

ПИШНИК: BECOME HUMAN

Глава 18: Различные Способы
Сериализация.

Автор презентации – Сагалов Даниил БПШ-196



Сериализация

Сериализация – процесс преобразования объекта в поток байтов с последующим его сохранением в некоторый источник данных.

Сериализация предназначена в первую очередь для того, чтобы сохранять состояние объекта и иметь возможность восстановить его (например, при перезапуске программы).

Обратным процессом является **десериализация**, то есть восстановление структуры данных/объекта из последовательности байтов. Именно сериализация и десериализация позволяют нам реализовывать сохранение и загрузку данных программ.

Стоит понимать, что существуют различные механизмы сериализации, как предоставляемые .NET, так и созданные сторонними разработчиками. Несколько примеров механизмов сериализации в .NET:

- Двоичная сериализация
- XML-сериализация, `IXmlSerializable`
- Контракты данных (англ. Data Contracts)
- JSON-сериализация

Сериализация Средствами System.Runtime.Serialization

Опечатанный класс **SerializableAttribute** – атрибут, указываемый над классами, для которых применима сериализация. По умолчанию для типа, помеченного данным атрибутом все открытые и закрытые поля будут сериализованы, для предотвращения этого используйте **NonSerializedAttribute** для **BinaryFormatter** и **XmlIgnoreAttribute** для **XmlSerializer**. Для применения NonSerializable к событиям также применим следующий код:

```
[field:NonSerializedAttribute( )]
```

```
public event ExampleEventHandler Changed;
```

Для непосредственного переопределения механизма сериализации требуется реализация интерфейса **ISerializable**, причём классы, реализующие данный интерфейс всё равно рекомендуется помечаться атрибутом **Serializable**.

Важно: всегда помните, что атрибут **Serializable** обязателен при использовании средств **System.Runtime.Serialization**, при его отсутствии возникнет исключение **SerializationException**. Для корректной работы все соответствующие поля (включая поля типов вложенных объектов) должны быть сериализуемыми. Автоматически реализуемые свойства тоже сериализуются, но помечены **NonSerialized** быть не могут.

Атрибут **Serializable** не наследуется, все наследники должны повторно помечаться атрибутом **Serializable**. **NonSerialized** в свою очередь наследуется. Десериализация открытого конструктора не требует.

Класс BinaryFormatter

Для дальнейшей работы введём понятие **графа объектов** — некоторого набора взаимосвязанных объектов, участвующих в процедуре сериализации/десериализации.

BinaryFormatter — тип пространства имён **System.Runtime.Serialization.Formatters.Binary**, предназначенный для сериализации и десериализации объектов в двоичном формате.

Для сериализации объекта или графа объектов используется метод:
public void Serialize(Stream source, Object graph).

Для десериализации объекта или графа объектов используйте метод:
public Object Deserialize(Stream source). Обратите внимание, что метод возвращает ссылку типа **Object**, что создаёт необходимость в приведении типов. Всегда помните, что для корректной десериализации позиция в потоке должна совпадать с началом данных объекта/графа объектов.

Шаги Двоичной Сериализации:

- 1) Добавить пространство имён `System.Runtime.Serialization.Formatters.Binary`.
- 2) Создать сохраняемый объект (и убедиться, что он не null) и объект-форматер типа `BinaryFormatter`.
- 3) Открыть байтовый поток для сохранения сериализуемого объекта.
- 4) Вызвать метод `Serialize(Stream source, object graph)`.

Шаги Двоичной Десериализации:

- 1) Создать поле/переменную для десериализуемого объекта и объект-форматер.
- 2) Создать поток, связав его с источником данных.
- 3) Вызвать метод `Deserialize(Stream source)`, выполнить приведение типов (т. к. возвращается ссылка типа `Object`).



Шаги Двоичной Сериализации

Пример Двоичной Сериализации

// Сериализуемый тип

Ссылки: 4

```
[Serializable] public class Mammoth {  
    public float downiness;  
    // Поля, создаваемые компилятором для автоматически  
    // реализуемых свойств тоже сериализуются  
    Ссылки: 2  
    public int Size { get; private set; }  
    // Данное поле не сериализуется и при десериализации  
    // всегда имеет значение по умолчанию  
    [NonSerialized] private bool isExtinct;  
  
    ссылка: 1  
    public Mammoth(float downiness, int size, bool isExtinct = true) {  
        this.downiness = downiness;  
        Size = size;  
        this.isExtinct = isExtinct;  
    }  
    ссылка: 1  
    public override string ToString() => $"Downiness :3 : {downiness}, Size: {Size}\n" +  
        $"{{(isExtinct ? "is extinct :(" : "in great shape!")}}";  
}
```

```
class Program {
```

Ссылки: 0

```
static void Main() {
```

// Создаём формater и поле типа

```
BinaryFormatter bf = new BinaryFormatter();
```

```
Mammoth dima;
```

// Десериализуем объект, если файл с данными уже существует

```
if (File.Exists("serial.bin")) {
```

```
    Console.WriteLine("Starting mammoth defrostation...");
```

// Открываем поток для десериализации

```
using (FileStream sw = new FileStream("serial.bin", FileMode.Open))
```

