Working with XML and JSON Serializing

Objectives

- Overview Serialization in .NET
- Understanding Serialization Engines in .NET
- Explain about how serialization works
- Describe use Serialization
- Overview XML Serialization
- Overview JSON (JavaScript Object Notation) Serialization
- Create demo using XML with WPF application
- Create demo XML Serialization in .NET application
- Create demo JSON Serialization in .NET application

Overview .NET Serialization

Understanding Serialization in .NET

- **Serialization** is the act of taking an in-memory object or object graph (set of objects that reference one another) and flattening it into a stream of bytes, XML, JSON, or a similar representation that can be stored or transmitted
- **Deserialization** works in reverse, taking a data stream and reconstituting it into an in-memory object or object graph
- There are four serialization engines in .NET :
  - XmlSerializer (XML)
  - JsonSerializer (JSON)
  - The data contract serializer (XML and JSON)
  - The binary serializer (binary)

Understanding Serialization Engines

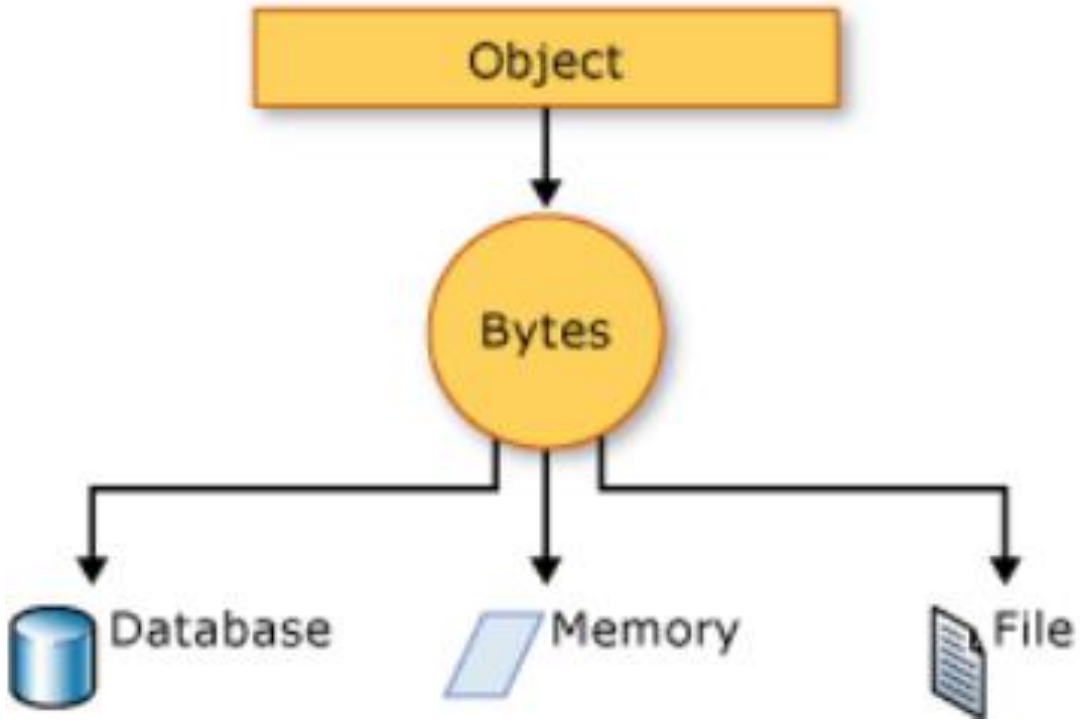
- **XmlSerializer:** Serializes and deserializes objects into and from XML documents. The XmlSerializer enables us to control how objects are encoded into XML
- **JsonSerializer:** Provides functionality to serialize objects or value types to JSON and to deserialize JSON into objects or value types
- **The Data Contract Serializer:** Serializes and deserializes an instance of a type into an XML stream or document using a supplied *Data Contract* (classes)

Understanding Serialization Engines

- **The Binary Serializer:** Serialization can be defined as the process of storing the state of an object to a storage medium. During this process, the public and private fields of the object and the name of the class, including the assembly containing the class, are converted to a stream of bytes, which is then written to a data stream. When the object is subsequently deserialized, an exact clone of the original object is created

How Serialization Works

- The object is serialized to a stream that carries the data. The stream may also have information about the object's type, such as its version, culture, and assembly name. From that stream, the object can be stored in a database, a file, or memory



Uses for Serialization

- Serialization allows the developer to save the state of an object and re-create it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions such as:
  - Sending the object to a remote application by using a web service
  - Passing an object from one domain to another
  - Passing an object through a firewall as a JSON or XML string
  - Maintaining security or user-specific information across applications

