

Серіалізація об'єктів

Поняття серіалізації

Термін «серіалізація» описує процес збереження (та, можливо, передачі) стану об'єкта в потоці (наприклад, файловому потоці чи потоці в пам'яті).

- Последовательность сохраняемых данных содержит всю информацию, необходимую для реконструкции (или десериализации) состояния объекта с целью последующего использования.
- Применяя эту технологию, очень просто сохранять большие объемы данных (в различных форматах) с минимальными усилиями.
- Во многих случаях сохранение данных приложения с использованием служб сериализации выливается в код меньшего объема, чем применение классов для чтения/записи из пространства имен `System.IO`.

Например, предположим, что требуется создать настольное приложение с графическим пользовательским интерфейсом, которое должно предоставлять конечным пользователям возможность сохранения их предпочтений (цвета окон, размер шрифта и т.п.).

- Для этого можно определить класс по имени `UserPrefs` и инкапсулировать в нем около двадцати полей данных. В случае применения типа `System.IO.BinaryWriter` придется вручную сохранять каждое поле объекта `UserPrefs`.
- Аналогично, когда вы понадобится загрузить данные из файла обратно в память, нужно будет использовать `System.IO.BinaryReader` и, опять-таки, вручную читать каждое значение, чтобы реконструировать новый объект `UserPrefs`.

Поняття серіалізації

Сэкономить значительное время можно, снабдив класс UserPrefs атрибутом [Serializable]

После этого полное состояние объекта может быть сохранено с помощью всего нескольких строк кода.

```
static void Main(string[] args)
{
    UserPrefs userData = new UserPrefs();
    userData.WindowColor = "Yellow";
    userData.FontSize = 50;

    // BinaryFormatter сохраняет данные в двоичном формате.
    // Чтобы получить доступ к BinaryFormatter, понадобится
    // импортировать System.Runtime.Serialization.Formatters.Binary.
    BinaryFormatter binFormat = new BinaryFormatter();

    // Сохранить объект в локальном файле.
    using(Stream fStream = new FileStream("user.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, userData);
    }
    Console.ReadLine();
}
```

```
[Serializable]
public class UserPrefs
{
    public string WindowColor;
    public int FontSize;
}
```

когда объект сохраняется в потоке, все ассоциированные с ним данные (т.е. данные базового класса и содержащиеся в нем объекты) также автоматически сериализируются.

- Поэтому при попытке сериализовать производный класс в игру вступают также все данные по цепочке наследования.
- И как будет показано, набор взаимосвязанных объектов, участвующих в этом, представляется с помощью графа объектов.

Службы сериализации .NET также позволяют сохранять граф объектов в различных форматах.

- В предыдущем примере кода применялся тип `BinaryFormatter`, поэтому состояние объекта `UserPrefs` сохраняется в компактном двоичном формате.

Граф объектов можно также сохранить в формате SOAP или XML, применяя другие типы форматов.

- Эти форматы полезны, когда необходимо гарантировать возможность передачи хранимых объектов между разными операционными системами, языками и архитектурами.
- В WCF предлагается слегка отличающийся механизм для сериализации объектов в и из операций служб WCF в нем используются атрибуты `[DataContract]` и `[DataMember]`.

Роль графов объектов

когда объект сериализуется, среда CLR учитывает все связанные объекты, чтобы гарантировать корректное сохранение данных.

- Этот набор связанных объектов называется графом объектов.
- Графы объектов предоставляют простой способ документирования того, как между собой связаны элементы из набора.
- Следует отметить, что графы объектов не обозначают отношения “является” и “имеет” объектно-ориентированного программирования.
- Вместо этого стрелки в графе объектов можно читать как “требуется” или “зависит от”.

Каждый объект в графе получает уникальное числовое значение.

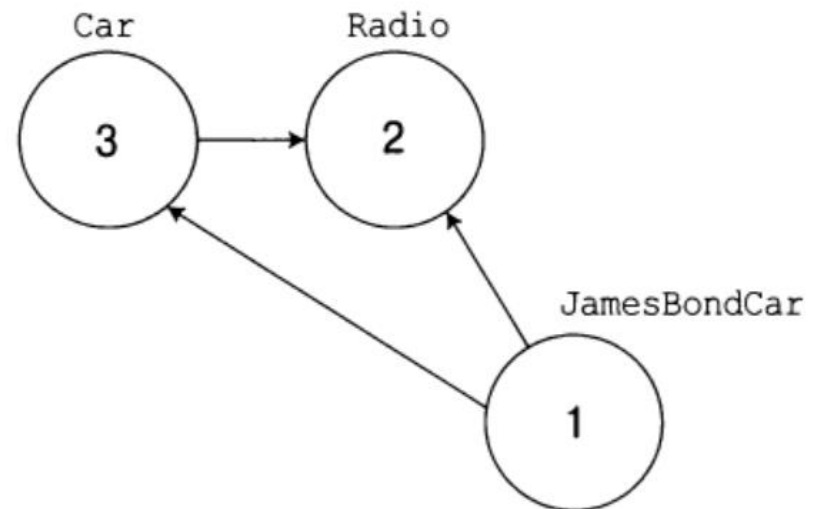
- Имейте в виду, что числа, назначенные объектам в графе, являются произвольными и не имеют никакого значения для внешнего мира.
- После того как всем объектам назначены числовые значения, граф объектов может записывать набор зависимостей для каждого объекта.

В качестве простого примера предположим, что создан набор классов, моделирующих автомобили.

Существует базовый класс по имени Car, который “имеет” класс Radio. Другой класс по имени JamesBondCar расширяет базовый тип Car.

При чтении графов объектов для описания соединяющих стрелок можно использовать выражение “зависит от” или “ссылается на”.