// Не забываем привести тип к Mammoth

```
    dima = bf.Deserialize(sw) as Mammoth;
```

```
    Console.WriteLine($"Result: {dima}");
```

```
}
```

// Создаём и сериализуем объект, если файла нет

```
else {
```

```
    Console.WriteLine("Creating a new mammoth...");
```

```
    dima = new Mammoth(100f, 500, false);
```

// Открываем поток для сериализации

```
using (FileStream sw = new FileStream("serial.bin", FileMode.Create))
```

// Сериализуем объект

```
    bf.Serialize(sw, dima);
```

```
    Console.WriteLine($"Result: {dima}, serialized!");
```

```
}
```

```
}
```

Двоичная Сериализация Коллекций Объектов

Как уже упоминалось, Вы можете сериализовывать не только сами объекты, но и графы объектов. Так, Вы можете сериализовать не только массивы объектов, но и другие коллекции, такие как списки, словари и т. д.

Главное правило сериализации графов объектов — наличие возможности сериализовать каждый из объектов графа.

При десериализации важно помнить, что для корректности результатов порядок десериализации обязан совпадать с порядком сериализации.

Пример Двоичной Сериализации Коллекций

```
class Program {  
    Ссылки: 0  
    static void Main() {  
        // Создаём различные коллекции объектов  
        BinaryFormatter bf = new BinaryFormatter();  
        Mammoth[] mammothsArray = {  
            new Mammoth(100, 1000, false),  
            new Mammoth(200, 400, true),  
            new Mammoth(50, 2500, false)  
        };  
        List<Mammoth> mammothsList = new List<Mammoth> {  
            new Mammoth(300, 380, false),  
            new Mammoth(450, 400, true),  
            new Mammoth(1000, 2500, true)  
        };  
        Dictionary<string, Mammoth> mammothsDictionary = new Dictionary<string, Mammoth> {  
            ["Dima"] = new Mammoth(300, 380, false),  
            ["George"] = new Mammoth(450, 400, true),  
            ["Benedict"] = new Mammoth(1000, 2500, true)  
        };  
        // Сериализуем в один поток  
        using (FileStream fs = new FileStream("collectionsSerialization.bin", FileMode.Create)) {  
            bf.Serialize(fs, mammothsArray);  
            bf.Serialize(fs, mammothsList);  
            bf.Serialize(fs, mammothsDictionary);  
        }  
        // Обнуляем ссылки на все коллекции  
        mammothsArray = null;  
        mammothsList = null;  
        mammothsDictionary = null;  
        // Десериализуем обратно из того же потока (обязательно в том же порядке!)  
        using (FileStream fs = new FileStream("collectionsSerialization.bin", FileMode.Open)) {  
            mammothsArray = bf.Deserialize(fs) as Mammoth[];  
            mammothsList = bf.Deserialize(fs) as List<Mammoth>;  
            mammothsDictionary = bf.Deserialize(fs) as Dictionary<string, Mammoth>;  
        }  
    }  
}
```


XML Сериализация

В пространстве имён **System.Xml.Serialization** предоставлены средства для XML-сериализации. Для сериализации используется специальный тип **XmlSerializer**.

При XML-сериализации атрибут `Serializable` не используется. Следует понимать, что в данном случае используется абсолютно другой механизм сериализации с другими правилами:

- Для XML-десериализации с помощью `XmlSerializer` обязателен конструктор по умолчанию (не обязательно открытый), атрибут `OnDeserializing` не распознаётся. Дело в том, что именно конструктор без параметров используется при десериализации и может задавать значения вместо недостающих полей. Эта же особенность используется при отсутствии некоторых значений объекта в XML-документе — вместо ошибки им может задаваться указанное в конструкторе без параметров значение. Также при десериализации выполняются инициализаторы полей.
- При создании `XmlSerializer` нужно указать как минимум 1 параметр — тип сериализуемого объекта (объект `System.Type`).
- При реализации интерфейса `IXmlSerializable` полностью исключается использование стандартного сериализатора и необходимо вручную определять поведение с помощью `XmlReader` и `XmlWriter`.

Основы Работы с XmlSerializer

Как уже упоминалось, тип `XmlSerializer` в качестве обязательного параметра конструктора принимает объект `Type` – тип сериализуемого объекта. Далее для сериализации применяются методы `Serialize()` и `Deserialize()`. `XmlSerializer` может сериализовать следующие элементы:

- Открытые поля и автоматически реализуемые свойства (с открытым доступом и на чтение, и на запись) открытых классов.
- Классы, реализующие интерфейсы `ICollection` и `IEnumerable`.
- Объекты `XmlElement`, `XmlNode` и `DataSet`.