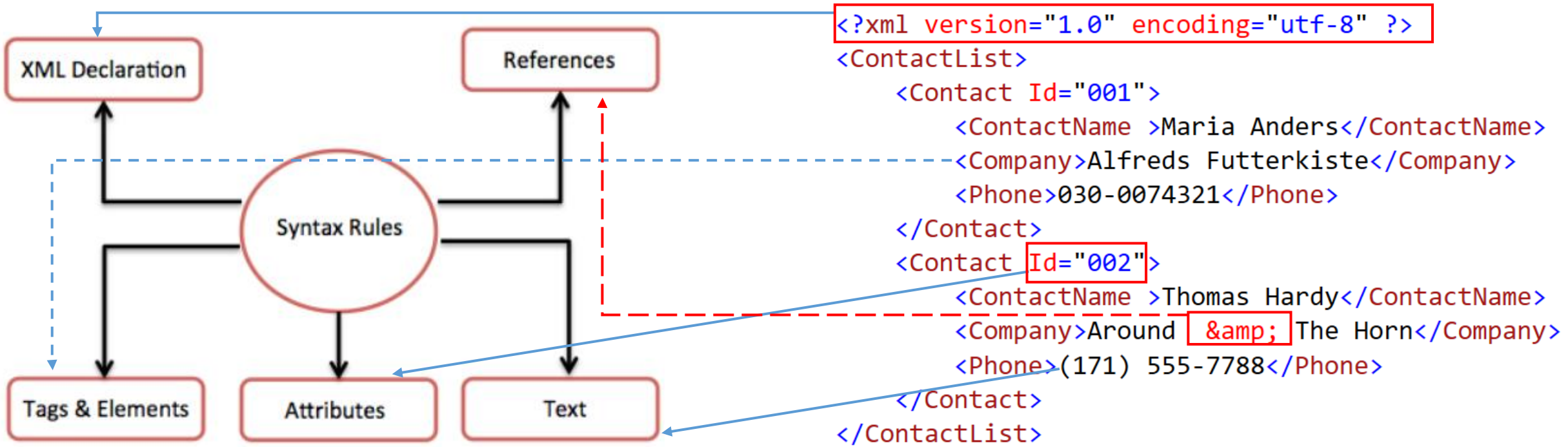
Overview XML Serialization

What is the XML?

- XML stands for Extensible Markup Language. It is a text-based markup language derived from Standard Generalized Markup Language (SGML)
- XML tags identify the data and are used to store and organize the data, rather than specifying how to display it like HTML tags, which are used to display the data. There are three important characteristics of XML that make it useful in a variety of systems and solutions:
  - XML is extensible: XML allows us to create our self-descriptive tags, or language, that suits our the application
  - XML carries the data, does not present it: XML allows us to store the data irrespective of how it will be presented
  - XML is a public standard: XML was developed by an organization called the World Wide Web Consortium (W3C) and is available as an open standard

What is the XML?

- An XML document can have only one root element. The following diagram depicts the syntax rules to write different types of markup and text in an XML document



XmlDataProvider in WPF Application Demonstration

1.Create a WPF app named **ContactListApp**

2.Right-click on the project | Add | New Item, select XML File then rename to **Contacts.xml** , click Add and write contents as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<ContactList>
    <Contact Id="001">
        <ContactName >Maria Anders</ContactName>
        <Company>Alfreds Futterkiste</Company>
        <Phone>030-0074321</Phone>
    </Contact>
        <Contact Id="002">
        <ContactName >Thomas Hardy</ContactName>
        <Company>Around  &amp;amp; The Horn</Company>
        <Phone>(171) 555-7788</Phone>
    </Contact>
        <Contact Id="003">
        <ContactName >Elizabeth Lincoln</ContactName>
        <Company>Bottom-Dollar &amp;amp; Markets</Company>
        <Phone>(604) 555-4729</Phone>
    </Contact>
</ContactList>
```

3.Right-click on the project, select **Edit Project File** and write config information as follows then press **Crtl+S** to save:

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>WinExe</OutputType>
    <TargetFramework>net5.0-windows</TargetFramework>
    <UseWPF>true</UseWPF>
  </PropertyGroup>

  <ItemGroup>
    <Content Include="Contacts.xml">
      <CopyToOutputDirectory>PreserveNewest</CopyToOutputDirectory>
    </Content>
  </ItemGroup>

</Project>
```

4.Write code for **MainWindow.xaml** as follows:

```
<Window x:Class="ContactListApp.MainWindow"
    //xmlns:……
    Title="Contact List" Height="300" SizeToContent="Width" WindowStartupLocation="CenterScreen">
    <Window.Resources>
        <XmlDataProvider Source="Contacts.xml" XPath="ContactList/Contact" x:Key="ContactList"/>
    </Window.Resources>
    <Grid>
        <ListView Name="lvContacts" Margin="31,14,31,16"
                  ItemsSource="{Binding Source={StaticResource ContactList}}">
            <ListView.View>
                <GridView>
                    <GridViewColumn Header="Id" DisplayMemberBinding="{Binding XPath=@Id}"/>
                    <GridViewColumn Header="Contact Name"  Width="100"
                                    DisplayMemberBinding="{Binding XPath=ContactName }"/>
                    <GridViewColumn Header="Company" Width="200"
                                    DisplayMemberBinding="{Binding XPath=Company}"/>
                    <GridViewColumn Header="Phone" Width="150"
                                    DisplayMemberBinding="{Binding XPath=Phone}"/>
                </GridView>
            </ListView.View>
        </ListView>
    </Grid>
</Window>
```

5.Press Crtl+F5 to run project

Understanding XmlSerializer

- The XML serialization engine can produce only XML, and it is less powerful than the binary and data contract serializers in saving and restoring a complex object
- The XML serialization is the process of converting an object's public properties and fields to a serial format (in this case, XML) for storage or transport. Deserialization re-creates the object in its original state from the XML output
- The data in our objects is described using programming language constructs like classes, fields, properties, primitive types, arrays, and even embedded XML in the form of XmlElement or XmlAttribute objects

Understanding XmlSerializer

- The following table describes some of the methods of the XmlSerializer class:

