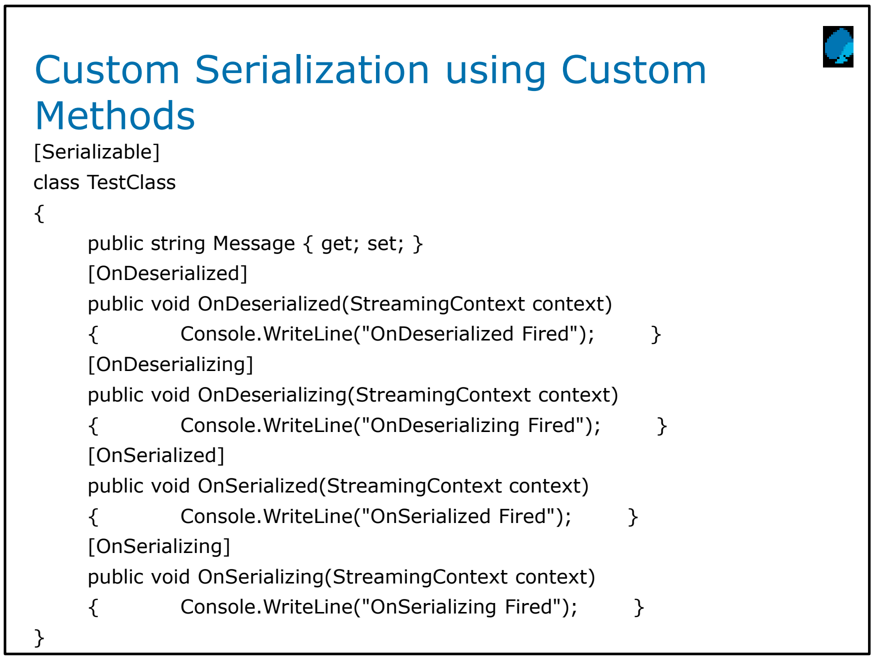
Demo

- Demo on XmlConvert Class
- Demo on XmlSerializer Class
- Demo on DataContractJsonSerializer Class



Custom Serialization

- Two methods for customizing serialization process
  - To add an attribute before a custom method that manipulates the objects data during and upon completion of serialization and deserialization
    - Four attributes used to accomplish the same :
      - OnDeserializedAttribute
      - OnDeserializingAttribute
      - OnSerializedAttribute
      - OnSerializingAttribute
  - Customizing the serialization process to implement the ISerializable interface
    - The ISerializable interface has one method that you must implement called GetObjectData.
    - This method is called when the object is serialized.
    - You must also implement a special constructor that will be called when the object is deserialized

