

Атрибуты



1. **Понятие атрибутов.**
2. **Что такое сериализация?**
3. **Отношения между объектами.**
4. **Графы отношений объектов.**
5. **Атрибуты для сериализации [Serializable] и [NonSerialized].**
6. **Форматы сериализации.**
 - a. **Пространство
System.Runtime.Serialization.Formatters.**
 - b. **Двоичное форматирование. Класс
BinaryFormatter.**
 - c. **Примеры использования сериализации.**

Атрибуты представляют собой аннотации программного кода, которые могут применяться к заданному типу (классу, интерфейсу, структуре и т.д.), а так же к членам типа (полям, методам, свойствам).

Основное назначение возможность определять *дополнительную функциональность (добавлять специальную информацию) к элементам кода* через *метаданные*.

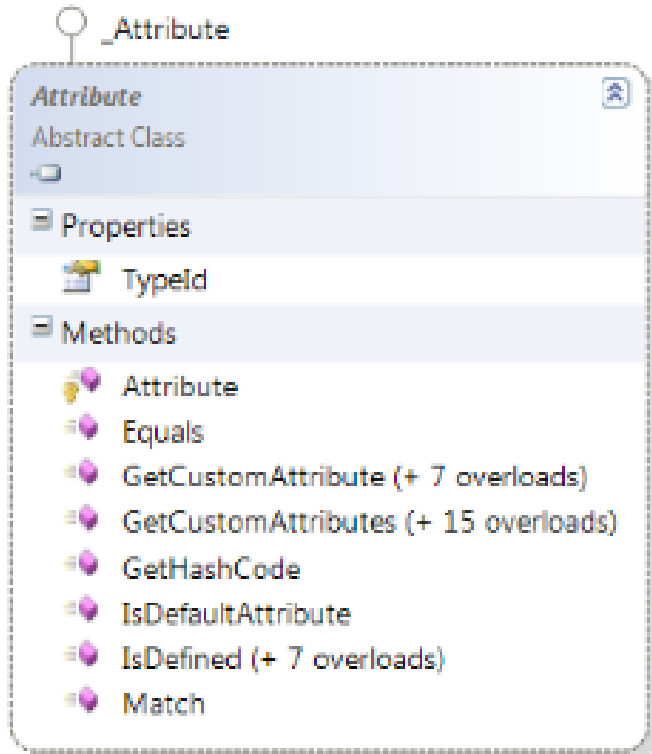
Информация об этих расширениях метаданных запрашивается *во время выполнения* с целью *динамического* изменения хода выполнения программы.

Атрибуты используются для служб, которые глубоко интегрированы в систему типов, и не требует специальных ключевых слов или конструкций в языке C#.

Существует два типа атрибутов:

- **Предопределенные** атрибуты (в составе .NET),
- **Пользовательские** атрибуты, создаваемые пользователем для добавления в код дополнительных сведений.

Атрибуты в платформе .NET являются **типами (классами)**, расширяющими абстрактный базовый класс **System.Attribute**.



- Наследование от класса **System.Attribute** обусловлено соответствием общезыковой спецификации CLS.
- Наследование может быть прямое или косвенное

Пример predefined атрибутов

[Obsolete]

Атрибут Obsolete позволяет пометить элемент программы как устаревший.

[Serializable]

информирует механизмы сериализации о том, что поля доступны для сериализации и десериализации.

[DllImport]

информирует CLR о том, что метод реализован в неуправляемом коде указанной DLL-библиотеки.

[NonSerialized]

Позволяет указать, что данное поле в классе или структуре не должно сохраняться в процессе сериализации

[AssemblyVersion]

при применении к сборке задает версию сборки.

[Flags]

указывает, что перечисление будет представлять набор битовых флагов.