Method Name	Description
CreateReader()	Returns an object used to read the XML document to be serialized
CreateWriter()	When overridden in a derived class, returns a writer used to serialize the object
Deserialize(Stream)	Deserializes the XML document contained by the specified Stream
Deserialize(TextReader)	Deserializes the XML document contained by the specified TextReader
Serialize(Stream, Object)	Serializes the specified Object and writes the XML document to a file using the specified Stream
Serialize(TextWriter, Object)	Serializes the specified Object and writes the XML document to a file using the specified TextWriter

Serializing as XML Demonstration

XmlSerializer Demo-01

1. Create a Console app named **DemoXmlSerializer**
2. Write code for **Program.cs** as follows then press Ctrl+F5 to run project

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;

[XmlRoot("Candidate")]
public class Person {
    [XmlElement("FirstName")]
    public string Name { get; set; }
    [XmlElement("RoughAge")]
    public int Age { get; set; }
}//end class
class Program{
    static void Main(string[] args){
        Person p1 = new Person() { Name="David", Age=30 };
        var xs = new XmlSerializer(typeof(Person));
        //Serialize
        using Stream s1 = File.Create("person.xml");
        xs.Serialize(s1, p1);
        s1.Close();

        //Deserialize
        using Stream s2 = File.OpenRead("person.xml");
        var p2 = (Person)xs.Deserialize(s2);
        Console.WriteLine("****Person Info****");
        Console.WriteLine($"Name: {p2.Name}, Age: {p2.Age}");

        Console.WriteLine("****Person.xml****");
        string xmlData = File.ReadAllText("person.xml");
        Console.WriteLine(xmlData);
        s2.Close();
        Console.ReadLine();
    }//end Main
}//end Program
```

```
****Person Info****
Name: David, Age: 30
****Person.xml****
<?xml version="1.0"?>
<Candidate xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <FirstName>David</FirstName>
  <RoughAge>30</RoughAge>
</Candidate>
```

XmlSerializer Demo-02

1. Create a Console app named **DemoXmlSerializer02**
2. Right-click on the project , select Add | Class named **Person.cs** then write codes as follows:

```
//...
using System.Xml.Serialization;
namespace Demo_XMLSerialization_02 {
    public class Person{
        public Person() { }
        public Person(decimal initialSalary) => Salary = initialSalary;
        public string FirstName { get; set; }
        public string LastName { get; set; }
        public DateTime DateOfBirth { get; set; }
        public HashSet<Person> Children { get; set; }
        //Salary property is not included
        protected decimal Salary { get; set; }
    }
}
```

3. Write code for **Program.cs** as follows then press Ctrl+F5 to run project

```
//...
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;
namespace Demo_XMLSerialization_02 {
   class Program{
       static void Main(string[] args){
           string fileName = "people.xml";
           // Create an object graph
           var people = new List<Person> {
               new Person(30000M) { FirstName = "Alice",
                   LastName = "Smith", DateOfBirth = new DateTime(1972, 3, 16) },
               new Person(20000M) { FirstName = "Charlie", LastName = "Cox",
                   DateOfBirth = new DateTime(1984, 5, 4),
                   Children = new HashSet<Person>
                   { new Person(0M) { FirstName = "Sally", LastName = "Cox",
                   DateOfBirth = new DateTime(2000, 7, 12) } } }
           };
           // Create object that will format a List of Persons as XML
           var xs = new XmlSerializer(typeof(List<Person>));
           // create a file to write to
           using FileStream stream = File.Create(fileName);
           // serialize the object graph to the stream
           xs.Serialize(stream, people);
           Console.WriteLine("Written {0:N0} bytes of XML to {1}",
               new FileInfo(fileName).Length,fileName);
           stream.Close();
```

```
            Console.WriteLine(new string('*', 30));
            // Display the serialized object graph
            Console.WriteLine(File.ReadAllText(fileName));
            Console.WriteLine(new string('*',30));
            // Deserializing with XML
            using FileStream xmlLoad = File.Open(fileName, FileMode.Open);
            // Deserialize and cast the object graph into a List of Person
            var loadedPeople = (List<Person>)xs.Deserialize(xmlLoad);
            foreach (var item in loadedPeople) {
                Console.WriteLine($"{item.LastName} has " +
                    $"{item.Children?.Count ?? 0} children.");
            }
            xmlLoad.Close();
            Console.ReadLine();
        }//end Main
    }//end Program
}
```

```
Written 635 bytes of XML to people.xml
******************************
<?xml version="1.0"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSch
ema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSche
ma">
  <Person>
    <FirstName>Alice</FirstName>
    <LastName>Smith</LastName>
    <DateOfBirth>1972-03-16T00:00:00</DateOfBirth>
  </Person>
  <Person>
    <FirstName>Charlie</FirstName>
    <LastName>Cox</LastName>
    <DateOfBirth>1984-05-04T00:00:00</DateOfBirth>
    <Children>
      <Person>
        <FirstName>Sally</FirstName>
        <LastName>Cox</LastName>
        <DateOfBirth>2000-07-12T00:00:00</DateOfBirth>
      </Person>
    </Children>
  </Person>
</ArrayOfPerson>
******************************
Smith has 0 children.
Cox has 1 children.
```

Overview JSON Serialization

What is the JSON?

- JSON stands for JavaScript Object Notation
- JSON is a 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, just like JavaScript object properties. A name/value pair consists of a field name (in double quotes), followed by a colon, followed by a value:

```
{
  "firstName":"John", "lastName":"Doe"
}
```

JSON Syntax Rules

- Data is in name/value pairs
- Data is separated by commas
- Curly braces hold objects
- Square brackets hold arrays

```
{
 "employees":
   [
     {"firstName":"John", "lastName":"Doe"},
     {"firstName":"Anna", "lastName":"Smith"},
     {"firstName":"Peter", "lastName":"Jones"}
   ]
}
```

JSON Data Types

- JSON supports mainly 06 data types: String, Number, Boolean, null, Object, and Array
- **String**: JSON strings must be written in double quotes like C-language there are various special characters(Escape Characters) in JSON that you can use in strings such as \ (backslash), / (forward slash), b (backspace), n (new line), r (carriage return), t (horizontal tab), etc

Example:

{ "name":"Vivek" }

{ "city":"Delhi\/India" }

here \ / is used for Escape Character / (forward slash)

JSON Data Types

- **Number**: Represented in base 10. The octal and hexadecimal formats are not used

Example: { "age": 20 } , { "percentage": 82.44}

- **Boolean**: This data type can be either true or false

Example: { "result" : true }

- **Null:** It is just a define null value

Example: { "middlename":null }

JSON Data Types

- **Object:** It is a set of name or value pairs inserted between {} (curly braces). The keys must be strings and should be unique and multiple key and value pairs are separated by a, (comma)

Syntax:

{ key : value, ...}

Example:

```
{
  “student":{ "name":“David", "age":20, "score": 50.05}
}
```

JSON Data Types

- **Array:** It is an ordered collection of values and begins with [(left bracket) and ends with] (right bracket). The values of array are separated by ,(comma) Syntax:

  [value, ...]

Example:

```
{
  "collection" :  [
        {"id" : 101},
        {"id" : 102},
        {"id" : 103}
   ]
}
```

Understanding JSON Serialization

- The JSON (JavaScript Object Notation) serializer is fast and efficient, and was introduced relatively recently to .NET (.NET Core). It also offers good version tolerance and allows the use of custom converters for flexibility
- JsonSerializer is used by ASP.NET Core 3, removing the dependency on Json.NET, though it is straightforward to opt back in to Json.NET should its features be required
- JsonSerializer (in the System.Text.Json namespace) is straightforward to use because of the simplicity of the JSON format. The root of a JSON document is either an array or an object. Under that root are properties, which can be an object, array, string, number, "true", "false", or "null"
- The JSON serializer directly maps class property names to property names in JSON

Understanding JSON Serialization

- The following table describes some of the methods of the JsonSerializer class:

Method Name	Description
Deserialize(String, Type, JsonSerializerOptions)	Parses the text representing a single JSON value into an instance of a specified type
Deserialize(Utf8JsonReader, Type, JsonSerializerOptions)	Reads one JSON value (including objects or arrays) from the provided reader and converts it into an instance of a specified type
Deserialize<TValue>(String, JsonSerializerOptions)	Parses the text representing a single JSON value into an instance of the type specified by a generic type parameter.
Serialize(Object, Type, JsonSerializerOptions)	Converts the value of a specified type into a JSON string
Serialize(Utf8JsonWriter, Object, Type, JsonSerializerOptions)	Writes the JSON representation of the specified type to the provided writer
SerializeToUtf8Bytes(Object, Type, JsonSerializerOptions)	Converts a value of the specified type into a JSON string, encoded as UTF-8 bytes
Serialize<TValue>(TValue, JsonSerializerOptions)	Converts the value of a type specified by a generic type parameter into a JSON string

Controlling Serialization with Attributes

- We can control the serialization process with attributes defined in the System.Text.Json.Serialization namespace
- The following subsections present the most useful attributes:

Attribute Name	Description
JsonIgnoreAttribute	Prevents a property from being serialized or deserialized
JsonPropertyNameAttribute	Specifies the property name that is present in the JSON when serializing and deserializing. This overrides any naming policy specified by JsonNamingPolicy
JsonExtensionDataAttribute	When placed on a property of type IDictionary<TKey,TValue>, any properties that do not have a matching member are added to that dictionary during deserialization and written during serialization
JsonConverterAttribute	Converts an object or value to or from JSON
JsonIncludeAttribute	Indicates that the member should be included for serialization and deserialization
JsonNumberHandlingAttribute	When placed on a type, property, or field, indicates what JsonNumberHandling settings should be used when serializing or deserializing numbers

Controlling Serialization with Attributes Demo

```
//…
using System.Text.Json;
using System.Text.Json.Serialization;

public class Employee {
    public Employee(){ }
    public Employee(decimal initialSalary) {
        Salary = initialSalary;
    }
    [JsonPropertyName("FullName")]
    public string Name { get; set; }
    [JsonIgnore]
    public string Email { get; set; }
    [JsonInclude]
    public decimal Salary;
}//end Employee
```

```
****Serialize*****
{
  "FullName": "Jack",
  "Salary": 1000
}
****Deserialize*****
Name:Jack, Salary:1000
```