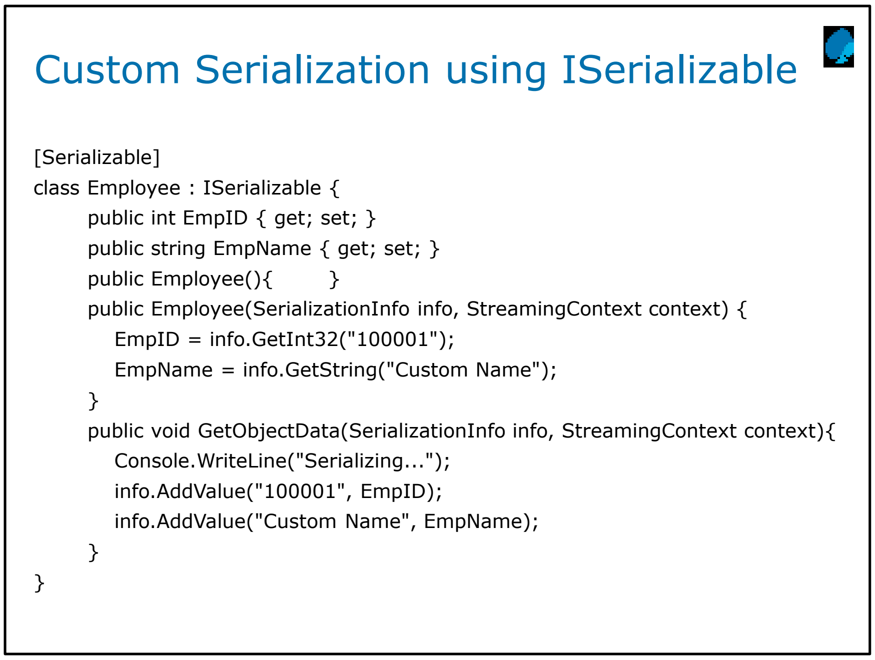


Custom Serialization using Custom Methods

```
[Serializable]
class TestClass
{
        public string Message { get; set; }
        [OnDeserialized]
        public void OnDeserialized(StreamingContext context)
        {           Console.WriteLine("OnDeserialized Fired");       }
        [OnDeserializing]
        public void OnDeserializing(StreamingContext context)
        {           Console.WriteLine("OnDeserializing Fired");       }
        [OnSerialized]
        public void OnSerialized(StreamingContext context)
        {           Console.WriteLine("OnSerialized Fired");       }
        [OnSerializing]
        public void OnSerializing(StreamingContext context)
        {           Console.WriteLine("OnSerializing Fired");       }
}
```

OnDeserializedAttribute - When applied to a method, specifies that the method is called immediately after deserialization of an object in an object graph. The order of deserialization relative to other objects in the graph is non-deterministic.
OnDeserializingAttribute - When applied to a method, specifies that the method is called during deserialization of an object in an object graph. The order of deserialization relative to other objects in the graph is non-deterministic.
OnSerializedAttribute - When applied to a method, specifies that the method is called after serialization of an object in an object graph. The order of serialization relative to other objects in the graph is non-deterministic.
OnSerializingAttribute - When applied to a method, specifies that the method is during serialization of an object in an object graph. The order of serialization relative to other objects in the graph is non-deterministic.

If you run the code for any of the serializer objects and put breakpoints in each method, you can see when each method is called. This enables you to customize the input or output in case you have enhancements to your objects in later versions, and properties are missing from your persisted files.



Custom Serialization using ISerializable

```
[Serializable]
class Employee : ISerializable {
        public int EmpID { get; set; }
        public string EmpName { get; set; }
        public Employee(){       }
        public Employee(SerializationInfo info, StreamingContext context) {
            EmpID = info.GetInt32("100001");
            EmpName = info.GetString("Custom Name");
        }
        public void GetObjectData(SerializationInfo info, StreamingContext context){
            Console.WriteLine("Serializing...");
            info.AddValue("100001", EmpID);
            info.AddValue("Custom Name", EmpName);
        }
}
```

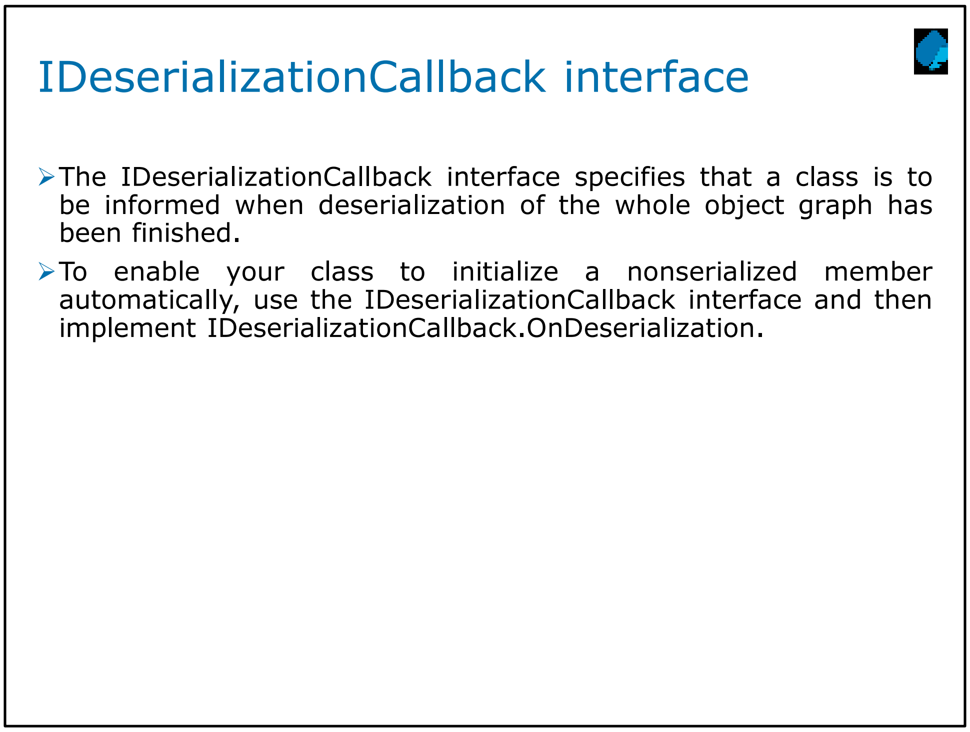
The ISerializable interface has one method that you must implement called GetObjectData. This method is called when the object is serialized. You must also implement a special constructor that will be called when the object is deserialized