Пример использования предопределённого атрибута





```
public sealed class ObsoleteAttribute : Attribute
{
    public ObsoleteAttribute();
    public ObsoleteAttribute(string message);
    public ObsoleteAttribute(string message, bool error);
    public bool IsError { get; }
    public string Message { get; }
}
```

```
// атрибут Obsolete - задает уведомление о том, что класс
// устарел и выводит дополнительно сообщение "Класс автомобиль!"
[Obsolete("Класс автомобиль!")]
class Car
{
    // значения атрибута Obsolete - устанавливает ошибки
    // при использовании метода ShowInfo()
    [Obsolete("Метод класса автомобиль!", true)]
    public void ShowInfo()
    {
        Console.WriteLine("Метод класса Car");
    }
}
```




```

class Program
{
    0 references
    static void Main(string[] args)
    {
        Car car = new Car();
        car.ShowInfo(); // Ошибка!
        Console.ReadLine();
    }
}

```


 1 of 1 Error
  2 of 2 Warnings
  0 of 0 Messages

Description ▲

-  1 'ExampleStandartAttribute.Car' is obsolete: 'Класс автомобиль!'
-  2 'ExampleStandartAttribute.Car' is obsolete: 'Класс автомобиль!'
-  3 'ExampleStandartAttribute.Car.ShowInfo()' is obsolete: 'Метод класса автомобиль!'

Параметры атрибутов

Многие атрибуты имеют параметры,

- позиционные,
- именованные.

```
[DllImport("user32.dll", ExactSpelling = false)]
```

Позиционные параметры:

аргументы конструктора
класса атрибута (всегда
указываются перед
именованными)

Именованные параметры:

все открытые нестатические
поля и свойства класса
атрибута, доступные для
записи

```
[DllImport("user32.dll")]
```

```
[DllImport("user32.dll", SetLastError=false, ExactSpelling=false)]
```

```
[DllImport("user32.dll", ExactSpelling=false, SetLastError=false)]
```

Целевые объекты атрибутов

Неявно целью атрибута является элемент кода, который находится непосредственно за атрибутом, но атрибуты можно присоединять и к сборке.

<i>assembly</i>	Целая сборка
<i>field</i>	Поле в классе или в структуре
<i>event</i>	События
<i>method</i>	Метод или метод доступа к свойствам get или set
<i>param</i>	Параметры методов или параметры методов доступа к свойствам set
<i>property</i>	Свойство
<i>return</i>	Возвращаемое значение метода, индексатор свойств или метод доступа к свойствам get
<i>type</i>	Структура, класс, интерфейс, перечисление или делегат

```
[assembly: AssemblyVersion("1.0.0.0")]  
[assembly: AssemblyFileVersion("1.0.0.0")]  
[assembly: AssemblyCopyright("Copyright © 2015")]
```

Указание атрибутов

Идентичные указания атрибутов:

```
[Serializable, Obsolete, CLSCompliant(false)]  
public class Car { }
```

```
[Serializable][Obsolete][CLSCompliant(false)]  
public class Car { }
```

```
[Serializable, Obsolete]  
[CLSCompliant(false)]  
public class Car { }
```

Правила создания пользовательского атрибута:

1. Имя атрибута должно содержать суффикс `Attribute`.
2. Класс-атрибут обязан наследоваться от системного класса `System.Attribute`
3. Класс-атрибут может быть отмечен атрибутом `[AttributeUsageAttribute]`

Пример объявление пользовательского атрибута

```
class InfoAboutAttribute : Attribute
{ // тело атрибута (класса)
    public string Desc {get;set;}
}
```

```
[InfoAbout(Desc="Класс автомобиль")]
class Car
{ // тело класса
}
```

Рекомендации по созданию пользовательского атрибута:

- Атрибут следует рассматривать как логический контейнер состояния - **этот класс должен быть крайне простым.**
- Атрибут должен содержать всего **один открытый конструктор**, принимающий обязательную (или позиционную) информацию о состоянии атрибута.
- Атрибут может содержать **открытые поля/свойства**, принимающие дополнительную (или именованную) информацию о состоянии атрибута .
- В классе **не должно быть открытых методов, событий или других членов.**

Важно! Лучше использовать в атрибуте свойства, так как они обеспечивают большую гибкость в случаях, когда требуется внести изменения в реализацию класса атрибутов.

Системный атрибут `AttributeUsage`

Используется для создания пользовательских атрибутов.

Позиционные параметры:

- **validOn** – имеет тип `AttributeTargets`, указывает, к чему можно применять данный атрибут.