```
//----------------------------------------
class Program
{
    static void Main(string[] args)
    {
        var emp1 = new Employee(1000M);
        emp1.Name = "Jack";
        emp1.Email = "jack@gmail.com";
        var option = new JsonSerializerOptions { WriteIndented = true };
        Console.WriteLine("****Serialize*****");
        string jsonData = JsonSerializer.Serialize(emp1, option);
        Console.WriteLine(jsonData);
        Console.WriteLine("****Deserialize*****");
        var emp2 = JsonSerializer.Deserialize<Employee>(jsonData);
        Console.WriteLine($"Name:{emp2.Name}, Salary:{emp2.Salary}");
        Console.ReadLine();
    }
}
```

JSON Serialization Options

- The serializer accepts an optional JsonSerializationOptions parameter, allowing additional control over the serialization and deserialization process
- The following subsections present the most useful options:

Property Name	Description
WriteIndented	Gets or sets a value that defines whether JSON should use pretty printing. By default, JSON is serialized without any extra white space
AllowTrailingCommas	Get or sets a value that indicates whether an extra comma at the end of a list of JSON values in an object or array is allowed (and ignored) within the JSON payload being deserialized
ReadCommentHandling	Gets or sets a value that defines how comments are handled during deserialization
PropertyNameCaseInsensitive	Gets or sets a value that determines whether a property's name uses a case-insensitive comparison during deserialization. The default value is false
ReferenceHandler	Configures how object references are handled when reading and writing JSON

JSON Serialization Options

Property Name	Description
PropertyNamingPolicy	Gets or sets a value that specifies the policy used to convert a property's name on an object to another format, such as camel-casing, or null to leave property names unchanged
DictionaryKeyPolicy	Gets or sets the policy used to convert a IDictionary key's name to another format, such as camel-casing
Encoder	Gets or sets the encoder to use when escaping strings, or null to use the default encoder
IgnoreNullValues	ets or sets a value that determines whether null values are ignored during serialization and deserialization. The default value is false.
IgnoreReadOnlyProperties	Determines whether read-only fields are ignored during serialization. A field is read-only if it is marked with the readonly keyword. The default value is false
MaxDepth	Gets or sets the maximum depth allowed when serializing or deserializing JSON, with the default value of 0 indicating a maximum depth of 64

JSON Serialization Options Demo

```
//…
using System.Text.Json;
using System.Text.Json.Serialization;

public class Employee{
    [JsonPropertyName("FullName")]
    public string Name { get; set; }
    public string Email { get; set; }
    public decimal Salary { get; set; }
}//end Employee

class Program{
    static void Main(string[] args){
        string json = @"{
            ""FullName"":""Mark"", // FullName
            ""Email"":""Mark@gmail.com"", // Email
             ""Salary"":1000, // Salary
        }";
        var option = new JsonSerializerOptions(){
            WriteIndented = true,
            ReadCommentHandling = JsonCommentHandling.Skip,
            AllowTrailingCommas = true
        };
        var emp = JsonSerializer.Deserialize<Employee>(json, option);
        Console.WriteLine($"Name:{emp.Name}, Email:{emp.Email}," +
            $" Salary:{emp.Salary}");
        Console.ReadLine();
    }
}//end Program
```

```
Name:Mark, Email:Mark@gmail.com, Salary:1000
```

Json Serialization Behavior

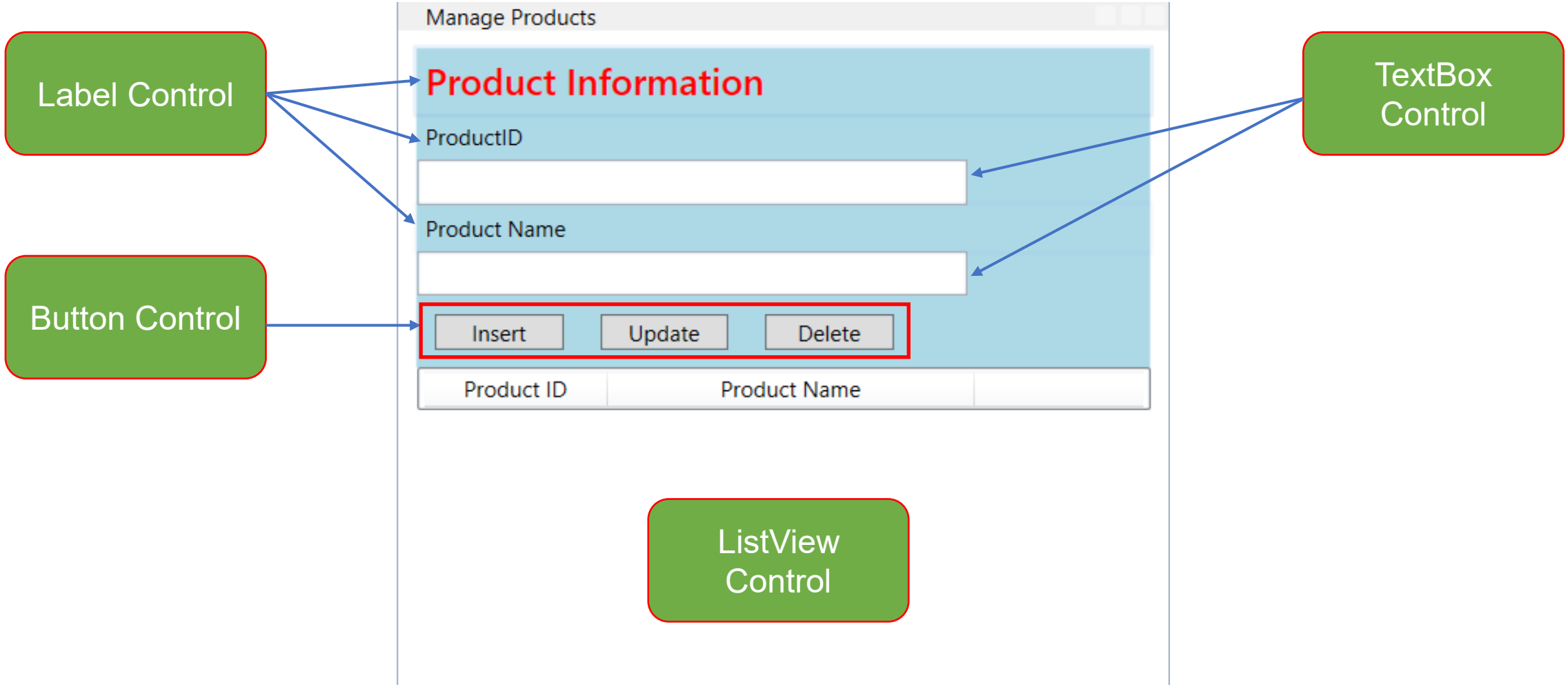
- By default, all public properties are serialized. To ignore individual properties, use the [JsonIgnore] attribute
- The default encoder escapes non-ASCII characters, HTML-sensitive characters within the ASCII-range, and characters that must be escaped according to the RFC 8259 JSON spec
- By default, JSON is minified. To pretty-print the JSON output, set JsonSerializerOptions.WriteIndented to true
- By default, casing of JSON names matches the .NET names. To set the name of individual properties, we can use the [JsonPropertyName] attribute
- By default, fields are ignored (use [JsonInclude] attribute to include fields)

Json Deserialization Behavior

- By default, property name matching is case-sensitive
- If the JSON contains a value for a read-only property, the value is ignored and no exception is thrown. Non-public constructors are ignored by the serializer
- Deserialization to immutable objects or read-only properties is supported
- By default, enums are supported as numbers. We can serialize enum names as strings
- By default, fields are ignored. Use the [JsonInclude] attribute to include fields
- The default maximum depth is 64
- By default, comments or trailing commas in the JSON throw exceptions. To allow comments in the JSON, we can set the JsonSerializerOptions .ReadCommentHandling property to JsonCommentHandling.Skip

Serializing as Json Demonstration

1.Create a WPF app named **ManageProductsApp** includes a window named **WindowManageProducts.xaml** that has controls as follows :



XAML code of **WindowManageProducts.xaml**:

```
<Window x:Class="ManageProductsApp.WindowManageProducts"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:Demo_JSON_Serialization"
        mc:Ignorable="d"
        Title="Manage Products" Height="430" Width="420"
        Loaded="Window_Loaded" WindowStartupLocation="CenterScreen"
        ResizeMode="NoResize">
    <DockPanel VerticalAlignment="Top" Margin="10">
        <Grid>
        </Grid>
    </DockPanel>
</Window>
```

View details in next slide

XAML code of **Grid** tag

```
<Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="*"/>
            <RowDefinition Height="*"/>
        </Grid.RowDefinitions>
        <StackPanel Background="LightBlue"  Orientation ="Vertical"
                    HorizontalAlignment="Left" Width="400">

            <Label Name="lblInstruction" Foreground="Red" FontWeight="DemiBold"
               FontSize="20" Content="Product Information"/>

            <Label Name="lblProductID" Content="ProductID"/>
            <TextBox Name="txtProductID" HorizontalAlignment="Left"
                 Height="25" Width="300"
                 Text="{Binding Path=ProductID, Mode=OneWay}"
                 DataContext="{Binding ElementName=lvProducts, Path=SelectedItem}" />

            <Label Name="lbProductName" Content="Product Name" />
            <TextBox Name="txtProductName" HorizontalAlignment="Left"
                 Height="25" Width="300"  Text="{Binding Path=ProductName, Mode=OneWay}"
                 DataContext="{Binding ElementName=lvProducts, Path=SelectedItem}" />
```

XAML code of **Grid** tag (cont.)

```
<StackPanel Orientation="Horizontal"  HorizontalAlignment="Left">
        <Button x:Name="btnInsert" Margin="10" Width="70" Content="Insert"
            Click="btnInsert_Click" />
        <Button x:Name="btnUpdate"  Margin="10" Width="70"  Content="Update"
            Click="btnUpdate_Click"/>
        <Button x:Name="btnDelete" Margin="10"  Width="70" Content="Delete"
            Click="btnDelete_Click"/>
    </StackPanel>
</StackPanel>
<ListView Grid.Row="1" Name="lvProducts" Width="400" >
    <ListView.View>
        <GridView>
            <GridViewColumn Header="Product ID"  Width="100"
                        DisplayMemberBinding="{Binding Path=ProductID }"/>
            <GridViewColumn Header="Product Name" Width="200"
                        DisplayMemberBinding="{Binding Path=ProductName}"/>
        </GridView>
    </ListView.View>
</ListView>
</Grid>
```

2.Right-click on the project | **Add** | **Class**, named **ManageProducts.cs** then write codes as follows:

```
//...
using System.IO;
using System.Text.Json;

//Declaring record Product
public record Product {
    public int ProductID { get; set; }
    public string ProductName { get; set; }
}
//----------------------------------------
public class ManageProducts{
    string fileName = "ProductList.json";
    List<Product> products = new List<Product>();
    public List<Product> GetProducts()        {
        GetDataFromFile();
        return products;
    }
    //----------------------------------------
```