Помните, что `XmlSerializer` не сериализует методы и не сохраняет информацию о сборке или удостоверение типа. `XmlSerializer` по-разному обрабатывает классы, реализующие `IEnumerable` и `ICollection`, а именно:

- При реализации **`IEnumerable`** класс должен содержать **открытый метод `Add`**, принимающий один параметр, соответствующий по типу свойству **`IEnumerator.Current`** (`IEnumerator`, возвращаемый методом `GetEnumerator()`).
- При реализации **`ICollection`** сериализуемые значения **берутся из индексатора**, а не путём вызова `GetEnumerator()`. Важно помнить, что в таком формате сериализуется только коллекция, содержащаяся в классе, а все остальные открытые свойства и поля останутся нетронутыми.
- Класс, реализующий и **`IEnumerable`**, и **`ICollection`** должен иметь **открытый индексатор, принимающий значение типа `int` и открытое свойство `Count` типа `int`**. Параметр, передаваемый **`Add`** должен быть того же типа, что и возвращаемое индексатором значение или одним из его базовых.

Управление XML Сериализацией с Помощью Атрибутов

По умолчанию `XmlSerializer` использует те же имена членов в XML-документе в порядке их объявления, а также не поддерживает наследование. Тем не менее, можно использовать атрибуты в качестве параметров сериализации:

[XmlAttribute] — позволяет не добавлять член в XML-документ. Применяется к членам класса.

[XmlElement(Type type)] — позволяет сериализовать открытые типы наследников типа по ссылке родителей. Применяется к классу.

[XmlElement(string ElementName, string DataType, bool IsNullable, string Namespace)] — позволяет задавать корневой элемент XML-документа. `ElementName` — имя корневого элемента; `DataType` — тип данных XSD (документа схемы XML); `IsNullable` — необходим при использовании `xsi:nil`, т. е. для отсутствия значения элемента (см. спецификацию xml); `Namespace` — пространство имён корневого элемента. Применяется к классу.

[XmlElement(string ElementName, int Order...)] — позволяет сериализовать член класса как XML-элемент, даёт возможность указать имя элемента, его порядок в списке и прочие параметры.

[XmlAttribute] — позволяет сериализовать член класса как атрибут XML.

[XmlAttribute] — задаёт имя члена перечисления при сериализации, применяется только к полям перечисления.

[XmlText] — свойство или поле сериализуется как XML текст.

Пример XML Сериализации

```
public class Book
{
    // Меняем название XML-элемента в документе
    [XmlAttribute("Name")] public string name;
    // Меняем название в документе, задаём порядок в схеме
    Ссылка: 3
    [XmlElement("AutorName", Order = 2)] public string Author { get; set; }
    Ссылка: 6
    [XmlElement("SellPrice", Order = 1)] public decimal Price { get; set; }
    Ссылка: 3
    private int Id { get; set; }
    Ссылка: 4
    public Book(int id, string name, string author, decimal price) {
        Id = id >= 0 ? id : -1;
        this.name = name ?? "";
        Author = author ?? "MissingNo";
        Price = price;
    }
    // Замкнутый конструктор без параметров для десериализации
    Ссылка: 0
    private Book() {
        Id = -1;
        name = "DefaultName";
        Author = "MissingNo";
        Price = -1;
    }

    Ссылка: 5
    public override string ToString() => $"Id: {Id}, {name}" +
        $" by {Author} costs ${Price}";
}
```

Пример XML Сериализации

```
// Задаём имя корневого узла XML-документа
[XmlRoot(Namespace = "www.teamalternative.com",
  ElementName = "XmlStockExample")]
[XmlType("BookShop")]
Ссылка: 6
public class BookStore {
    // Сериализуемая коллекция должна быть открытой
    public readonly List<Book> bookList;
    Ссылка: 0
    public void AddBook(Book book) => bookList.Add(book);
    Ссылка: 0
    public decimal RemoveBook(Book book, decimal payment) {
        if (!bookList.Contains(book))
            throw new ArgumentOutOfRangeException("There is no such book");
        if (payment < book.Price)
            throw new ArgumentException($"You need extra" +
                $" ${book.Price - payment} to afford this book!");
        decimal soldBookChange = payment - book.Price;
        bookList.Remove(book);
        return soldBookChange;
    }
    Ссылка: 1
    public BookStore(IEnumerable<Book> books) {
        if (books.Count() == 0)
            throw new ArgumentException("The collection provided contains 0 elements.");
        bookList = new List<Book>();
        foreach (var book in books) {
            bookList.Add(book);
        }
    }
    // Конструктор для десериализации
    Ссылка: 0
    private BookStore() => bookList = new List<Book>();
    Ссылка: 5
    public override string ToString() {
        StringBuilder stock = new StringBuilder();
        stock.Append("Books in stock: " + bookList.Count + Environment.NewLine);
        foreach (var book in bookList)
            stock.Append($"{book}" + Environment.NewLine);

        return stock.ToString();
    }
}
```

```
class Program {
    Ссылка: 0
    static void Main() {
        Book[] books =
        {
            new Book(0, "C# via CLR", "Jeffrey Richter", 3500m),
            new Book(1, "Head first C#", "J. Greene, A. Stellman", 1800m),
            new Book(2, "C# in a nutshell", "J. Albahari", 2500m),
            new Book(3, "C# 4.0 the complete reference", "H. Schildt", 4000m)
        };
        BookStore store = new BookStore(books);
        Console.WriteLine(store);
        // Указываем тип сериализуемого объекта в конструкторе
        XmlSerializer ser = new XmlSerializer(typeof(BookStore));
        using (FileStream fs = new FileStream("bookstore.xml", FileMode.Create))
            ser.Serialize(fs, store);

        store = null;
        using (FileStream fs = new FileStream("bookstore.xml", FileMode.Open))
            store = ser.Deserialize(fs) as BookStore;
        Console.WriteLine(store);
    }
}
```