- Класс Car ссылается на класс Radio учитывая отношение “имеет”), JamesBondCar ссылается на Car учитывая отношение “является”), а также на Radio (поскольку наследует эту защищенную переменную-член).



Конечно, среда CLR не рисует картинки в памяти для представления графа связанных объектов.

Вместо этого отношение, документированное в предыдущей диаграмме, представляется формулой, которая выглядит примерно так:

[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]

Удобство процесса сериализации состоит в том, что граф, представляющий отношения между объектами, устанавливается автоматически, “за кулисами”.

- Если необходимо вмешаться в конструирование графа объектов, это можно сделать посредством настройки процесса сериализации через атрибуты и интерфейсы.
- Строго говоря, тип XmlSerializer не сохраняет состояние с использованием графа объектов; тем не менее, он сериализует и десериализует связанные объекты в предсказуемой манере.

Конфигурирование объектов для сериализации

Чтобы сделать объект доступным для служб сериализации .NET, понадобится только декорировать каждый связанный класс (или структуру) атрибутом [Serializable].

Если выясняется, что какой-то тип имеет члены-данные, которые не должны (или не могут) участвовать в сериализации, можно пометить такие поля атрибутом [NonSerialized].

Это помогает сократить размер хранимых данных, при условии, что в сериализируемом классе есть переменные-члены, которые не следует “запоминать” (например, фиксированные значения, случайные значения или кратковременные данные).

Определение сериализируемых типов

Для начала создадим новое консольное приложение по имени SimpleSerialize.

Добавим в него новый класс по имени Radio, помеченный атрибутом [Serializable], у которого исключается одна переменная-член (radioID), помеченная атрибутом [NonSerialized] и потому не сохраняемая в указанном потоке данных.

```
[Serializable]
public class Radio
{
    public bool hasTweeters;
    public bool hasSubWoofers;
    public double[] stationPresets;

    [NonSerialized]
    public string radioID = "XF-552RR6";
}
```

Добавим два дополнительных типа, представляющих классы JamesBondCar и Car (оба они также помечены атрибутом [Serializable]), и определим в них следующие поля данных:

```
[Serializable]
public class Car
{
    public Radio theRadio = new Radio();
    public bool isHatchBack;
}

[Serializable]
public class JamesBondCar : Car
{
    public bool canFly;
    public bool canSubmerge;
}
```

атрибут [Serializable] не может наследоваться от родительского класса.

Поэтому при наследовании типа, помеченного как [Serializable], дочерний класс также должен быть помечен атрибутом [Serializable] или же его нельзя будет сохранить в потоке.

В действительности все объекты в графе объектов должны быть помечены атрибутом [Serializable].

Открытые поля, закрытые поля и открытые свойства

в каждом из предыдущих классов поля данных определены как `public` для упрощения примера.

- закрытые данные, представленные открытыми свойствами, были бы более предпочтительными с точки зрения объектно-ориентированного программирования.
- для простоты в этих типах не определились никакие специальные конструкторы, и потому все неинициализированные поля данных получают ожидаемые стандартные значения.

Какого определения полей данных типа требуют различные форматы, чтобы сериализовать их в поток?

- Если вы сохраняете состояние объекта с применением `BinaryFormatter` или `SoapFormatter`, то разницы никакой.
- Эти типы запрограммированы для сериализации **всех сериализуемых полей** типа независимо от того, представлены они открытыми полями, закрытыми полями или закрытыми полями с соответствующими открытыми свойствами.
- Однако вспомните, что если есть элементы данных, которые не должны сохраняться в графе объектов, можно выборочно пометить открытые или закрытые поля атрибутом `[NonSerialized]`, как сделано со строковым полем в типе `Radio`.

Ситуация существенно меняется, если вы собираетесь использовать тип XmlSerializer.

- Этот тип будет сериализовать только открытые поля данных или закрытые поля, представленные открытыми свойствами.
- Закрытые данные, не представленные свойствами, будут игнорироваться.

```
[Serializable]
public class Person
{
    // Открытое поле.
    public bool isAlive = true;

    // Закрытое поле.
    private int personAge = 21;

    // Открытое свойство/закрытые данные.
    private string fName = string.Empty;
    public string FirstName
    {
        get { return fName; }
        set { fName = value; }
    }
}
```

При обработке этого типа с помощью BinaryFormatter или SoapFormatter обнаружится, что поля isAlive, personAge и fName сохраняются в выбранном потоке.

Однако XmlSerializer не сохранит значение personAge, поскольку эта часть закрытых данных не инкапсулирована в открытом свойстве. Чтобы сохранять personAge с помощью XmlSerializer, это поле понадобится определить как public или же инкапсулировать его в открытом свойстве.

Выбор формatera сериализации

После конфигурирования типов для участия в схеме сериализации .NET за счет применения необходимых атрибутов следующий шаг состоит в выборе формата (двоичного, SOAP или XML) для сохранения состояния объектов.

Перечисленные возможности представлены следующими классами:

- BinaryFormatter
- SoapFormatter
- XmlSerializer

Тип BinaryFormatter сериализирует состояние объекта в поток, используя компактный двоичный формат.

- Этот тип определен в пространстве имен `System.Runtime.Serialization.Formatters.Binary`, которое входит в сборку `mscorlib.dll`.

Таким образом, чтобы получить доступ к этому типу, необходимо указать следующую директиву `using`:

- `// Получить доступ к BinaryFormatter в mscorlib.dll.`
- `using System.Runtime.Serialization.Formatters.Binary;`

Выбор формatera сериализации

Тип `SoapFormatter` сохраняет состояние объекта в виде сообщения SOAP (стандартный XML-формат для передачи и приема сообщений от веб-служб).

