Contents

Serialization (C#)

- How to: Write Object Data to an XML File
- How to: Read Object Data from an XML File
- Walkthrough: Persisting an Object in Visual Studio

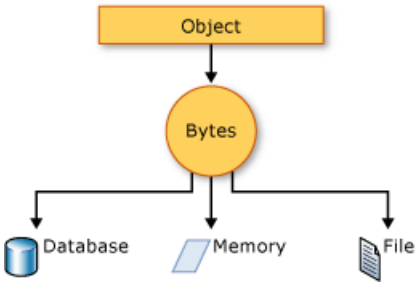
Serialization (C#)

9/5/2018 • 3 minutes to read • Edit Online

Serialization is the process of converting an object into a stream of bytes to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called deserialization.

How serialization works

This illustration shows the overall process of serialization.



The object is serialized to a stream, which carries not just the data, but information about the object's type, such as its version, culture, and assembly name. From that stream, it 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 recreate it as needed, providing storage of objects as well as data exchange. Through serialization, a developer can perform actions like sending the object to a remote application by means of a Web Service, passing an object from one domain to another, passing an object through a firewall as an XML string, or maintaining security or user-specific information across applications.

Making an object serializable

To serialize an object, you need the object to be serialized, a stream to contain the serialized object, and a Formatter. System.Runtime.Serialization contains the classes necessary for serializing and deserializing objects.

Apply the SerializableAttribute attribute to a type to indicate that instances of this type can be serialized. An exception is thrown if you attempt to serialize but the type doesn't have the SerializableAttribute attribute.

If you don't want a field within your class to be serializable, apply the NonSerializedAttribute attribute. If a field of a serializable type contains a pointer, a handle, or some other data structure that is specific to a particular environment, and the field cannot be meaningfully reconstituted in a different environment, then you may want to make it nonserializable.

If a serialized class contains references to objects of other classes that are marked SerializableAttribute, those objects will also be serialized.

Binary and XML serialization

You can use binary or XML serialization. In binary serialization, all members, even members that are read-only, are serialized, and performance is enhanced. XML serialization provides more readable code, and greater flexibility of object sharing and usage for interoperability purposes.

Binary serialization

Binary serialization uses binary encoding to produce compact serialization for uses such as storage or socket-

based network streams.

XML serialization

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

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

Basic and custom serialization

Serialization can be performed in two ways, basic and custom. Basic serialization uses the .NET Framework to automatically serialize the object.

Basic serialization

The only requirement in basic serialization is that the object has the SerializableAttribute attribute applied. The NonSerializedAttribute can be used to keep specific fields from being serialized.

When you use basic serialization, the versioning of objects may create problems. You would use custom serialization when versioning issues are important. Basic serialization is the easiest way to perform serialization, but it does not provide much control over the process.

Custom serialization

In custom serialization, you can specify exactly which objects will be serialized and how it will be done. The class must be marked SerializableAttribute and implement the ISerializable interface.

If you want your object to be deserialized in a custom manner as well, you must use a custom constructor.

Designer serialization

Designer serialization is a special form of serialization that involves the kind of object persistence associated with development tools. Designer serialization is the process of converting an object graph into a source file that can later be used to recover the object graph. A source file can contain code, markup, or even SQL table information.

Related Topics and Examples

Walkthrough: Persisting an Object in Visual Studio (C#)
Demonstrates how serialization can be used to persist an object's data between instances, allowing you to store values and retrieve them the next time the object is instantiated.

How to: Read Object Data from an XML File (C#)
Shows how to read object data that was previously written to an XML file using the XmlSerializer class.

How to: Write Object Data to an XML File (C#)
Shows how to write the object from a class to an XML file using the XmlSerializer class.

How to: Write Object Data to an XML File (C#)

1/23/2019 • 2 minutes to read • Edit Online

This example writes the object from a class to an XML file using the XmlSerializer class.

Example

```
public class XMLWrite
{

   static void Main(string[] args)
    {
        WriteXML();
    }

    public class Book
    {
        public String title;
    }

    public static void WriteXML()
    {
        Book overview = new Book();
        overview.title = "Serialization Overview";
        System.Xml.Serialization.XmlSerializer writer =
            new System.Xml.Serialization.XmlSerializer(typeof(Book));

        var path = Environment.GetFolderPath(Environment.SpecialFolder.MyDocuments) +
"//SerializationOverview.xml";
        System.IO.FileStream file = System.IO.File.Create(path);

        writer.Serialize(file, overview);
        file.Close();
    }
}
```

Compiling the Code

The class must have a public constructor without parameters.