Пример XML Сериализации

```
<?xml version="1.0"?>
<XmlStockExample xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns="www.teamalternative.com">
  <bookList>
    <Book Name="C# via CLR">
      <SellPrice>3500</SellPrice>
      <AutorName>Jeffrey Richter</AutorName>
    </Book>
    <Book Name="Head first C#">
      <SellPrice>1800</SellPrice>
      <AutorName>J. Greene, A. Stellman</AutorName>
    </Book>
    <Book Name="C# in a nutshell">
      <SellPrice>2500</SellPrice>
      <AutorName>J. Albahari</AutorName>
    </Book>
    <Book Name="C# 4.0 the complete reference">
      <SellPrice>4000</SellPrice>
      <AutorName>H. Schildt</AutorName>
    </Book>
  </bookList>
</XmlStockExample>
```

В результате работы продемонстрированного фрагмента будет получен XML-документ с заданным корневым узлом в указанном пространстве имён.

Обратите внимание, что все элементы имеют имена, отличные от указанных в качестве имён полей (кроме bookList).

Название книги при этом задаётся как XML-атрибут, а порядок отображения имени автора и цены изменён.

Контракты Данных

Контракт данных — формальное соглашение, описывающее данные, обмен которыми осуществляется. Вводит концепцию использования одних и тех же контрактов данных клиентом и службой при отсутствии необходимости использования одних и тех же типов. Изначально были введены как часть Windows Communication Foundation (WCF).

Для XML сериализации используется тип **DataContractSerializer** пространства имён **System.Runtime.Serialization**, для JSON сериализации — **DataContractJsonSerializer** пространства имён **System.Runtime.Serialization.Json**.

Альтернативами для JSON сериализации служат **System.Text.Json** и **Newtonsoft.Json**, о которых будет рассказано далее в этой презентации.

Типы, Используемые с Сериализатором Контрактов Данных

Все ниже перечисленные типы могут быть сериализованы с помощью сериализатора контрактов данных:

- Все открытые типы с конструктором без параметров.
- Все типы данных, помеченные атрибутом [DataContract].
- Типы коллекций, для настройки сериализации типов коллекций используется опциональный атрибут [CollectionDataContract].
- Типы перечислений, включая перечисления флагов. При этом в дополнение к [DataContract] все члены перечисления должны помечаться атрибутом [EnumMember], непомеченные не сериализуются.
- Типы-примитивы .NET – Int16, Int32, Boolean, Double, Char и другие.
- Примитивы в контексте XML – DateTime, DateTimeOffset, TimeSpan, Guid, Uri, XmlQualifiesName, Массивы Byte.
- Все типы, помеченные атрибутом Serializable.
- Тип XElement, массивы XmlNode, XDocument, XElement и все типы, реализующие IXmlSerializable.

Класс DataContractSerializer

DataContractSerializer – опечатанный класс пространства имён **System.Runtime.Serialization**, предназначенный для сериализации/десериализации объектов с помощью контрактов данных в/из XML.

Для сериализации тип должен быть помечен атрибутом **[DataContract]**, а его сериализуемые члены – атрибутом **[DataMember]** (**[EnumMember]** для членов перечислений).

[DataContract] позволяет дополнительно задать имя (**Name**), пространство имён (**Namespace**) и определить, сохраняются ли ссылки на объекты (**IsRefence**, если задать значение **true**, то один и тот же объект по разным ссылкам не будет сериализовываться несколько раз). Не наследуется.

[DataMember] позволяет дополнительно задать имя (**Name**), порядок сериализации (**Order**) и определить значения, обязательные для сериализации (**IsRequired**).

[KnownType(Type childType)] – крайне важный атрибут, позволяющий избежать ошибки при сериализации наследников.

Для записи объекта в поток используется метод **public virtual void WriteObject(Stream stream, Object object)**, кроме того, вместо **Stream** может использоваться **XmlWriter** или **XmlDictionaryWriter**.

Для чтения используется метод **public override Object ReadObject(Stream stream)**, вместо **Stream** может использоваться **XmlReader** или **XmlDictionaryReader**. При десериализации конструктор по умолчанию не вызывается.