- Этот тип определен в пространстве имен `System.Runtime.Serialization.Formatters.Soap`, находящемся в отдельной сборке.
- Поэтому для форматирования графа объектов в сообщение SOAP необходимо сначала установить ссылку на `System.Runtime.Serialization.Formatters.Soap.dll`, используя диалоговое окно `Add Reference` (Добавить ссылку) в `Visual Studio` и затем указать директиву `using`:
- `using System.Runtime.Serialization.Formatters.Soap;`

И, наконец, для сохранения дерева объектов в документе XML предусмотрен тип `XmlSerializer`.

- Чтобы использовать этот тип, нужно указать директиву `using` для пространства имен `System.Xml.Serialization` и установить ссылку на сборку `System.Xml.dll`.
- К счастью, шаблоны проектов `Visual Studio` автоматически ссылаются на `System.Xml.dll`, так что достаточно просто указать соответствующее пространство имен:
- `using System.Xml.Serialization;`

Интерфейсы IFormatter и IRemotingFormatter

Независимо от того, какой формater выбран, имейте в виду, все они унаследованы непосредственно от `System.Object`, так что они не разделяют общий набор членов от какого-то базового класса сериализации.

- Однако типы `BinaryFormatter` и `SoapFormatter` поддерживают общие члены через реализацию интерфейсов `IFormatter` и `IRemotingFormatter` (как ни странно, `XmlSerializer` не реализует ни одного из них).

В интерфейсе `System.Runtime.Serialization.IFormatter` определены основные методы `Serialize()` и `Deserialize()`, которые выполняют черновую работу по перемещению графов объектов в определенный поток и обратно.

- Помимо этих членов в `IFormatter` определено несколько свойств, используемых “за кулисами” реализующим типом

Интерфейс IFormatter

```
public interface IFormatter
{
    SerializationBinder Binder { get; set; }
    StreamingContext Context { get; set; }
    ISurrogateSelector SurrogateSelector { get; set; }
    object Deserialize(Stream serializationStream);
    void Serialize(Stream serializationStream, object graph);
}
```

Интерфейс System.Runtime.Remoting.Messaging.IRemotingFormatter (который внутренне используется уровнем удаленного взаимодействия .NET Remoting) перегружает члены Serialize() и Deserialize() в манере, более подходящей для распределенного хранения.

- Обратите внимание, что интерфейс IRemotingFormatter унаследован от более общего интерфейса IFormatter:

```
public interface IRemotingFormatter : IFormatter
{
    object Deserialize(Stream serializationStream, HeaderHandler handler);
    void Serialize(Stream serializationStream, object graph, Header[] headers);
}
```

Хотя взаимодействовать с этими интерфейсами в большинстве сценариев сериализации не понадобится, вспомните, что полиморфизм на основе интерфейсов позволяет подставлять экземпляры `BinaryFormatter` или `SoapFormatter` там, где ожидается `IFormatter`.

- Таким образом, если необходимо построить метод, который может сериализовать граф объектов с применением любого из этих классов, можно записать так:

```
static void SerializeObjectGraph(IFormatter itfFormat,  
                                Stream destStream, object graph)  
{  
    itfFormat.Serialize(destStream, graph);  
}
```

Точность типов среди форматов

Наиболее очевидное отличие между тремя формateraми связано с тем, **как граф объектов сохраняется в потоке** (двоичном, SOAP или XML).

- Когда используется тип `BinaryFormatter`, он сохраняет не только данные полей объектов из графа, но также полностью заданное имя каждого типа и полное имя определяющей его сборки (имя, версия, маркер открытого ключа и культура).
- Эти дополнительные элементы данных делают `BinaryFormatter` идеальным выбором, когда необходимо передавать объекты по значению (т.е. полные копии) между границами машин для использования в .NET-приложениях.

Форматер `SoapFormatter` сохраняет трассировки сборок-источников за счет применения пространства имен XML.

- Например, вспомните тип `Person`. Если понадобится сохранить этот тип в сообщении SOAP, вы обнаружите, что открывающий элемент `Person` снабжен сгенерированным параметром `xmlns`.

-
- Взгляните на следующее частичное определение, обратив особое внимание на пространство имен XML под названием a1:

```
<a1:Person id="ref-1" xmlns:a1=
  "http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/MyApp%2C%20
  Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
  <isAlive>true</isAlive>
  <personAge>21</personAge>
  <fName id="ref-3">Mel</fName>
</a1:Person>
```

Однако XmlSerializer не старается предохранить точную информацию о типе, поэтому не записывает его полностью заданное имя или сборку, в которой он определен.

Хотя на первый взгляд это может показаться ограничением, причина состоит в открытой природе представления данных XML.

Ниже показано возможное XML-представление типа Person:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <isAlive>true</isAlive>
  <PersonAge>21</PersonAge>
  <FirstName>Frank</FirstName>
</Person>
```

Если необходимо сохранить состояние объекта так, чтобы его можно было использовать в среде любой операционной системы (Windows, Mac OS X и различных дистрибутивах Linux), на любой платформе приложений (.NET, Java Enterprise Edition, COM и т.п.) или в любом языке программирования, придерживаться полной точности типов не следует, поскольку нельзя рассчитывать, что все возможные адресаты смогут понять специфичные для .NET типы данных.

- Учитывая это, SoapFormatter и XmlSerializer являются идеальным выбором, когда требуется гарантировать как можно более широкое распространение дерева объектов.

Сериализация объектов с использованием BinaryFormatter

Чтобы проиллюстрировать, насколько просто сохранять экземпляры JamesBondCar в физическом файле, воспользуемся типом BinaryFormatter.

- Двумя ключевыми методами типа BinaryFormatter являются Serialize () и Deserialize():
- Serialize () сохраняет граф объектов в указанный поток в виде последовательности байтов;
- Deserialize() преобразует сохраненную последовательность байтов в граф объектов.

Предположим, что после создания экземпляра JamesBondCar и модификации некоторых данных состояния требуется сохранить этот экземпляр в файле *.dat.

- Начать следует с создания самого файла *.dat.
- Для этого можно создать экземпляр типа System.IO.FileStream.
- Затем понадобится создать экземпляр BinaryFormatter и передать ему FileStream и граф объектов для сохранения.