Именованные параметры:

- **AllowMultiple** – имеет тип `bool`, разрешает или запрещает множественное применение атрибута (по умолчанию - `false`)
(возможно ли для одного элемента программы задать более одного экземпляра указанного атрибута)
- **Inherited** – имеет тип `bool`, разрешает или запрещает наследование атрибута в производных классах (по умолчанию - `true`).

```
public enum AttributeTargets
{ All, Assembly, Class, Constructor,
  Delegate, Enum, Event, Field,
  GenericParameter, Interface, Method,
  Parameter, Property, ReturnValue, Struct }
```

Варианты применения атрибута `AttributeUsage`

```
[AttributeUsage (AttributeTargets.Class)]  
// указывает, что атрибут буде применен только к классу  
class InfoAboutAttribute : Attribute  
{    public string Desc; }
```

```
[AttributeUsage (AttributeTargets.Class |  
                  AttributeTargets.Interface)]  
class InfoAboutAttribute : Attribute  
{    public string Desc; }
```

```
[AttributeUsage (AttributeTargets.Class | AttributeTargets.Interface,  
Inherited=true)]  
class InfoAboutAttribute : Attribute  
{    public string Desc; }
```

```
[AttributeUsage (AttributeTargets.Class | AttributeTargets.Interface,  
AllowMultiple=false, Inherited=true)]  
class InfoAboutAttribute : Attribute  
{    public string Desc; }
```


Пример передачи параметров атрибуту

```
[System.AttributeUsage(System.AttributeTargets.Class |  
                        System.AttributeTargets.Struct) ]  
public class AuthorAttribute : System.Attribute  
{ private string name;  
  public double version {get; set;};  
  public Author(string name)  
  {  
    this.name = name; version = 1.0;  
  }  
}
```

```
[Author("Г. Шилдт", version = 4.0)]  
class SampleClass  
{  
  //...  
}
```

Выявление настраиваемых атрибутов

Важно! После определения собственных классов атрибутов нужно также **написать код, проверяющий, существует ли экземпляр класса атрибута** (для указанных элементов), и в зависимости от результата изменять порядок выполнения программы.

```
InfoAboutAttribute att =  
    (InfoAboutAttribute)Attribute.  
        GetCustomAttribute(t,  
            typeof(InfoAboutAttribute));  
Console.WriteLine("{0}", att.Desc);
```

Пример объявление пользовательского атрибута

```
class InfoAboutAttribute : Attribute
{ // тело атрибута (класса)
    public string Desc;
}
```

```
[InfoAbout(Desc="Класс автомобиль")]
class Car
{ // тело класса
}
```

```
class InfoAboutClass
{
    public static void GetAttribute(Type t)
    {    // получение сведений об атрибуте
        InfoAboutAttribute att =
            (InfoAboutAttribute)Attribute.GetCustomAttribute
                (t, typeof(InfoAboutAttribute));

        Console.WriteLine("{0}", att.Desc); // Вывод переданной информации
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        // получение атрибута их типа!
        InfoAboutClass.GetAttribute(typeof(Car));
        Console.ReadKey();
    }
}
```

Дополнительно...

В версии C# 5.0, **необязательные параметры** можно помечать с помощью одного из трех атрибутов о вызывающем компоненте, которые инструктируют компилятор о необходимости передачи информации, полученной из исходного кода вызывающего компонента, в стандартное значение параметра:

- `[CallerMemberName]` применяет имя члена вызывающего компонента;
- `[CallerFilePath]` применяет путь к файлу исходного кода вызывающего компонента;
- `[CallerLineNumber]` применяет номер строки в файле исходного кода вызывающего компонента,

```
public static void LogWrite([CallerMemberName] string MemberName="")
{
    File.AppendAllText("log.txt", MemberName);
}
```