XML: Контракты Данных

```
// Указываем, что ссылочные типы не дублируются
[DataContract(IsReference = true)]

Ссылка: 5
public class Llama {
    // Сериализуем открытое свойство с private get
    ссылка: 1
    [DataMember] public string Name { get; private set; }
    // Сериализуем свойство с другим именем
    ссылка: 1
    [DataMember(Name = "ShotRange")] public double SpitDistance { get; private set; }
    // Две ссылки на один объект, который будет сериализован единожды
    [DataMember] public LlamaSpit spit1, spit2;

    ссылка: 1
    public Llama(string name, double spitDistance, double spitLife, double toxicity) {
        Name = name;
        SpitDistance = spitDistance;
        spit1 = new LlamaSpit(spitLife, toxicity);
        spit2 = spit1;
    }
}

// Сохранение ссылок
[DataContract(IsReference = true)]

Ссылка: 3
public class LlamaSpit {
    public double lifespan;
    [DataMember] private double toxicity;

    ссылка: 1
    public LlamaSpit(double lifespan, double toxicity) {
        this.lifespan = lifespan;
        this.toxicity = toxicity;
    }
}
```

```
class Program {
    Ссылка: 0
    static void Main() {
        Llama myLlama = new Llama("Karl", 1.0, 5.0, 100);
        // Создаём сериализатор лам
        DataContractSerializer sz = new DataContractSerializer(typeof(Llama));
        using (FileStream fs = new FileStream("datacontracts.xml", FileMode.Create))
            sz.WriteObject(fs, myLlama);

        myLlama = null;
        // Все несохранённые значения при десериализации будут иметь значения по умолчанию
        using (FileStream fs = new FileStream("datacontracts.xml", FileMode.Open))
            myLlama = sz.ReadObject(fs) as Llama;
    }
}

<Llama z:Id="i1" xmlns="http://schemas.datacontract.org/2004/07/SEStudent_Become_Human"
    xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/">
    <Name>Karl</Name>
    <ShotRange>1</ShotRange>
    <spit1 z:Id="i2">
        <toxicity>100</toxicity>
    </spit1>
    <spit2 z:Ref="i2"/>
</Llama>
```

JSON Сериализация: Контракты Данных

Для JSON сериализации с использованием контрактов данных используются объекты типа **DataContractJsonSerializer** пространства имён **System.Runtime.Serialization.Json**.

Обратите внимание, что свойство **IsReference** не должно использоваться при JSON сериализации, т. к. приводит к исключению **SerializationException** (JSON не поддерживает ссылочные типы).

```
[DataContract] public class Llama {  
    ссылка: 1  
    [DataMember] public string Name { get; private set; }  
    ссылка: 1  
    [DataMember(Name = "ShotRange")] public double SpitDistance { get; private set; }  
    [DataMember] public LlamaSpit spit1, spit2;  
  
    ссылка: 1  
    public Llama(string name, double spitDistance, double spitLife, double toxicity) {  
        Name = name;  
        SpitDistance = spitDistance;  
        spit1 = new LlamaSpit(spitLife, toxicity);  
        spit2 = spit1;  
    }  
}  
Ссылок: 3  
[DataContract] public class LlamaSpit {  
    public double lifespan;  
    [DataMember] private double toxicity;  
    ссылка: 1  
    public LlamaSpit(double lifespan, double toxicity) {  
        this.lifespan = lifespan;  
        this.toxicity = toxicity;  
    }  
}
```

JSON: Контракты Данных

```
class Program {  
    Ссылка: 0  
    static void Main() {  
        Llama myLlama = new Llama("Karl", 1.0, 5.0, 100);  
        // Создаём сериализатор лам (теперь в JSON)  
        DataContractJsonSerializer sz = new DataContractJsonSerializer(typeof(Llama));  
        using (FileStream fs = new FileStream("datacontracts2.json", FileMode.Create))  
            sz.WriteObject(fs, myLlama);  
  
        myLlama = null;  
        // Все несохранённые значения при десериализации будут иметь значения по умолчанию  
        using (FileStream fs = new FileStream("datacontracts2.json", FileMode.Open))  
            myLlama = sz.ReadObject(fs) as Llama;  
    }  
}
```

```
{  
    "Name": "Karl",  
    "ShotRange": 1,  
    "spit1": { "toxicity": 100 },  
    "spit2": { "toxicity": 100 }  
}
```


Обзор System.Json.Text

Вместе с .NET Core 3.0 дебютировало пространство имён **System.Text.Json**, содержащее классы для более удобной и быстрой работы с файлами формата JSON. Хотя System.Text.Json входит в состав платформы .NET Core 3.0, для работы с .NET Framework 4.7.2+, .NET Core 2.0, 2.1 и 2.2, .NET Standard 2.0+ необходимо устанавливать данную библиотеку как пакет NuGet (Средства → Диспетчер Пакетов NuGet).

Основой JSON сериализации средствами System.Text.Json является **статический класс JsonSerializer**, предоставляющей набор для синхронной и асинхронной сериализации/десериализации в JSON.

Обратите внимание, что в JSON используется кодировка UTF-8, в связи с этим использование string в качестве сериализуемого типа не так эффективно (string использует кодировку UTF-16). По этой причине для методов JsonSerializer существует ряд перегрузок, принимающих типы **Utf8JsonReader** и **Utf8JsonWriter**.

Ссылка на JSON RFC: <https://tools.ietf.org/html/rfc8259>

JsonSerializer ведёт себя следующим образом:

- Все открытые свойства по умолчанию сериализуются, для исключения из сериализации используйте атрибут **[JsonIgnore]**.
- Кодировщик по умолчанию экранирует не ASCII символы, символы, учитывающие HTML, в пределах диапазона ASCII и символы, которые должны быть экранированы в соответствии со спецификацией JSON.
- По умолчанию JSON сокращается, это можно настроить с помощью **JsonSerializerOptions**.
- По умолчанию регистр имён совпадает с регистром .NET, но Вы можете настроить имя элемента с помощью атрибута **[JsonPropertyname]**.