```
// Не забудьте импортировать пространства имен  
// System.Runtime.Serialization.Formatters.Binary и System.IO!  
static void Main(string[] args)  
{  
    Console.WriteLine("***** Fun with Object Serialization *****\n");  
    // Создать JamesBondCar и установить состояние.  
    JamesBondCar jbc = new JamesBondCar();  
    jbc.canFly = true;  
    jbc.canSubmerge = false;  
    jbc.theRadio.stationPresets = new double[]{89.3, 105.1, 97.1};  
    jbc.theRadio.hasTweeters = true;  
  
    // Сохранить объект в указанном файле в двоичном формате.  
    SaveAsBinaryFormat(jbc, "CarData.dat");  
    Console.ReadLine();  
}
```

Метод SaveAsBinaryFormat()

```
static void SaveAsBinaryFormat(object objGraph, string fileName)
{
    // Сохранить объект в файл CarData.dat в двоичном виде.
    BinaryFormatter binFormat = new BinaryFormatter();
    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in binary format!");
}
```

метод `BinaryFormatter.Serialize()` — это член, отвечающий за построение графа объектов и передачу последовательности байтов в некоторый объект производного от `Stream` типа.

- В данном случае поток представляет физический файл.
- Однако сериализовать объекты можно также в любой тип, производный от `Stream`, такой как область памяти или сетевой поток.

После выполнения программы
можно просмотреть содержимое
файла CarData.dat

CarData.dat	CarCollection.xml	CarData.soap
00000000	00 01 00 00 00 FF FF FF	FF 01 00 00 00 00 00 00
00000010	00 0C 02 00 00 00 46 53	69 6D 70 6C 65 53 65 72FSimpleSer
00000020	69 61 6C 69 7A 65 2C 20	56 65 72 73 69 6F 6E 3D ialize, Version=
00000030	31 2E 30 2E 30 2E 30 2C	20 43 75 6C 74 75 72 65 1.0.0.0, Culture
00000040	3D 6E 65 75 74 72 61 6C	2C 20 50 75 62 6C 69 63 =neutral, Public
00000050	4B 65 79 54 6F 6B 65 6E	3D 6E 75 6C 6C 05 01 00 KeyToken=null...
00000060	00 00 1C 53 69 6D 70 6C	65 53 65 72 69 61 6C 69 ...SimpleSeriali
00000070	7A 65 2E 4A 61 6D 65 73	42 6F 6E 64 43 61 72 04 ze.JamesBondCar.
00000080	00 00 00 06 63 61 6E 46	6C 79 0B 63 61 6E 53 75canFly.canSu
00000090	62 6D 65 72 67 65 08 74	68 65 52 61 64 69 6F 0B bmerge.theRadio.
000000a0	69 73 48 61 74 63 68 42	61 63 6B 00 00 04 00 01 isHatchBack.....
000000b0	01 15 53 69 6D 70 6C 65	53 65 72 69 61 6C 69 7A ..SimpleSerializ
000000c0	65 2E 52 61 64 69 6F 02	00 00 00 01 02 00 00 00 e.Radio.....
000000d0	01 00 09 03 00 00 00 00	05 03 00 00 00 15 53 69
000000e0	6D 70 6C 65 53 65 72 69	61 6C 69 7A 65 2E 52 61 mpleSerialize.Ra
000000f0	64 69 6F 03 00 00 00 0B	68 61 73 54 77 65 65 74 dio.....hasTweet
00000100	65 72 73 0D 68 61 73 53	75 62 57 6F 6F 66 65 72 ers.hasSubWoofers
00000110	73 0E 73 74 61 74 69 6F	6E 50 72 65 73 65 74 73 s.stationPresets
00000120	00 00 07 01 01 06 02 00	00 00 01 00 09 04 00 00
00000130	00 0F 04 00 00 00 03 00	00 00 06 33 33 33 33 33
00000140	53 56 40 66 66 66 66 66	46 5A 40 66 66 66 66 66 SV@ffffffFZ@fffff
00000150	46 58 40 0B	FX@.

Десериализация объектов с использованием BinaryFormatter

Теперь предположим, что необходимо прочитать сохраненный объект JamesBondCar из двоичного файла обратно в объектную переменную.

- После открытия файла CataData.dat (посредством метода File.OpenRead()) можно вызвать метод Deserialize () класса BinaryFormatter.
- Имейте в виду, что Deserialize () возвращает объект общего типа System.Object, так что понадобится применить явное приведение