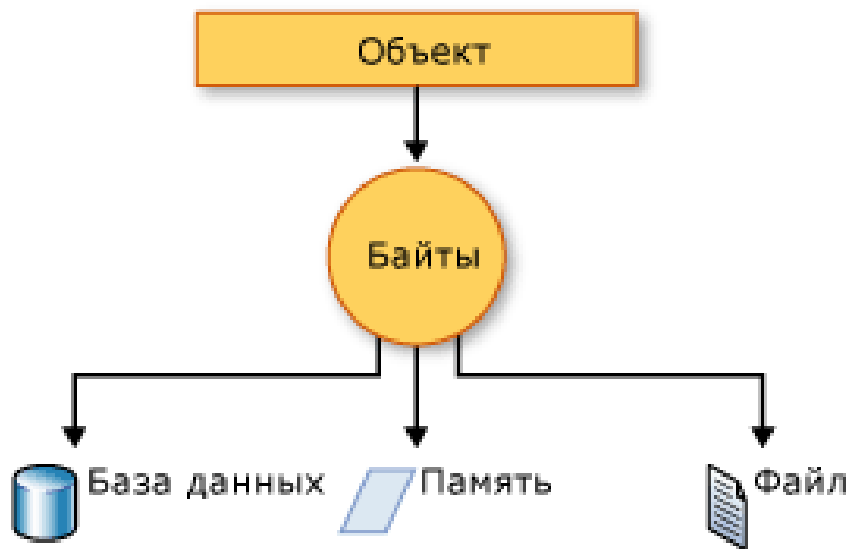
Сериализация



Понятие сериализации объектов

Сериализация представляет собой процесс преобразования находящегося в памяти объекта (графа объектов - набора объектов, ссылающихся друг на друга) в поток байтов (XML-узлов) для последующего его воссоздания.

- Сохраненная последовательность байт содержит всю необходимую информацию для воссоздания объекта.
- Обратный процесс называется **десериализацией**.



Объект сериализуется в поток, который переносит не только данные, но и сведения о типе объекта.

Превращение объекта в сериализуемый

Для сериализации объекта необходимо класс отметить атрибутом **[Serializable]**.

Для сериализации объектов можно использовать следующие классы:

- **BinaryFormatter** (двоичный файл);
- **SoapFormatter** (файл формата SOAP);
- **XmlSerializer** (XML-файл);

System.Runtime.Serialization
System.xml.Serialization

Класс BinaryFormatter

Сериализует состояние объекта в поток, используя компактный *двоичный формат* (двоичная кодировка).

Этот тип определен в пространстве имен **System.Runtime.Serialization.Formatters.Binary**

- **BinaryFormatter** представляет собой наиболее эффективный способ сериализации.
- Обеспечивает совместимость **ТОЛЬКО** между версиями.NET Framework.

Ограничение: сериализация и десериализация должны выполняться только.NET приложениями.

Пример сериализации объекта

Создание класса сериализуемого объекта

```
using System;
using System.Runtime.Serialization;
[Serializable]
class Book
{
    int id;
    string name;
    double price;
    public Book(int ID, string Name, double Price)
    { id=ID; name=Name; price=Price;}
    public override string ToString()
    {return string.Format("{0}.{1},{2}", id, name, price);}
}
```

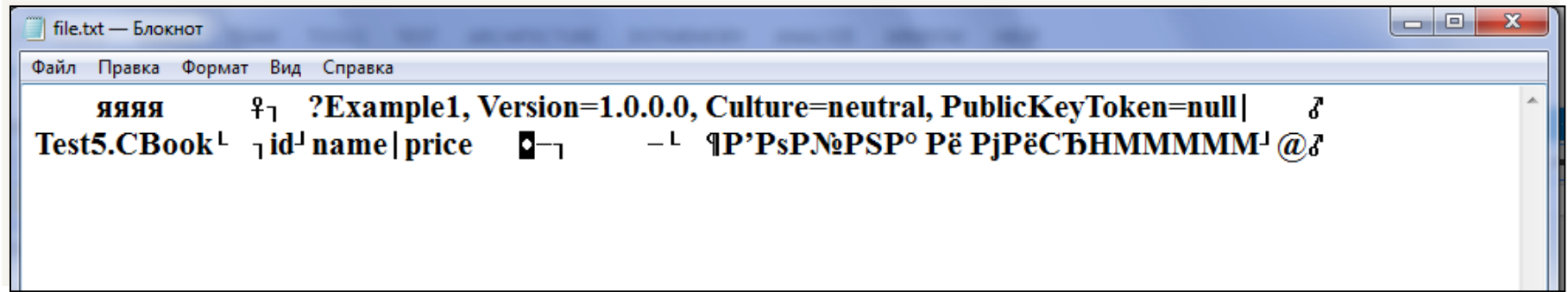
Пример сериализации объекта

1. Война и мир, 2, 6

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
class Program{ public static void Main()
{
    Book book=new Book(1, "Война и мир", 2.6);
    using (FileStream file=new FileStream("file.txt", FileMode.Create))
        {    BinaryFormatter binFormat = new BinaryFormatter();
            binFormat.Serialize(file,book);}