Важно: сериализация полей не поддерживается!

При JSON сериализации **поддерживаются следующие типы:**

- Прimitives типы, сопоставимые с типами JavaScript.
- РОСО-объекты (Plain Old Class Objects, т. е. классы, не возвращающие специальных объектов и не наследующиеся от них).
- Одномерные массивы и массивы массивов.
- Коллекции пространств имён: **System.Collections**, **System.Collections.Generic**, **System.Collections.Immutable**.
- Dictionary<string,TValue>, где TValue – Object, JsonElement или РОСО.



Поведение JSON Сериализатора

JsonSerializer ведёт себя следующим образом при десериализации:

- По умолчанию при сопоставлении имён учитывается регистр, для настройки используйте **JsonSerializerOptions**.
- Свойства только для чтения, записанные в JSON-файл при десериализации игнорируются, исключений не создаётся.
- Для десериализации обязательно должен быть конструктор без параметров (не обязательно публичный). Также при десериализации повторно сработают инициализаторы.
- Поля не поддерживаются (как и в случае с сериализацией)
- По умолчанию комментарии и завершающие запятые не поддерживаются и вызывают исключение, для настройки используйте **JsonSerializerOptions**.
- Максимальная глубина вложенности по умолчанию равна 64.



Поведение JSON Десериализатора

Класс Utf8JsonWriter

Utf8JsonWriter – опечатанный класс, реализующий `IDisposable` и `IAsyncDisposable`, предназначенный для некэшированной записи текста JSON в кодировке UTF-8. Соблюдает формат JSON RFC, за исключением написания комментариев.

Для создания объекта существует 2 конструктора, один из которых имеет вид:

public Utf8JsonWriter (Stream utf8Json, JsonWriterOptions options = default). Параметры по умолчанию дают возможность записывать JSON без дополнительных пробелов с проверкой соответствия с RFC.

Структура `JsonWriterOptions` содержит 3 свойства типа `bool`:

Свойство	Назначение
Encoder	Кодировщик при экранировании строк. При значении <code>null</code> используется кодировщик по умолчанию.
Intended	Следует ли кодировщику форматировать выходные данные JSON: добавлять отступы для вложенных токенов, новые строки и пробелы между именами и значениями.
SkipValidation	Разрешается ли запись недопустимого JSON.

Класс JsonSerializerOptions

Опечатанный класс JsonSerializerOptions предназначен для определения параметров сериализатора JSON и содержит следующие свойства:

public bool AllowTrailingCommas { get; set; } – разрешена ли лишняя запятая в конце списка значений JSON в объекте и игнорируется ли она при десериализации. По умолчанию false – запятая запрещена и при её наличии возникнет **JsonException**.

public int DefaultBufferSize { get; set; } – размер буфера в байтах, используется при создании временных буферов. По умолчанию равен 16384, чего достаточно для стандартных нагрузок. Рекомендуется не менять без необходимости.

public bool IgnoreNullValues { get; set; } – пропускаются ли значения null при сериализации, по умолчанию false.

public bool IgnoreReadOnlyProperties { get; set; } – игнорируются ли свойства только для чтения при сериализации, по умолчанию false. Не влияет на десериализацию.

public int MaxDepth { get; set; } – максимальная глубина вложенности JSON при сериализации и десериализации – при значении 0 задаётся числом 64. Отрицательные значения приводят к **ArgumentOutOfRangeException**.

public bool PropertyNameCaseInsensitive { get; set; } – используется ли сравнение имён без учёта регистра, по умолчанию false.

public bool WriteIndented { get; set; } – используется ли красивое форматирование, по умолчанию false.

Некоторые Методы Utf8JsonWriter

public void Reset()

public void Reset(Stream newStream) – восстанавливает внутреннее состояние объекта, чтобы его можно было использовать повторно, перегрузка допускает использование с новым потоком.

public void WriteBoolean(string propertyName, bool value) – записывает строчное имя свойства и соответствующее ему логическое значение в виде литерала JSON как пару имя-значение объекта JSON.

public void WriteBooleanValue(bool value) – записывает логическое значение в качестве элемента массива JSON.

public void WriteCommentValue(string value) – записывает строку в качестве JSON комментария в кодировке UTF-8.

public void WriteNumber(string propertyName, <Primitive Type> value) – записывает строчное имя свойства и соответствующее ему значение в составе пары имя-значение объекта JSON. WriteNumberValue – только число.

public void WriteString(string propertyName, <allowed Type> value) – записывает строчное имя свойства и соответствующее ему значение в составе пары имя-значение объекта JSON. allowedType может быть: String, DateTime, DateTimeOffset, Guid, JsonEncodedText, ReadOnlySpan<Byte>, ReadOnlySpan<Char>.