```
static void LoadFromBinaryFile(string fileName)
{
    BinaryFormatter binFormat = new BinaryFormatter();
    // Прочитать JamesBondCar из двоичного файла.
    using(Stream fStream = File.OpenRead(fileName))
    {
        JamesBondCar carFromDisk =
            (JamesBondCar)binFormat.Deserialize(fStream);
        Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
    }
}
```

Сериализация объектов с использованием SoapFormatter

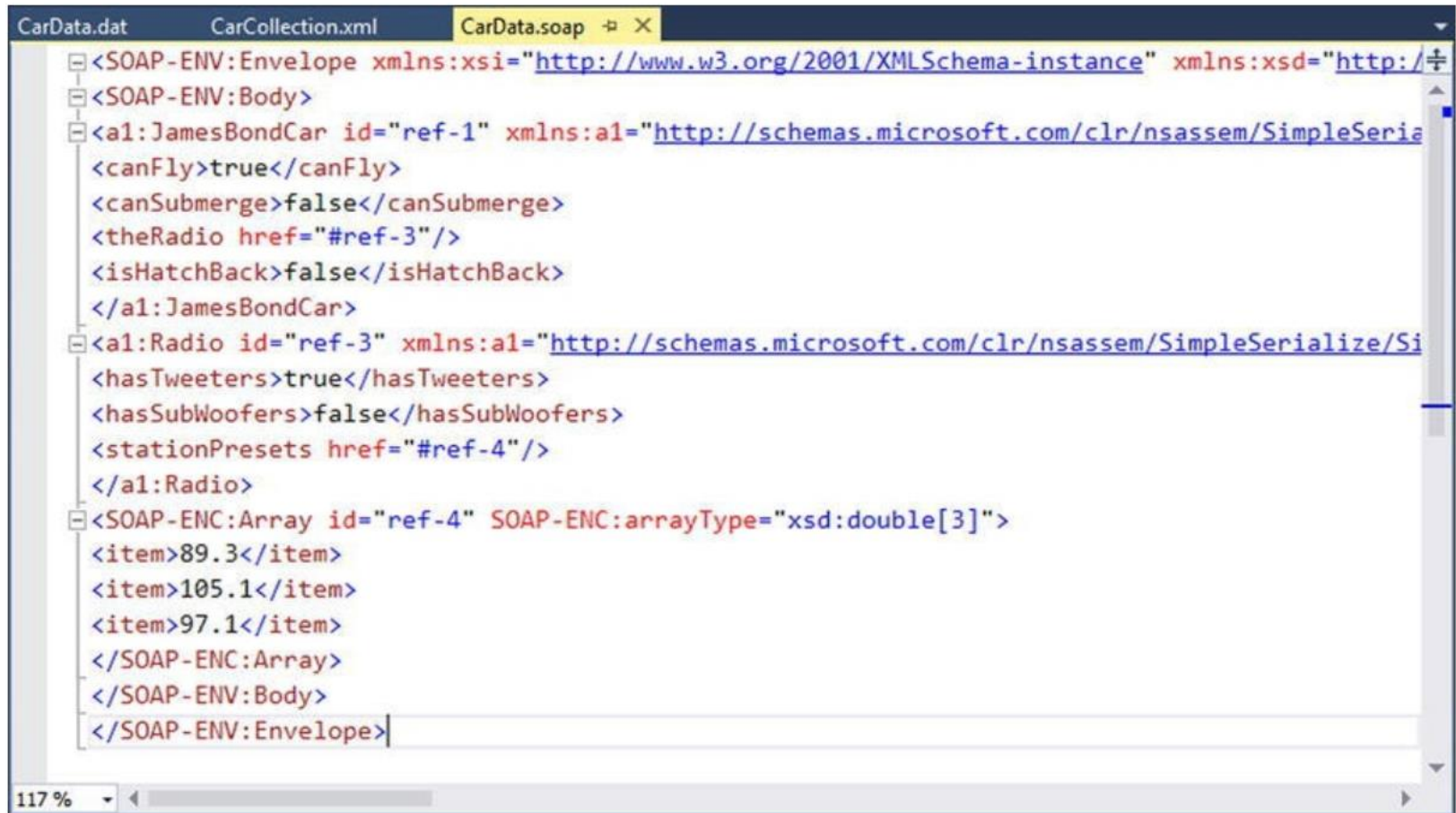
Протокол SOAP (Simple Object Access Protocol — простой протокол доступа к объектам) определяет стандартный процесс вызова методов в независимой от платформы и операционной системы манере.

- ссылка на сборку System.Runtime.Serialization.Formatters.Soap.dll установлена, а пространство имен System.Runtime.Serialization.Formatters.Soap импортировано, для сохранения и извлечения JamesBondCar в виде сообщения SOAP можно просто заменить в предыдущем примере все вхождения BinaryFormatter на SoapFormatter.

```
// Не забудьте импортировать пространства имен
// System.Runtime.Serialization.Formatters.Soap
// и установить ссылку на System.Runtime.Serialization.Formatters.Soap.dll!
static void SaveAsSoapFormat (object objGraph, string fileName)
{
    // Сохранить объект в файле CarData.soap в формате SOAP.
    SoapFormatter soapFormat = new SoapFormatter();

    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        soapFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in SOAP format!");
}
```


Как и ранее, для перемещения графа объектов в поток и обратно применяются методы `Serialize()` и `Deserialize()`



```
CarData.dat  CarCollection.xml  CarData.soap  X
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org/2001/XMLSchema-instance">
  <SOAP-ENV:Body>
    <a1:JamesBondCar id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/SimpleSerialize" >
      <canFly>true</canFly>
      <canSubmerge>false</canSubmerge>
      <theRadio href="#ref-3"/>
      <isHatchBack>false</isHatchBack>
    </a1:JamesBondCar>
    <a1:Radio id="ref-3" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/SimpleSerialize" >
      <hasTweeters>true</hasTweeters>
      <hasSubWoofers>false</hasSubWoofers>
      <stationPresets href="#ref-4"/>
    </a1:Radio>
    <SOAP-ENC:Array id="ref-4" SOAP-ENC:arrayType="xsd:double[3]">
      <item>89.3</item>
      <item>105.1</item>
      <item>97.1</item>
    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

В файле находятся XML-элементы, которые описывают значения состояния текущего объекта `JamesBondCar`, а также отношения между объектами в графе, представленные с помощью лексем `#ref`

Сериализация объектов с использованием XmlSerializer

Этот формater может использоваться для сохранения открытого состояния заданного объекта в виде чистой XML-разметки, в противоположность данным XML внутри сообщения SOAP.