GetObjectData method has a parameter of type SerializationInfo, which enables you to customize the name and data that will be written to the stream.

When the data is deserialized, the constructor is called. Here you call methods on the SerializationInfo object to get the value based on the custom name you gave the field.

Demo

- Demo on Custom Serialization using Attributes
- Demo on Custom Serialization using ISerializable



IDeserializationCallback interface

- The IDeserializationCallback interface specifies that a class is to be informed when deserialization of the whole object graph has been finished.
- To enable your class to initialize a nonserialized member automatically, use the IDeserializationCallback interface and then implement IDeserializationCallback.OnDeserialization.

When an object is deserialized, the contained objects in its serialization stream have not yet been deserialized. If your code tries to call into any instance property or method on the sub object instance, it may get an exception because that instance has not been initialized yet. If you need to make calls into contained objects to complete the deserialization process, then you need to implement the IDeserializationCallback interface.

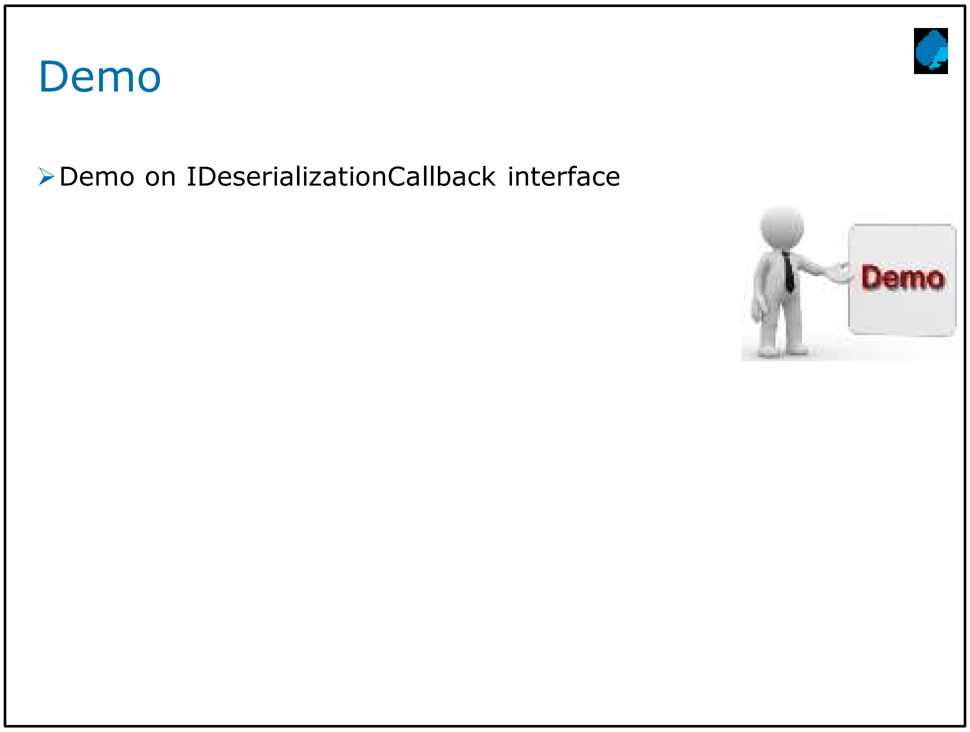
Adding support for the IDeserializationCallback interface enables your object to perform additional initialization after the deserialization of your object and all sub-objects. This interface is not called until after your object and all of its child objects have been deserialized. This interface has one method to implement and that is the OnDeserialization method.