ОСНОВНЫЕ МЕТОДЫ JsonSerializer

public static string Serialize(object value, Type inputType, JsonSerializerOptions options = default) – преобразует значение указанного типа в строку JSON.

public static string Serialize<TValue>(TValue value, JsonSerializerOptions options = default) – универсальный вариант преобразования значения указанного типа в строку JSON.

public static void Serialize(Utf8JsonWriter writer, Object value, Type type, JsonSerializerOptions options = default) – записывает JSON-представление указанного типа в предоставленный источник.

public static void Serialize<TValue>(Utf8JsonWriter writer, TValue value, JsonSerializerOptions options = default) – универсальный вариант записи JSON-представления указанного типа в предоставленный источник.

public static object Deserialize(string json, Type returnType, JsonSerializerOptions options = default) – выполняет синтаксический анализ и преобразует строку текста JSON к указанному типу.

public static TValue Deserialize<TValue>(string json, JsonSerializerOptions options = default) – выполняет синтаксический анализ и преобразует строку текста JSON к указанному типу, универсальный вариант.

Основные Методы JsonSerializer

public static byte[] SerializeToUtf8Bytes(object value, Type inputType, JsonSerializerOptions options = default) – преобразует значение указанного типа в строку JSON, закодированную как массив байтов. На 5-10% быстрее по причине отсутствия необходимости перекодировать байты в строки.

public static byte[] SerializeToUtf8Bytes<TValue>(TValue value, JsonSerializerOptions options = default) – преобразует значение указанного типа в строку JSON, закодированную как массив байтов, универсальный вариант.

Помните, что для «красивой» (с отступами) печати сериализуемых данных с помощью `Utf8JsonWriter` должно быть указано `Indented = true` для `JsonWriterOptions`, `WriteIndented = true` для `JsonSerializerOptions` при этом не обязательно.

```

public class DrEdgarZomboss {
    // Авто. реализуемое свойство с инициализатором
    Ссылка: 2
    public int Intelligence { get; private set; } = 5000;
    Ссылка: 3
    public DateTime HouseAttackTime { get; private set; }
    Ссылка: 2
    public int ZombieCount { get; private set; } = 100000;
    // Секретный план в JSON не входит
    Ссылка: 2
    [JsonIgnore] public string SecretPlan { get; private set; }
    // Словарь - тоже свойство
    Ссылка: 2
    public Dictionary<string, string> armedZombies { get; private set; }
    Ссылка: 2
    public string[] threats { get; private set; }
    // Поля не сериализуются, инициализатор для десериализации
    public double zombotModel = 5000.05;
    ссылка: 1
    public DrEdgarZomboss(int iq, DateTime attackTime, int zombieCount,
        string plan, Dictionary<string, string> zombies, string[] threats) {
        Intelligence = iq;
        HouseAttackTime = attackTime;
        ZombieCount = zombieCount;
        SecretPlan = plan;
        armedZombies = zombies;
        this.threats = threats;
    } // Конструктор без параметров для десериализации
    Ссылка: 0
    private DrEdgarZomboss() {
        HouseAttackTime = DateTime.Now;
        SecretPlan = "There are no zombies coming actually";
        armedZombies = new Dictionary<string, string>();
        threats = new string[0];
    }
    Ссылка: 6
    public override string ToString() {
        return $"Hello, dear neighbour. Looks like there are \n" +
            $"{ZombieCount} zombies coming to your house at {HouseAttackTime.TimeOfDay}.\n" +
            $"Please do not underestimate my intelligence ({Intelligence},)\n" +
            $"give your brains and nobody will be harmed by my Zombot-{zombotModel}.";
    }
}

```

JSON Сериализация: Пример

В данном примере сериализуется тип, содержащий множество свойств различного типа — некоторые из них являются коллекциями, присутствует структура DateTime.

Обратите внимание, что аксессор set закрытый.

Поле в сериализации не участвует, однако за счёт инициализатора его значение восстанавливается.

```

class Program {
    Ссылка: 0
    static void Main() {
        string[] threats = { "fool", "loser", "slug", "insane" };
        string plan = "Attacking the roof first";
        Dictionary<string, string> zombies = new Dictionary<string, string>
        {
            ["Basic"] = "None",
            ["ConeHead"] = "Traffic Cone",
            ["BucketHead"] = "Bucket",
            ["Gargantua"] = "Light Pole",
            ["Flying"] = "Balloon",
            ["Geeky"] = "Arcade Machine"
        };

        DrEdgarZomboss zomboss = new DrEdgarZomboss(1000, DateTime.Now,
            zombies.Count, plan, zombies, threats);
        // Сериализует весь объект без пробелов в виде одной строки
        string result = JsonSerializer.Serialize<DrEdgarZomboss>(zomboss);
        Console.WriteLine(result);
        // Используем красивую печать с отступами
        JsonSerializerOptions writerOptions = new JsonSerializerOptions { Indented = true };
        using (FileStream fs = new FileStream("zomboss.json", FileMode.Create)) {
            using (Utf8JsonWriter jw = new Utf8JsonWriter(fs, writerOptions)) {
                JsonSerializer.Serialize<DrEdgarZomboss>(jw, zomboss);
            }
        }
        // Считываем файл, десериализуем полученную строку, смотрим результат
        string jsonString = File.ReadAllText("zomboss.json");
        zomboss = JsonSerializer.Deserialize<DrEdgarZomboss>(jsonString);
        Console.WriteLine(zomboss);
    }
}