    Book openBook;    // десериализация объекта
    using (FileStream file=new FileStream("file.txt", FileMode.Open))
        {    BinaryFormatter binFormat = new BinaryFormatter();
            openBook= (Book)binFormat.Deserialize(file);    }

    Console.WriteLine(openBook);
    Console.ReadKey(true);
}}
```



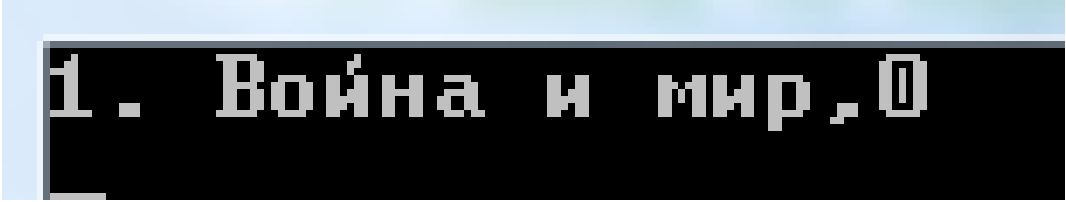
```
яяяя ?Example1, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null|
Test5.CBook id name price P'PsPNPSP Pë PjPëCЪHMMMMM @'
```

ВАЖНО! В C# внутри типов, помеченных атрибутом [Serializable], не стоит определять автоматически реализуемые свойства. Так как имена полей, генерируемые компилятором, **могут меняться после каждой следующей компиляции**, что сделает невозможной десериализацию экземпляров типа.

Пример сериализации объекта

Создание класса сериализуемого объекта с применением атрибута [NonSerialized]

```
[Serializable]
class Book
{
    public int id {get; set;}
    public string name {get; set;}
    [NonSerialized]// определение несериализуемого поля
    public double price{get; set;}
    public Book(int ID, string Name, double Price)
    { id = ID; name = Name; price = Price; }
    public override string ToString()
    {
        return string.Format("{0}.{1},{2}", id,name,price);
    }
}
```



[Serializable]

class Book

```
{ public int id {get; set;}
  public string name{get; set;}
  [NonSerialized] // определение несериализуемого поля
  public double price {get; set;}
  public Book(int ID, string Name, double Price)
  { id = ID; name = Name; price = Price; }
  public override string ToString()
  { return string.Format("{0}. {1},{2}", id, name, price); }
  [OnDeserialized] // определяются атрибутом [OnDeserialized]
  private void OnDeserialized(StreamingContext context)
  { // Инициализация после десериализации
    price = 5000;
  }
  [OnDeserializing]
  private void OnDeserializing(StreamingContext context)
  { // Присвоение полям значений в новой версии типа
  }
  [OnSerializing]
  private void OnSerializing(StreamingContext context)
  { // Модификация состояния перед сериализации
  }
}
```

1. Война и мир, 5000

Пример сериализации коллекции объектов

```
public static void Main()
{
    List <Book> books=new List<Book>()
    { new Book(1, "Война и мир", 2.6),
      new Book(2, "Отцы и дети", 5.1),
      new Book(3, "Анна Каренина", 7.3)};

    // сериализация коллекции объектов
    using (FileStream file=new FileStream("file.txt", FileMode.Create))
    { BinaryFormatter binFormat = new BinaryFormatter();
      binFormat.Serialize(file, books);
    }

    // десериализация коллекции объектов
    List <Book> newbooks;
    using (FileStream file=new FileStream("file.txt", FileMode.Open))
    { BinaryFormatter binFormat = new BinaryFormatter();
      newbooks= (List<Book>)binFormat.Deserialize(file);
    }

    foreach(Book book in newbooks)
        Console.WriteLine(book);

    Console.ReadKey(true);
}
```

Пример сериализации для копирования объектов

```
static void Main(string[] args)
{
    Book Book = new Book(1, "Троелсен C# 4.5", 2500);
    Book newBook = (Book)DeepClone(Book);
    Console.WriteLine(Book);    Console.WriteLine(newBook);

