.NET Framework 4.6 & C# 6.0

Lesson 9

IO & 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

9.1: System.IO Namespace

Using I/O

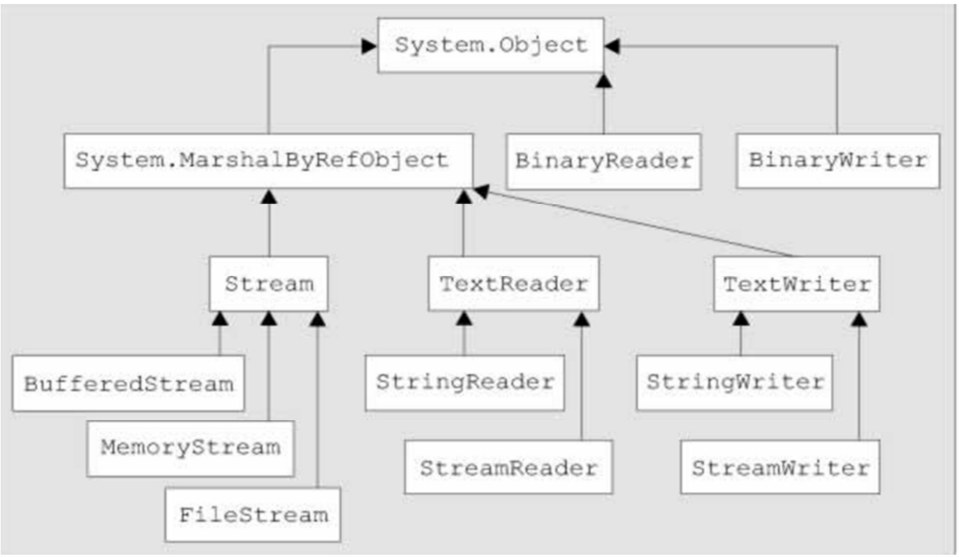
- The System.IO namespace contains types that allow synchronous and asynchronous reading and writing on data streams and files.
- A file is an ordered and named collection of a particular sequence of bytes having persistent storage.
- In contrast, streams provide a way to write and read bytes to and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