Robust Programming

The following conditions may cause an exception:

- The class being serialized does not have a public, parameterless constructor.
- The file exists and is read-only (IOException).
- The path is too long (PathTooLongException).
- The disk is full (IOException).

.NET Framework Security

This example creates a new file, if the file does not already exist. If an application needs to create a file, that application needs `Create` access for the folder. If the file already exists, the application needs only `Write` access, a

lesser privilege. Where possible, it is more secure to create the file during deployment, and only grant `Read` access to a single file, rather than `Create` access for a folder.

See also

- StreamWriter
- How to: Read Object Data from an XML File (C#)
- Serialization (C#)

How to: Read Object Data from an XML File (C#)

1/23/2019 • 2 minutes to read • Edit Online

This example reads object data that was previously written to an XML file using the XmlSerializer class.

Example

```
public class Book
{
    public String title;
}

public void ReadXML()
{
    // First write something so that there is something to read ...
    var b = new Book { title = "Serialization Overview" };
    var writer = new System.Xml.Serialization.XmlSerializer(typeof(Book));
    var wfile = new System.IO.StreamWriter(@"c:\temp\SerializationOverview.xml");
    writer.Serialize(wfile, b);
    wfile.Close();

    // Now we can read the serialized book ...
    System.Xml.Serialization.XmlSerializer reader =
        new System.Xml.Serialization.XmlSerializer(typeof(Book));
    System.IO.StreamReader file = new System.IO.StreamReader(
        @"c:\temp\SerializationOverview.xml");
    Book overview =  (Book)reader.Deserialize(file);
    file.Close();

    Console.WriteLine(overview.title);

}
```

Compiling the Code

Replace the file name "c:\temp\SerializationOverview.xml" with the name of the file containing the serialized data. For more information about serializing data, see How to: Write Object Data to an XML File (C#).

The class must have a public constructor without parameters.

Only public properties and fields are deserialized.

Robust Programming

The following conditions may cause an exception:

- The class being serialized does not have a public, parameterless constructor.
- The data in the file does not represent data from the class to be deserialized.
- The file does not exist (IOException).

.NET Framework Security

Always verify inputs, and never deserialize data from an untrusted source. The re-created object runs on a local computer with the permissions of the code that deserialized it. Verify all inputs before using the data in your

application.

See also

- StreamWriter
- How to: Write Object Data to an XML File (C#)
- Serialization (C#)
- C# Programming Guide

Walkthrough: persisting an object using C#

1/23/2019 • 4 minutes to read • Edit Online

You can use serialization to persist an object's data between instances, which enables you to store values and retrieve them the next time that the object is instantiated.

In this walkthrough, you will create a basic `Loan` object and persist its data to a file. You will then retrieve the data from the file when you re-create the object.

> **IMPORTANT**
>
> This example creates a new file if the file does not already exist. If an application must create a file, that application must have `Create` permission for the folder. Permissions are set by using access control lists. If the file already exists, the application needs only `Write` permission, a lesser permission. Where possible, it's more secure to create the file during deployment and only grant `Read` permissions to a single file (instead of Create permissions for a folder). Also, it's more secure to write data to user folders than to the root folder or the Program Files folder.

> **IMPORTANT**
>
> This example stores data in a binary format file. These formats should not be used for sensitive data, such as passwords or credit-card information.

Prerequisites

- To build and run, install the .NET Core SDK.
- Install your favorite code editor, if you haven't already.

> **TIP**
>
> Need to install a code editor? Try Visual Studio!

- The example requires C# 7.3. See Select the C# language version

You can examine the sample code online at the .NET samples GitHub repository.