    Console.WriteLine(new string('-', 25));
    newBook.Price = 3000;
    Console.WriteLine(Book);    Console.WriteLine(newBook);
}

private static Object DeepClone(Object original)
{
    // Создание временного потока в памяти
    using (MemoryStream stream = new MemoryStream())
    {
        // Создания объекта для сериализации
        BinaryFormatter formatter = new BinaryFormatter ();
        // Сериализация объекта в поток в памяти
        formatter.Serialize(stream, original);
        // Возвращение к началу потока в памяти
        stream.Position = 0;
        // Десериализация в новый объект
        return formatter.Deserialize(stream);
    }
}
```

```
1. Троелсен C# 4.5,2500
1. Троелсен C# 4.5,2500
-----
1. Троелсен C# 4.5,2500
1. Троелсен C# 4.5,3000
```


Сериализация в XML-файл

- Подходит для сериализации открытых типов (классов) и членов типов (полей, свойств, методов...).
- Позволяет сериализовать только отдельные объекты.
- Для выполнения XML сериализации не обязательно использовать атрибут [**Serializable**].

```
using System.Xml.Serialization;
```

Для того, чтобы сериализовать объект в **формате XML** необходимо выполнить следующие действия:

1. Объявить класс как **открытый**.
2. Объявить все члены класса, которые необходимо сериализовать, как **открытые**.
3. Создать конструктор **не принимающий параметров**.

Сериализация в XML-файл

```
public class Book // public
{
    public int id;           // открытое поля сериализуются
    public string name;      // открытое поля сериализуются

    private double price;    // закрытые поля не сериализуются

    public int Pages { get; set; }

    // Обязательный конструктор!
    public Book()
    {

    }

    public Book(int ID, string Name, int pages, double Price)
    { id = ID; name = Name; price = Price; Pages = pages; }
    public override string ToString()
    { return string.Format("{0}. {1}, {2} стр. {3}",
                           id, name, Pages, price); }
}
```

Сериализация в XML-файл

```
using System.Xml.Serialization;
```

```
// коллекция для сериализации
```

```
List<Book> books = new List<Book>()
```

```
{ new Book(1, "Война и мир", 1500, 2.6),
```

```
  new Book(2, "Отцы и дети", 800, 5.1),
```

```
  new Book(3, "Анна Каренина", 860, 7.3)
```

```
};
```

```
List<Book> newBooks;
```

```
using (FileStream file = new FileStream("file.xml",
```

```
  FileMode.Create))
```

```
{ XmlSerializer xmlFormat = new XmlSerializer(typeof(List<Book>));
```

```
  xmlFormat.Serialize(file, books); // сериализация
```

```
}
```

```
// десериализация объекта
```

```
using (FileStream file = new FileStream("file.xml", FileMode.Open))
```

```
{ XmlSerializer xmlFormat = new XmlSerializer(typeof(List<Book>));
```

```
  newBooks = (List<Book>)xmlFormat.Deserialize(file);
```

```
}
```

```
// вывод на экран newBook - десериализованного объекта
```

```
foreach (Book book in newBooks) // вывод на экран коллекции
```

```
  Console.WriteLine(book.ToString());
```

Сериализация в XML-файл

```
<?xml version="1.0"?>
<ArrayOfBook
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Book>
    <id>1</id>
    <name>Война и мир</name>
    <Pages>1500</Pages>
  </Book>
  <Book>
    <id>2</id>
    <name>Отцы и дети</name>
    <Pages>800</Pages>
  </Book>
  <Book>
    <id>3</id>
    <name>Анна Каренина</name>
    <Pages>860</Pages>
  </Book>
</ArrayOfBook>
```

Сериализация в XML-файл

```
public class Book
{
    // переименовываем и делаем XML атрибутом.
    [XmlAttribute("Number")]
    public int Num { get; set; }
    //XML элемент.
    [XmlElement("Title")]
    public string Name { get; set; }
    // Исключаем свойство из процесса сериализации/десериализации.
    [XmlIgnore]
    public int Pages { get; set; }
    public double Price { get; set; }

    // Обязательный конструктор!
    public Book() { }

    public Book(int n, string name, int pages, double price)
    { Num = n; Name = name; Price = price; Pages = Pages; }
    public override string ToString()
    { return string.Format("{0}. {1}, {2} стр. {3}",
                            Num, Name, Pages, Price); }
}
```

Сериализация в XML-файл