- Работа с этим типом несколько отличается от работы с типами SoapFormatter или BinaryFormatter.
- Рассмотрим следующий код (в нем предполагается, что было импортировано пространство имен System.Xml.Serialization):

```
static void SaveAsXmlFormat(object objGraph, string fileName)
{
    // Сохранить объект в файле CarData.xml в формате XML.
    XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar),
        new Type[] { typeof(Radio), typeof(Car) });
    using(Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        xmlFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in XML format!");
}
```

Ключевое отличие состоит в том, что тип XmlSerializer требует указания информации о типе, представляющей класс, который необходимо сериализировать. В сгенерированном файле XML находятся показанные ниже данные XML:

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <theRadio>
    <hasTweeters>true</hasTweeters>
    <hasSubWoofers>false</hasSubWoofers>
    <stationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
    </stationPresets>
    <radioID>XF-552RR6</radioID>
  </theRadio>
  <isHatchBack>false</isHatchBack>
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Управление генерацией данных XML

Если у вас есть опыт работы с технологиями XML, то вы хорошо знаете, насколько важно удостовериться, что данные внутри документа XML отвечают набору правил, которые устанавливают действительность данных. Понятие действительного документа XML не имеет отношения к синтаксической правильности элементов XML (наподобие того, что все открывающие элементы должны иметь соответствующие закрывающие элементы).

Действительные документы — это те, что отвечают согласованным правилам форматирования (например, поле X должно быть выражено как атрибут, но не как подэлемент), которые обычно определены в схеме XML или файле определения типа документа (Document-type Definition — DTD).

По умолчанию класс `XmlSerializer` сериализует все открытые поля/свойства как элементы XML, а не как атрибуты XML. Чтобы управлять генерацией результирующего документа XML с помощью класса `XmlSerializer`, необходимо декорировать типы любым количеством дополнительных атрибутов из пространства имен `System.Xml.Serialization`.

Избранные атрибуты из пространства имен System.Xml.Serialization

Атрибут .NET	Описание
[XmlAttribute]	Этот атрибут .NET можно применять к полю или свойству для того, чтобы сообщить XmlSerializer о необходимости сериализовать данные как атрибут XML (а не как подэлемент)
[XmlElement]	Поле или свойство будет сериализовано как элемент XML с указанным именем
[XmlAttribute]	Этот атрибут предоставляет имя элемента, являющееся членом перечисления
[XmlRoot]	Этот атрибут управляет тем, как будет сконструирован корневой элемент (пространство имен и имя элемента)
[XmlText]	Свойство или поле будет сериализовано как текст XML (т.е. содержимое, находящееся между начальным и конечным дескрипторами корневого элемента)
[XmlType]	Этот атрибут предоставляет имя и пространство имен типа XML

В простом примере показано текущее представление данных полей JamesBondCar в XML

```
<?xml version="1.0" encoding="utf-8"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    ...
    <canFly>true</canFly>
    <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Если необходимо указать специальное пространство имен XML, которое уточняет JamesBondCar и кодирует значения canFly и canSubmerge в виде атрибутов XML, модифицируем определение класса JamesBondCar:

```
[Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
    [XmlAttribute]
    public bool canFly;
    [XmlAttribute]
    public bool canSubmerge;
}
```

Это выдает показанный ниже документ XML (обратите внимание на открывающий элемент <JamesBondCar>):

```
<?xml version="1.0" ""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xmlns:xsd="http://www.w3.org/2001/XMLSchema"
               canFly="true" canSubmerge="false"
               xmlns="http://www.MyCompany.com">
...
</JamesBondCar>
```

Естественно, для управления генерацией результирующего XML-документа с помощью XmlSerializer могут применяться многие другие атрибуты.

- За подробной информацией обращайтесь к описанию пространства имен System.Xml.Serialization в документации .NET Framework

Сериализация коллекций объектов

метод `Serialize()` интерфейса `IFormatter` не предоставляет способа указать произвольное количество объектов в качестве ввода (допускается только единственный объект `System.Object`).

- Вдобавок возвращаемое значение `Deserialize()` также представляет собой одиночный объект `System.Object` (то же базовое ограничение касается и `XmlSerializer`):

```
public interface IFormatter
{
    ...
    object Deserialize(Stream serializationStream);
    void Serialize(Stream serializationStream, object graph);
}
```

Вспомните, что `System.Object` представляет целое дерево объектов.

- С учетом этого, если передать объект, который помечен атрибутом `[Serializable]` и содержит в себе другие объекты `[Serializable]`, то с помощью единственного вызова данного метода будет сохраняться весь набор объектов.
- К счастью, большинство типов из пространств имен `System.Collections` и `System.Collections.Generic` уже помечены атрибутом `[Serializable]`.
- Таким образом, чтобы сохранить множество объектов, просто добавьте это множество в контейнер (такой как обычный массив, `ArrayList` или `List<T>`) и сериализуйте данный объект в выбранный поток.

Предположим, что класс JamesBondCar дополнен конструктором, принимающим два аргумента, для установки нескольких фрагментов данных состояния (обратите внимание, что должен быть также добавлен стандартный конструктор, как того требует XmlSerializer):

```
[Serializable, XmlRoot(Namespace = "http://www.MyCompany.com")]
public class JamesBondCar : Car
{
    public JamesBondCar(bool skyWorthy, bool seaWorthy)
    {
        canFly = skyWorthy;
        canSubmerge = seaWorthy;
    }
    // XmlSerializer требует стандартного конструктора!
    public JamesBondCar() {}
    ...
}
```

Теперь можно сохранять любое количество объектов JamesBondCar

```
static void SaveListOfCars()
{
    // Сохранить список List<T> объектов JamesBondCar.
    List<JamesBondCar> myCars = new List<JamesBondCar>();
    myCars.Add(new JamesBondCar(true, true));
    myCars.Add(new JamesBondCar(true, false));
    myCars.Add(new JamesBondCar(false, true));
    myCars.Add(new JamesBondCar(false, false));
    using(Stream fStream = new FileStream("CarCollection.xml",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        XmlSerializer xmlFormat = new XmlSerializer(typeof(List<JamesBondCar>));
        xmlFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars!");
}
```

Поскольку здесь используется XmlSerializer, необходимо указать информацию о типе для каждого из подобъектов внутри корневого объекта (которым в данном случае является List<JamesBondCar>).

- Если бы применялись типы BinaryFormatter или SoapFormatter, то логика была бы еще проще.