9.1: System.IO Namespace

Exploring System.IO Namespace



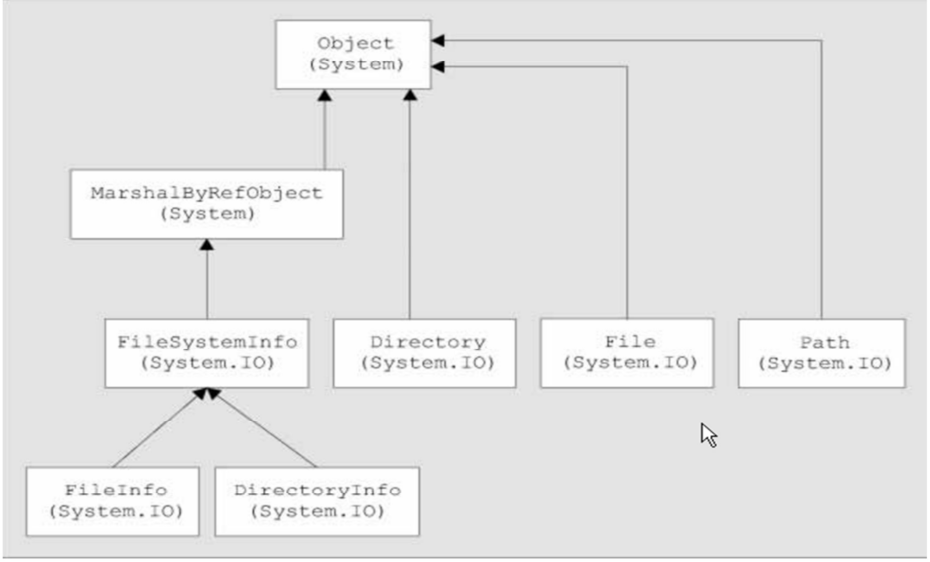
4

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



9.1: System.IO Namespace

Directory and File Info Types

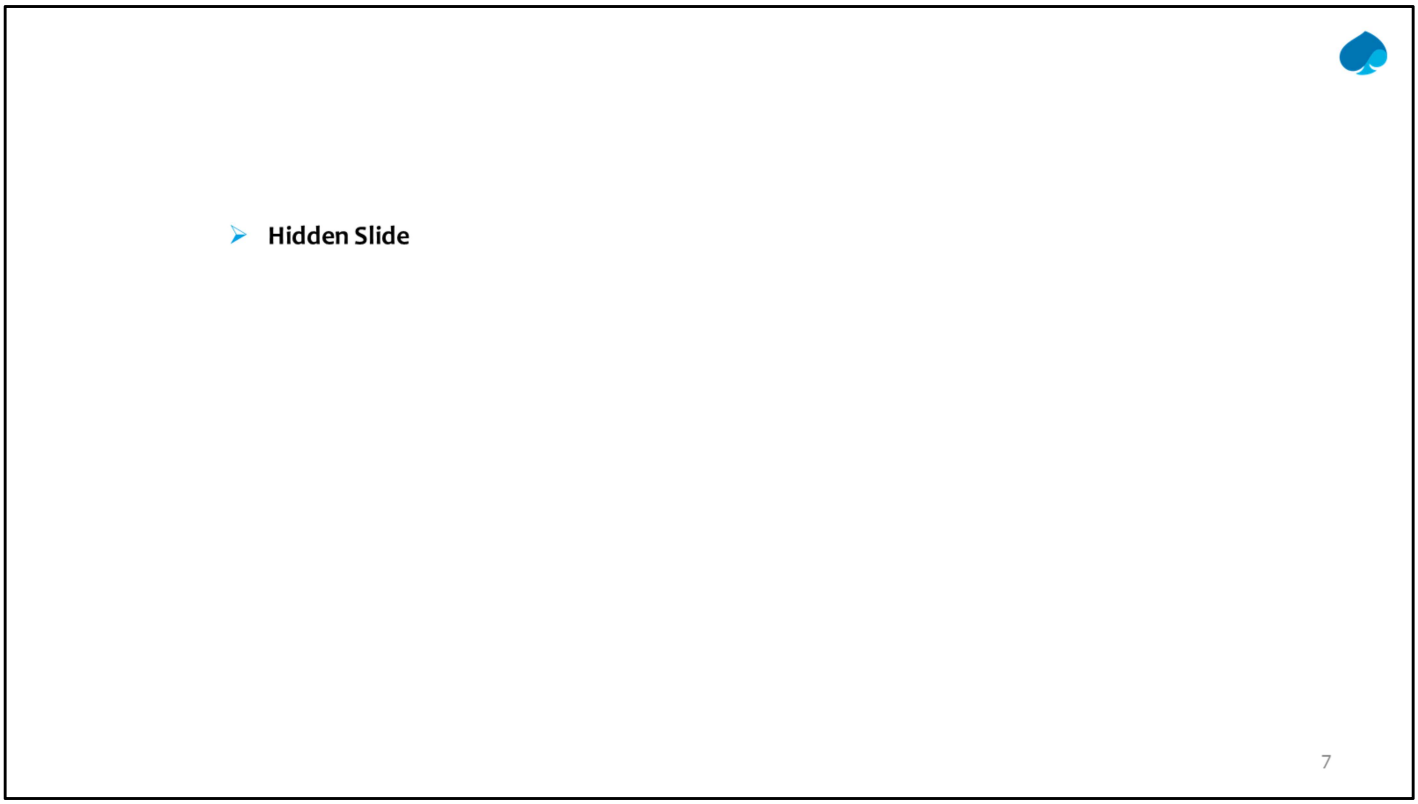
- System.IO provides four types that allow you to manipulate individual files, as well as interact with a machine's directory structure.
- The Directory and File types, expose creation, deletion, and manipulation operations using various static members.
- The closely related FileInfo and DirectoryInfo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



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.

8

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..

- Hidden Slide

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.

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.

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

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

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.