```
public void StoreToFile(){
    try{
        // Serialize the object graph into a string
        string jsonData = JsonSerializer.Serialize(products,
            new JsonSerializerOptions { WriteIndented = true });
        File.WriteAllText(fileName, jsonData);
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
}
//----------------------------------------
public void GetDataFromFile() {
    try{
        if (File.Exists(fileName)){
            string jsonData = File.ReadAllText(fileName);
            // Deserialize object graph into a List of Product
            products = JsonSerializer.Deserialize<List<Product>>(jsonData);
        }
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
}
//----------------------------------------
```

```
public void InsertProduct(Product Product){
    try{
        Product p = products.SingleOrDefault(p => p.ProductID == Product.ProductID);
        if (p != null){
            throw new Exception("This product already exists.");
        }
        products.Add(Product);
        StoreToFile();
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }
}//end InsertProduct
```

```
public void UpdateProduct(Product Product){
    try{
        Product p = products.SingleOrDefault(p => p.ProductID == Product.ProductID);
        if (p == null){
            throw new Exception("This product did not exist.");
        }
        else{
            p.ProductName = Product.ProductName;
            StoreToFile();
        }
    }
    catch (Exception ex){
        throw new Exception(ex.Message);
    }

}//end UpdateProduct
//----------------------------------------
```

```
    //----------------------------------------
    public void DeleteProduct(Product Product){
        try{
            Product p = products.SingleOrDefault(p => p.ProductID == Product.ProductID);
            if (p == null){
                throw new Exception("This product did not exist.");
            }
            else{
                products.Remove(p);
                StoreToFile();
            }
        }
        catch (Exception ex){
            throw new Exception(ex.Message);
        }
    }//end DeleteProduct
}//end ManageProducts
```

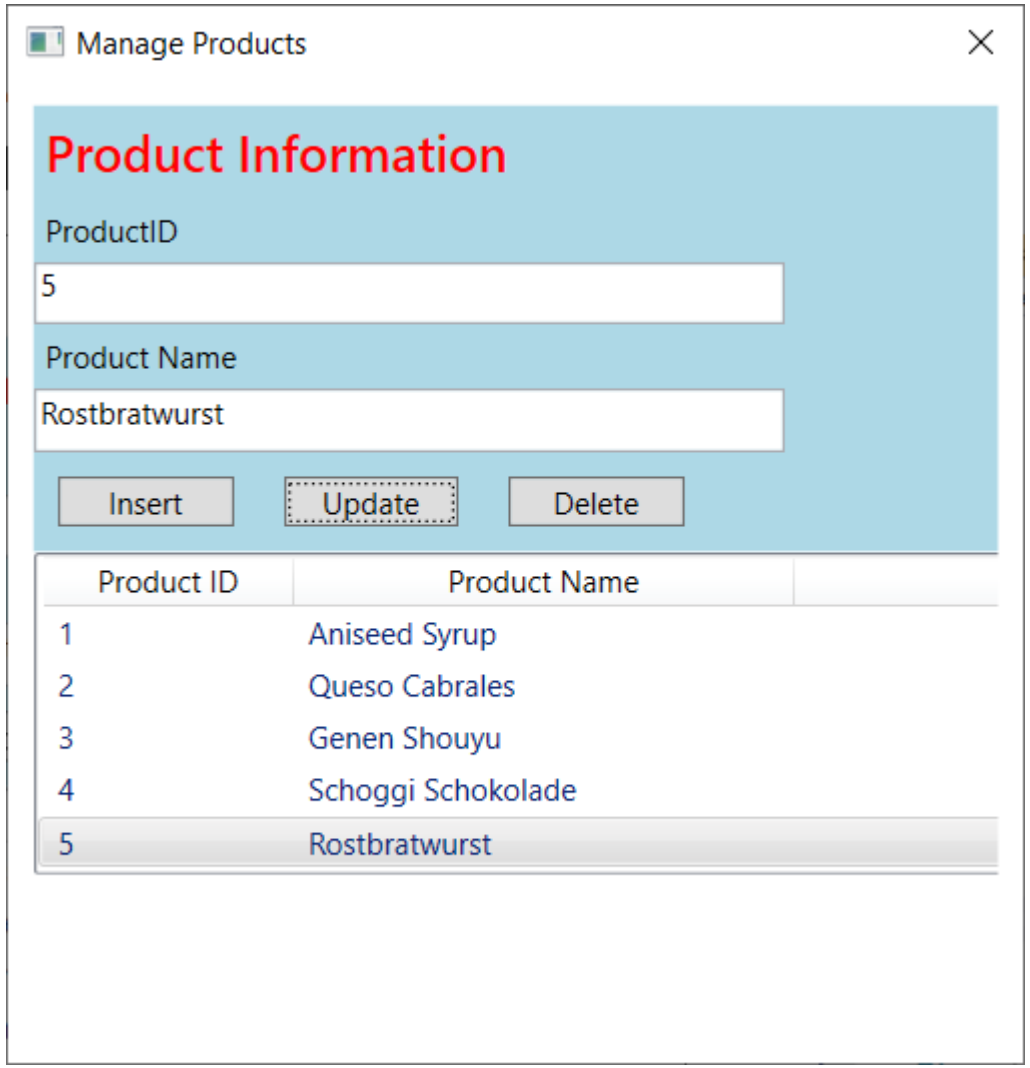
3.Write codes in **WindowManageProducts.xaml.cs** as follows

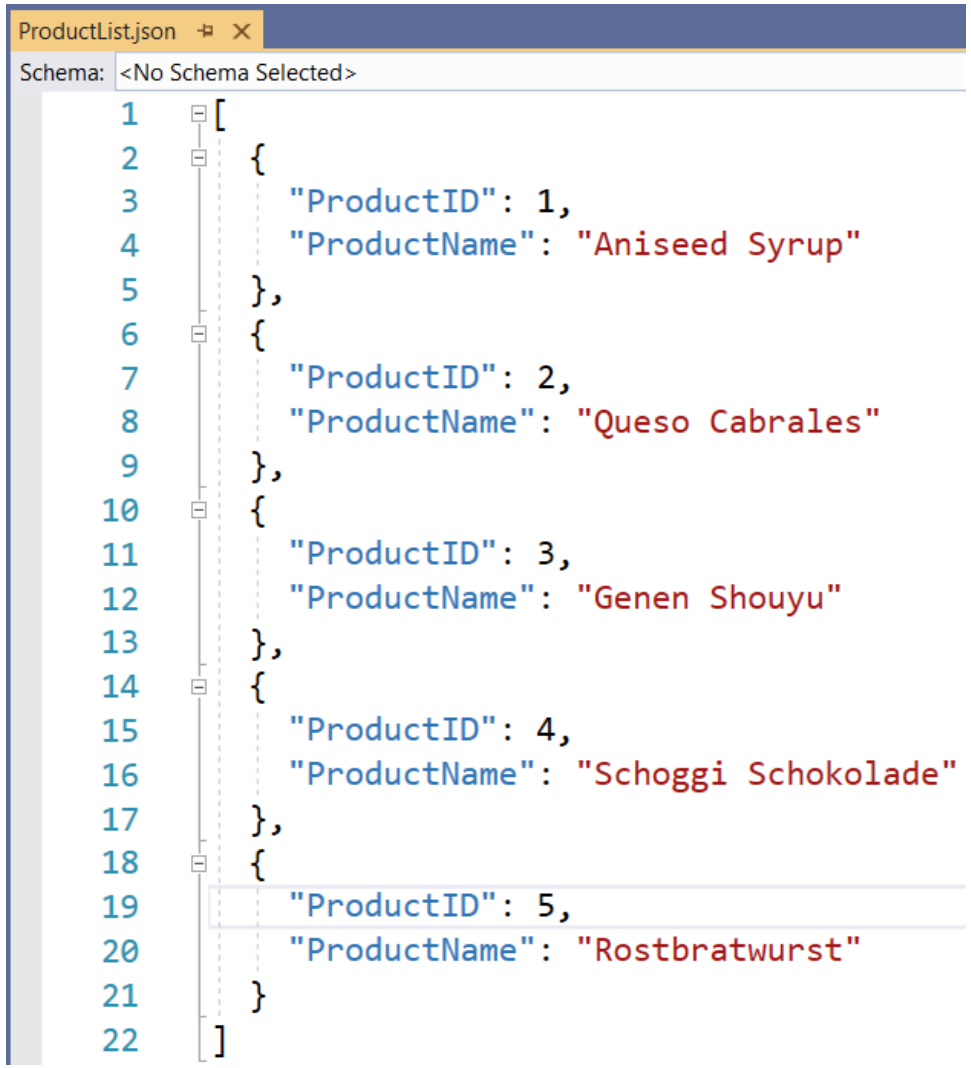
```
public partial class WindowManageProducts : Window
{
    public WindowManageProducts()...