```
static void SaveListOfCarsAsBinary()
{
    // Сохранить объект ArrayList (myCars) в двоичном виде.
    List<JamesBondCar> myCars = new List<JamesBondCar>();

    BinaryFormatter binFormat = new BinaryFormatter();
    using(Stream fStream = new FileStream("AllMyCars.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars in binary!");
}
```

Настройка процессов сериализации SOAP и двоичной сериализации

В большинстве случаев стандартная схема сериализации, предоставляемая платформой .NET, вполне подходит.

- Нужно лишь применить атрибут [Serializable] к связанным типам и передать дерево объектов выбранному формату для обработки.

Однако в некоторых случаях может понадобиться вмешательство в процесс конструирования дерева и процесс сериализации.

- Например, может существовать бизнес-правило, которое гласит, что все поля данных должны сохраняться в определенном формате, или же необходимо добавить дополнительные данные в поток, которые напрямую не отображаются на поля сохраняемого объекта (скажем, временные метки или уникальные идентификаторы).

Основные типы пространства имен System.Runtime.Serialization

Тип	Описание
ISerializable	Этот интерфейс может быть реализован типом [Serializable] для управления его сериализацией и десериализацией
ObjectIDGenerator	Этот тип генерирует идентификаторы для членов в графе объектов
[OnDeserialized]	Этот атрибут позволяет указать метод, который будет вызван немедленно после десериализации объекта
[OnDeserializing]	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса десериализации
[OnSerialized]	Этот атрибут позволяет указать метод, который будет вызван немедленно после того, как объект сериализирован
[OnSerializing]	Этот атрибут позволяет указать метод, который будет вызван перед началом процесса сериализации
[OptionalField]	Этот атрибут позволяет определить поле типа, которое может быть пропущено в указанном потоке
SerializationInfo	В сущности, этот класс является <i>пакетом свойств</i> , который поддерживает пары “имя/значение”, представляющие состояние объекта во время процесса сериализации

Углубленный взгляд на сериализацию объектов

Когда `BinaryFormatter` сериализирует граф объектов, он отвечает за передачу следующей информации в указанный поток:

- полностью заданное имя объекта в графе (например, `MyApp.JamesBondCar`);
- имя сборки, определяющей граф объектов (например, `MyApp.exe`);
- экземпляр класса `SerializationInfo`, содержащего все данные состояния, которые поддерживаются членами графа объектов.

Во время процесса десериализации `BinaryFormatter` использует ту же самую информацию для построения идентичной копии объекта с применением данных, извлеченных из потока-источника.

- Процесс, выполняемый `SoapFormatter`, очень похож.

Вспомните, что для обеспечения максимальной мобильности объекта форматор `XmlSerializer` не сохраняет полностью заданное имя типа или имя сборки, в которой он содержится.

- Этот тип может сохранять только открытые данные.

Помимо перемещения необходимых данных в поток и обратно, формтеры также анализируют члены графа объектов на предмет перечисленных ниже частей инфраструктуры.

- Проверка пометки объекта атрибутом [Serializable]. Если объект не помечен, генерируется исключение `SerializationException`.
- Если объект помечен атрибутом [Serializable], производится проверка, реализует ли объект интерфейс `ISerializable`. Если да, на этом объекте вызывается метод `GetObjectData()`.
- Если объект не реализует интерфейс `ISerializable`, используется стандартный процесс сериализации, который обрабатывает все поля, не помеченные как [NonSerialized].

В дополнение к определению того, поддерживает ли тип интерфейс `ISerializable`, формтеры также отвечают за исследование типов на предмет поддержки членов, которые оснащены атрибутами [OnSerializing], [OnSerialized], [OnDeserializing] или [OnDeserialized].

- Мы рассмотрим назначение этих атрибутов чуть позже, а сначала давайте посмотрим на предназначение `ISerializable`.

Настройка сериализации с использованием ISerializable

Для объектов, которые помечены атрибутом [Serializable], имеется возможность реализации интерфейса ISerializable.

- Реализация этого интерфейса позволяет вмешаться в процесс сериализации и выполнить необходимое форматирование данных до и после сериализации.
- После выхода версии .NET 2.0 предпочтительный способ настройки процесса сериализации стал предусматривать применение атрибутов сериализации. Тем не менее, знание интерфейса ISerializable важно для сопровождения систем, которые уже существуют.

Интерфейс ISerializable довольно прост, учитывая, что в нем определен единственный метод GetObjectData():

```
// Чтобы получить возможность настройки процесса сериализации,  
// необходимо реализовать ISerializable.  
public interface ISerializable  
{  
    void GetObjectData(SerializationInfo info,  
        StreamingContext context);  
}
```

Метод `GetObjectData ()` вызывается автоматически заданным форматером во время процесса сериализации.

- Реализация этого метода заполняет входной параметр `SerializationInfo` последовательностью пар "Имя/значение", которые (обычно) отображают данные полей сохраняемого объекта.
- В `SerializationInfo` определены многочисленные вариации перегруженного метода `AddValue ()`, а также небольшой набор свойств, которые позволяют устанавливать и получать имя типа, определять сборку и счетчик членов.