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";
    }
```

14

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.

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.

9.2: 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.

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.

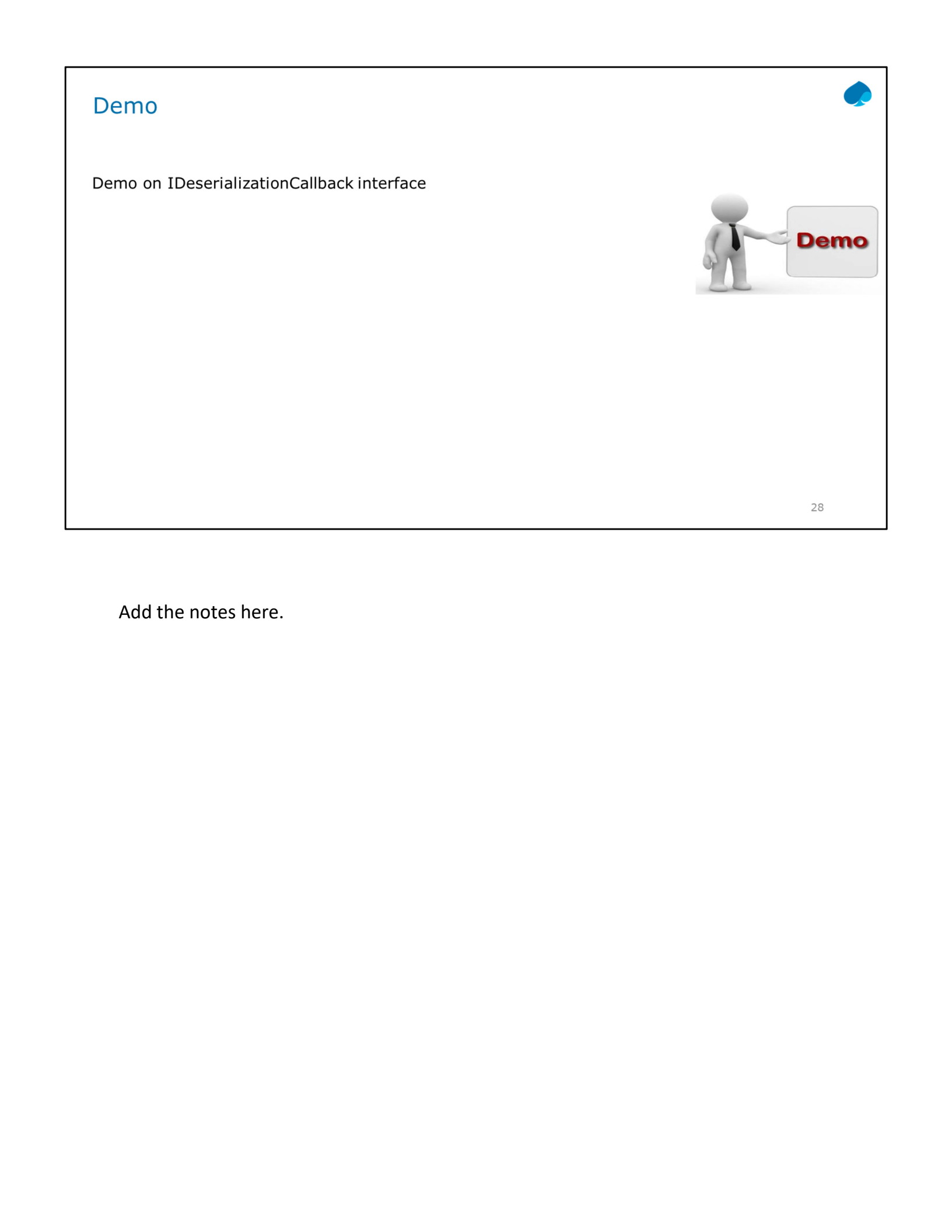
IDeserializationCallback interface

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

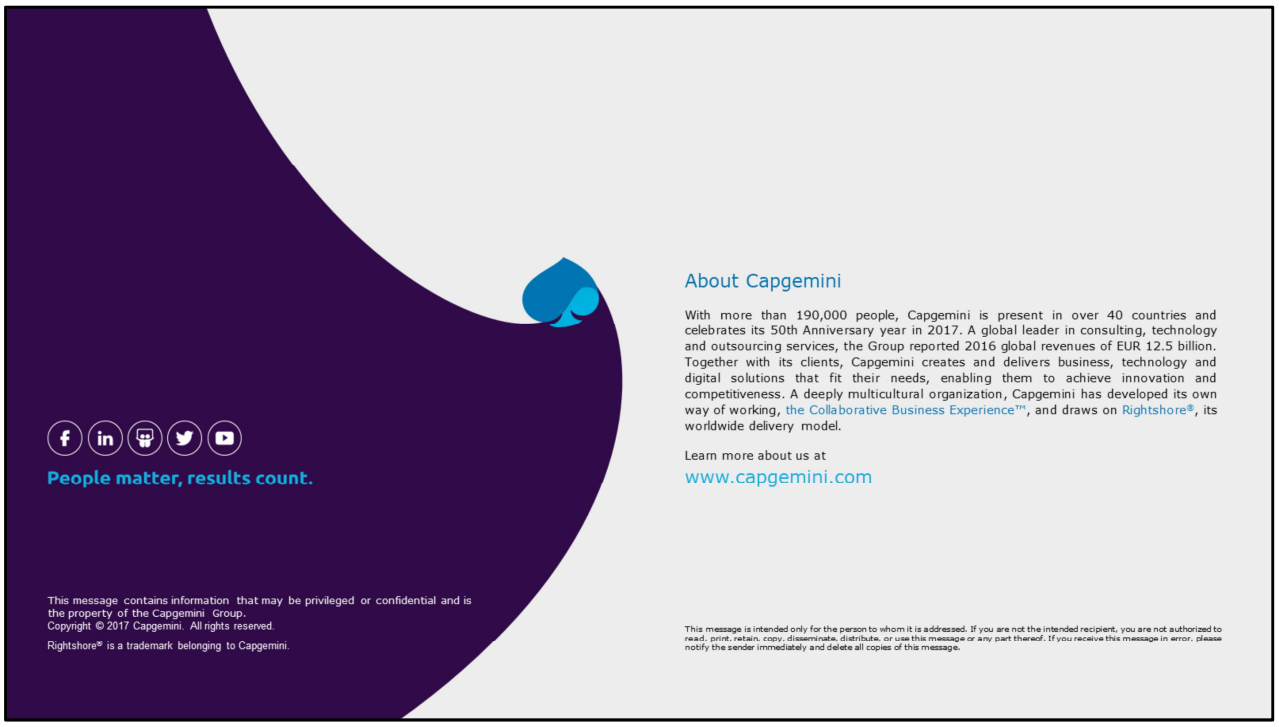
In this module we studied:

- 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

Add the notes here.

Review Questions

1. What are files and streams?
2. What is Serialization?
3. What is the use of [NonSerialized()] attribute?
4. What are the benefits and drawbacks of Binary Serialization?
5. What are the benefits and drawbacks of SOAP Serialization?


People matter, results count.
This message contains information that may be privileged or confidential and is the property of the Capgemini Group.
Copyright © 2017 Capgemini. All rights reserved.
Rightshore® is a trademark belonging to Capgemini.
About Capgemini
With more than 190,000 people, Capgemini is present in over 40 countries and celebrates its 50th Anniversary year in 2017. A global leader in consulting, technology and outsourcing services, the Group reported 2016 global revenues of EUR 12.5 billion. Together with its clients, Capgemini creates and delivers business, technology and digital solutions that fit their needs, enabling them to achieve innovation and competitiveness. A deeply multicultural organization, Capgemini has developed its own way of working, the Collaborative Business Experience™, and draws on Rightshore®, its worldwide delivery model.
Learn more about us at
www.capgemini.com
This message is intended only for the person to whom it is addressed. If you are not the intended recipient, you are not authorized to read, print, retain, copy, disseminate, distribute, or use this message or any part thereof. If you receive this message in error, please notify the sender immediately and delete all copies of this message.