IDeserializationCallback interface

- Each time your class is deserialized, the runtime calls the IDeserializationCallback.OnDeserialization method after deserialization is complete

Example

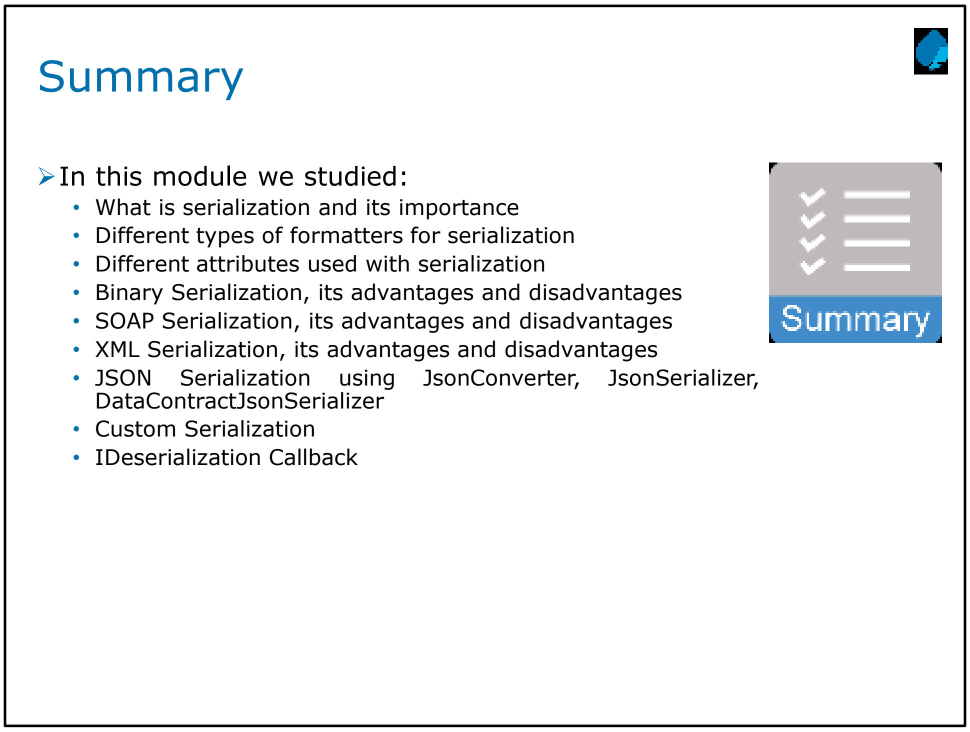
```
[Serializable]
class ShoppingCartItem : IDeserializationCallback
{
    public int productId;
    public decimal price;
    public int quantity;
    [NonSerialized]
    public decimal total;
    public ShoppingCartItem(int _productID, decimal _price, int _quantity)
    {
        productId = _productID;
        price = _price;
        quantity = _quantity;
        total = price * quantity;
    }
    void IDeserializationCallback.OnDeserialization(Object sender)
    {
        // After deserialization, calculate the total
        total = price * quantity;
    }
}
```



Demo

- Demo on IDeserializationCallback interface

Add the notes here.



Summary

- In this module we studied:
  - What is serialization and its importance
  - Different types of formatters for serialization
  - Different attributes used with serialization
  - Binary Serialization, its advantages and disadvantages
  - SOAP Serialization, its advantages and disadvantages
  - XML Serialization, its advantages and disadvantages
  - JSON Serialization using JsonConverter, JsonSerializer, DataContractJsonSerializer
  - Custom Serialization
  - IDeserialization Callback

Summary

Add the notes here.

Review Questions

- What are files and streams?
- What is Serialization?
- What is the use of [NonSerialized()] attribute?
- What are the benefits and drawbacks of Binary Serialization?
- What are the benefits and drawbacks of SOAP Serialization?
- Explain XML Serialization
- Explain JSON Serialization
- What is Custom Serialization?