```

JSON Сериализация: Пример

```

{
  "Intelligence": 1000,
  "HouseAttackTime": "2020-06-01T10:47:22.9741853+03:00",
  "ZombieCount": 6,
  "armedZombies": {
    "Basic": "None",
    "ConeHead": "Traffic Cone",
    "BucketHead": "Bucket",
    "Gargantua": "Light Pole",
    "Flying": "Balloon",
    "Geeky": "Arcade Machine"
  },
  "threats": [
    "fool",
    "loser",
    "slug",
    "insane"
  ]
}

```

Консоль отладки Microsoft Visual Studio

```

{"Intelligence":1000,"HouseAttackTime":"2020-06-01T11:13:12.5546049+03:00","ZombieCount":6,"armedZombies":{"Basic":"No
ne","ConeHead":"Traffic Cone","BucketHead":"Bucket","Gargantua":"Light Pole","Flying":"Balloon","Geeky":"Arcade Machine
"},"threats":["fool","loser","slug","insane"]}
Hello, dear neighbour. Looks like there are
100000 zombies coming to your house at 11:13:12.5915069.
Please do not underestimate my intelligence (5000,)
give your brains and nobody will be harmed by my Zombot-5000,05.

```

Задание 1

В результате выполнения фрагмента программы:

```
using System;
using System.IO;
```

```
class Program {
    static void Main() {
        FileStream fs = new FileStream("file.txt",
        FileMode.OpenOrCreate);
        using (StreamWriter sw = new StreamWriter(fs)) {
            for (int i = 0; i < 5; i++) {
                sw.Write(i.CompareTo(i * i - i));
            }
            using (StreamReader sr = new StreamReader(fs)) {
                Console.Write(sr.ReadToEnd());
            }
        }
    }
}
```

на экран будет выведено (файл file.txt до запуска программы не существовал):

Примечание:

*Если возникнет ошибка компиляции, введите: ****

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Задачи

Задание 2

В результате выполнения фрагмента программы:

```
using System;
using System.IO;
```

```
class Program {
    static void Main() {
        FileStream fs = new FileStream("file.txt",
        FileMode.OpenOrCreate);
        using (StreamWriter sw = new StreamWriter(fs)) {
            for (int i = 0; i < 5; i++) {
                sw.Write(i.CompareTo(i * i - i));
            }
        }
        fs = new FileStream("file.txt", FileMode.OpenOrCreate);
        using (StreamReader sr = new StreamReader(fs)) {
            Console.Write(sr.ReadToEnd());
        }
    }
}
```

на экран будет выведено (файл file.txt до запуска программы не существовал):

Примечание:

*Если возникнет ошибка компиляции, введите: ****

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Задание 3

В результате выполнения фрагмента программы:

```
using System;
using System.IO;
class Program {
    static void Main() {
        FileStream fs = new FileStream("int.txt", FileMode.Open);
        using (StreamWriter sw = new StreamWriter(fs)) {
            sw.WriteLine(1);
            sw.WriteLine(2);
            sw.WriteLine(3);
            using (StreamReader sr = new StreamReader(fs)) {
                Console.Write(sr.ReadToEnd());
            }
        }
    }
}
```

могут быть выброшены исключения:

- 1) UnauthorizedAccessException
- 2) FileNotFoundException
- 3) ArgumentException
- 4) ObjectDisposedException
- 5) ArgumentNullException

Задание 4

Абстрактными классами являются (укажите все верные ответы):

- 1) TextWriter;
- 2) StringWriter;
- 3) TextReader;
- 4) StringReader;
- 5) Stream;



ЗАДАЧИ

Задание 5

В результате выполнения фрагмента программы:

```
using System;
using System.IO;

class Program {
    static void Main() {
        TextWriter tr = Console.Out;
        Console.SetOut(new StreamWriter("text.txt"));
        for (int i = 0; i < sizeof(long); i += 2) {
            Console.Write(i * i);
            Console.SetOut(tr);
        }
    }
}
```

на экран будет выведено (файл text.txt до запуска программы не существовал):

Примечание:

*Если возникнет ошибка компиляции, введите: ****

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Задание 6

Метод Dispose() реализуют следующие классы (укажите все верные ответы):

- 1) StreamWriter;
- 2) StringWriter;
- 3) TextReader;
- 4) StreamReader;
- 5) MemoryStream;

Задачи

Задание 7

В результате выполнения фрагмента программы:

```
using System;
using System.IO;

class Program {
    static void Main() {
        StreamWriter sw = new StreamWriter(new FileStream("tt.txt",
        FileMode.OpenOrCreate));
        Console.SetOut(sw);
        for (int i = 0; i < sizeof(bool); i++) {
            Console.WriteLine(i * i);
            Console.SetOut(Console.Out);
        }
    }
}
```

на экран будет выведено (файл tt.txt до запуска программы не существовал):

Примечание:

*Если возникнет ошибка компиляции, введите: ****

Если ошибок и исключений нет, но на экран не выведется ничего, введите: ---

Если возникнет ошибка исполнения или исключение, введите: +++

Ответы

Задание	Ответ
1	+++ (<i>ObjectDisposedException</i>)
2	010-1-1
3	12345
4	135
5	41636
6	12345
7	--- (<i>Консольный вывод перенаправлен в файл</i>)