```
<?xml version="1.0"?>
<ArrayOfBook
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Book Number="1">
    <Title>Война и мир</Title>
    <Price>2.6</Price>
  </Book>
  <Book Number="2">
    <Title>Отцы и дети</Title>
    <Price>5.1</Price>
  </Book>
  <Book Number="3">
    <Title>Анна Каренина</Title>
    <Price>7.3</Price>
  </Book>
</ArrayOfBook>
```

Сериализация в JSON

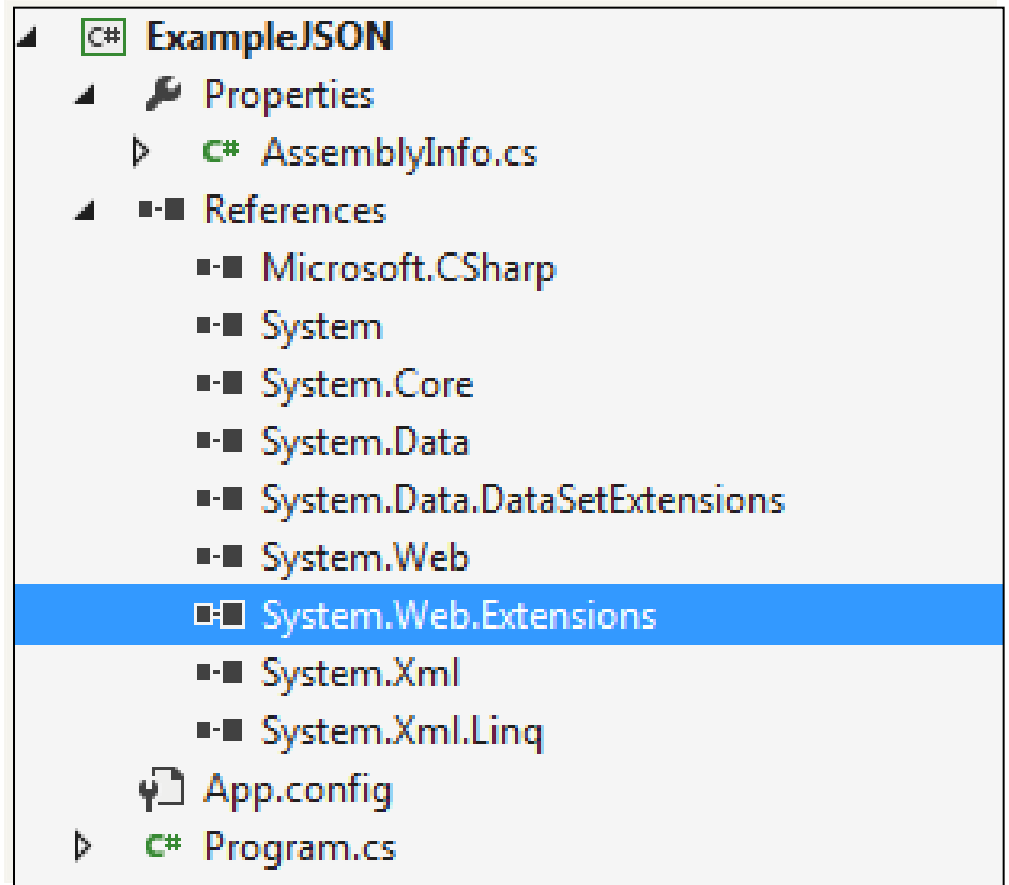
```
List<Book> books= new List<Book>
    {
        new Book{id=1, name="Война и мир", price=25000},
        new Book{id=2, name="Отцы и дети", price=18000}
    };
```

```
var jSerialize = new JavaScriptSerializer();
string json = jSerialize.Serialize(books);
Console.WriteLine(json);
```

```
Console.WriteLine(new string('-', 20));
List<Book> Books = jSerialize.Deserialize<List<Book>>(json);
foreach (Book book in Books)
    Console.WriteLine(book);
```

```
public class Book
{
    public int id;    public string name;    public double price;
    public Book() { }
    public Book(int ID, string Name, double Price)
    { id = ID; name = Name; price = Price; }
    public override string ToString()
    { return string.Format("{0}. {1},{2}", id, name, price); }
}
```

Сериализация в JSON



```
[{"id":1,"name":"Война и мир","price":25000}, {"id":2,"name":"Отцы и дети","price":18000}]
```