Creating the loan object

The first step is to create a `Loan` class and a console application that uses the class:

1. Create a new application. Type `dotnet new console -o serialization` to create a new console application in a subdirectory named `serialization`.
2. Open the application in your editor, and add a new class named `Loan.cs`.
3. Add the following code to your `Loan` class:

```
public class Loan : INotifyPropertyChanged
 {
     public double LoanAmount { get; set; }
     public double InterestRate { get; set; }

     [field:NonSerialized()]
     public DateTime TimeLastLoaded { get; set; }

     public int Term { get; set; }

     private string customer;
     public string Customer
     {
         get { return customer; }
         set
         {
             customer = value;
             PropertyChanged?.Invoke(this,
               new PropertyChangedEventArgs(nameof(Customer)));
         }
     }

     [field: NonSerialized()]
     public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;

     public Loan(double loanAmount,
                 double interestRate,
                 int term,
                 string customer)
     {
         this.LoanAmount = loanAmount;
         this.InterestRate = interestRate;
         this.Term = term;
         this.customer = customer;
     }

 }
```

You will also have to create an application that uses the `Loan` class.

Serialize the loan object

1. Open `Program.cs`. Add the following code:

```
Loan TestLoan = new Loan(10000.0, 0.075, 36, "Neil Black");
```

Add an event handler for the `PropertyChanged` event, and a few lines to modify the `Loan` object and display the changes. You can see the additions in the following code:

```
TestLoan.PropertyChanged += (_, __) => Console.WriteLine($"New customer value: {TestLoan.Customer}");

TestLoan.Customer = "Henry Clay";
Console.WriteLine(TestLoan.InterestRate);
TestLoan.InterestRate = 7.1;
Console.WriteLine(TestLoan.InterestRate);
```

At this point, you can run the code, and see the current output:

```
New customer value: Henry Clay
7.5
7.1
```

Running this application repeatedly always writes the same values. A new Loan object is created every time you run the program. In the real world, interest rates change periodically, but not necessarily every time that the application is run. Serialization code means you preserve the most recent interest rate between instances of the application. In the next step, you will do just that by adding serialization to the Loan class.

Using Serialization to Persist the Object

In order to persist the values for the Loan class, you must first mark the class with the `Serializable` attribute. Add the following code above the Loan class definition:

```
[Serializable()]
```

The SerializableAttribute tells the compiler that everything in the class can be persisted to a file. Because the `PropertyChanged` event does not represent part of the object graph that should be stored, it should not be serialized. Doing so would serialize all objects that are attached to that event. You can add the NonSerializedAttribute to the field declaration for the `PropertyChanged` event handler.

```
[field: NonSerialized()]
public event System.ComponentModel.PropertyChangedEventHandler PropertyChanged;
```

Beginning with C# 7.3, you can attach attributes to the backing field of an auto-implemented property using the `field` target value. The following code adds a `TimeLastLoaded` property and marks it as not serializable:

```
[field:NonSerialized()]
public DateTime TimeLastLoaded { get; set; }
```

The next step is to add the serialization code to the LoanApp application. In order to serialize the class and write it to a file, you use the System.IO and System.Runtime.Serialization.Formatters.Binary namespaces. To avoid typing the fully qualified names, you can add references to the necessary namespaces as shown in the following code:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

The next step is to add code to deserialize the object from the file when the object is created. Add a constant to the class for the serialized data's file name as shown in the following code:

```
const string FileName = @"../../../SavedLoan.bin";
```

Next, add the following code after the line that creates the `TestLoan` object:

```
if (File.Exists(FileName))
{
    Console.WriteLine("Reading saved file");
    Stream openFileStream = File.OpenRead(FileName);
    BinaryFormatter deserializer = new BinaryFormatter();
    TestLoan = (Loan)deserializer.Deserialize(openFileStream);
    TestLoan.TimeLastLoaded = DateTime.Now;
    openFileStream.Close();
}
```

You first must check that the file exists. If it exists, create a Stream class to read the binary file and a BinaryFormatter class to translate the file. You also need to convert from the stream type to the Loan object type.

Next you must add code to serialize the class to a file. Add the following code after the existing code in the `Main` method:

```
Stream SaveFileStream = File.Create(FileName);
BinaryFormatter serializer = new BinaryFormatter();
serializer.Serialize(SaveFileStream, TestLoan);
SaveFileStream.Close();
```

At this point, you can again build and run the application. The first time it runs, notice that the interest rates starts at 7.5, and then changes to 7.1. Close the application and then run it again. Now, the application prints the message that it has read the saved file, and the interest rate is 7.1 even before the code that changes it.

See also

- Serialization (C#)
- C# Programming Guide