//-----------------------------------------------------------------------
ManageProducts products = new ManageProducts();
private void Window_Loaded(object sender, RoutedEventArgs e) => LoadProducts();
//-----------------------------------------------------------------------
private void btnInsert_Click(object sender, RoutedEventArgs e){
    try{
        var Product = new Product {
            ProductID=int.Parse(txtProductID.Text),
            ProductName = txtProductName.Text
        };
        products.InsertProduct(Product);
        LoadProducts();
    }
    catch (Exception ex){
        MessageBox.Show(ex.Message, "Insert Product");
    }
}
//-----------------------------------------------------------------------
```

```
private void btnUpdate_Click(object sender, RoutedEventArgs e){
    try {
        var Product = new Product {
            ProductID = int.Parse(txtProductID.Text),
            ProductName = txtProductName.Text
        };
        products.UpdateProduct(Product);
        LoadProducts();
    }
    catch (Exception ex){
        MessageBox.Show(ex.Message, "Update Product");
    }
}
//-------------------------------------------------------------------------------
private void btnDelete_Click(object sender, RoutedEventArgs e) {
    try{
        var Product = new Product{
            ProductID = int.Parse(txtProductID.Text)
        };
        products.DeleteProduct(Product);
        LoadProducts();
    }
    catch (Exception ex){
        MessageBox.Show(ex.Message, "Delete Product");
    }
}
}//End Class
```

4. Press **Ctrl+F5** to run project and view the output





Summary

- Concepts were introduced:
  - Overview Serialization in .NET
  - Understanding Serialization Engines in .NET
  - Explain about how serialization works
  - Describe use Serialization
  - Overview XML Serialization
  - Overview JSON Serialization
  - Create demo using XML with WPF application
  - Create demo XML Serialization in .NET application
  - Create demo JSON Serialization in .NET application