```
public sealed class SerializationInfo
{
    public SerializationInfo(Type type, IFormatterConverter converter);
    public string AssemblyName { get; set; }
    public string FullTypeName { get; set; }
    public int MemberCount { get; }
    public void AddValue(string name, short value);
    public void AddValue(string name, ushort value);
    public void AddValue(string name, int value);
    ...
}
```

Типы, реализующие интерфейс `ISerializable`, также должны определять специальный конструктор со следующей сигнатурой:

```
// Необходимо предоставить специальный конструктор с такой сигнатурой,  
// чтобы позволить исполняющей среде устанавливать состояние объекта.  
[Serializable]  
class SomeClass : ISerializable  
{  
    protected SomeClass (SerializationInfo si, StreamingContext ctx) {...}  
    ...  
}
```

Обратите внимание, что видимость конструктора указана как `protected`.

- Это разрешено, поскольку формater будет иметь доступ к этому члену независимо от его видимости.
- Такие специальные конструкторы обычно определяются как `protected` (или `private`), гарантируя тем самым, что небрежный пользователь объекта никогда не создаст объект с их помощью.
- Первым параметром конструктора является экземпляр типа `SerializationInfo` (который был показан ранее).

Второй параметр этого специального конструктора имеет тип `StreamingContext` и содержит информацию относительно источника или места назначения передаваемых данных.

- Наиболее информативным членом `StreamingContext` является свойство `State`, которое хранит значение из перечисления `StreamingContextStates`.
- Значения этого перечисления представляют базовую композицию текущего потока.
- Если только не планируется реализовать какие-то низкоуровневые службы удаленного взаимодействия, то иметь дело с этим перечислением напрямую придется редко.

Тем не менее, ниже приведены члены перечисления `StreamingContextStates` (за более подробной информацией обращайтесь в документацию `.NET Framework 4.5 SDK`):

```
public enum StreamingContextStates
{
    CrossProcess,
    CrossMachine,
    File,
    Persistence,
    Remoting,
    Other,
    Clone,
    CrossAppDomain,
    All
}
```


пример настройки процесса сериализации с использованием ISerializable

- Представим, что требуется обеспечить сериализацию объектов string в верхнем регистре, а десериализацию — в нижнем. Для удовлетворения этих правил можно реализовать интерфейс ISerializable:

```
[Serializable]
class StringData : ISerializable
{
    private string dataItemOne = "First data block";
    private string dataItemTwo = "More data";

    public StringData() {}
    protected StringData(SerializationInfo si, StreamingContext ctx)
    {
        // Восстановить переменные-члены из потока.
        dataItemOne = si.GetString("First_Item").ToLower();
        dataItemTwo = si.GetString("dataItemTwo").ToLower();
    }

    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext ctx)
    {
        // Наполнить объект SerializationInfo форматированными данными.
        info.AddValue("First_Item", dataItemOne.ToUpper());
        info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
    }
}
```

Обратите внимание, что при наполнении объекта типа `SerializationInfo` внутри метода `GetObjectData()` именовать элементы данных идентично именам внутренних переменных-членов типа не обязательно.

- Это очевидно полезно, если требуется отвязать данные типа от формата хранения.
- Однако имейте в виду, что получать значения в специальном защищенном конструкторе необходимо с указанием тех же самых имен, что были назначены внутри `GetObjectData()`.
- Чтобы опробовать специализированную сериализацию, предположим, что экземпляр `MyStringData` сохраняется с применением `SoapFormatter` (обновите соответствующим образом ссылки на сборки и директивы `using`):

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Serialization *****");

    // Вспомните, что этот тип реализует ISerializable.
    StringData myData = new StringData();

    // Сохранить в локальный файл в формате SOAP.
    SoapFormatter soapFormat = new SoapFormatter();
    using(Stream fStream = new FileStream("MyData.soap",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        soapFormat.Serialize(fStream, myData);
    }
    Console.ReadLine();
}
```

Просматривая полученный файл *.soap, вы заметите, что строковые поля действительно сохранены в верхнем регистре

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-  
instance"  
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"  
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"  
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"  
  xmlns:clr="http://schemas.microsoft.com/soap/encoding clr/1.0"  
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">  
<SOAP-ENV:Body>  
  
  <a1:StringData id="ref-1" ...>  
    <First_Item id="ref-3">FIRST DATA BLOCK</First_Item>  
    <dataItemTwo id="ref-4">MORE DATA</dataItemTwo>  
  </a1:StringData>  
</SOAP-ENV:Body>  
  
</SOAP-ENV:Envelope>
```

Настройка сериализации с использованием атрибутов

Хотя реализация интерфейса `ISerializable` является одним из возможных способов настройки процесса сериализации, с момента выхода версии .NET 2.0 предпочтительным способом такой настройки стало определение методов, оснащенных атрибутами из следующего перечня: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` и `[OnDeserialized]`.

Использование этих атрибутов дает менее громоздкий код, чем реализация интерфейса `ISerializable`, учитывая, что не приходится вручную взаимодействовать с параметром `SerializationInfo`.

Вместо этого можно напрямую модифицировать данные состояния, когда формater работает с типом.

На заметку! Эти атрибуты сериализации определены в пространстве имен `System.Runtime.Serialization`.

В случае применения этих атрибутов метод должен быть определен так, чтобы принимать параметр `StreamingContext` и не возвращать ничего (иначе будет сгенерировано исключение времени выполнения).

Обратите внимание, что применять каждый из атрибутов сериализации не обязательно, а можно просто вмешиваться в те стадии процесса сериализации, которые интересуют.

В целях иллюстрации ниже приведен новый тип `[Serializable]`, который имеет те же самые требования, что и `StringData`, но на этот раз он полагается на использование атрибутов `[OnSerializing]` и `[OnDeserialized]`:

```
[Serializable]
class MoreData
{
    private string dataItemOne = "First data block";
    private string dataItemTwo = "More data";

    [OnSerializing]
    private void OnSerializing(StreamingContext context)
    {
        // Вызывается во время процесса сериализации.
        dataItemOne = dataItemOne.ToUpper();
        dataItemTwo = dataItemTwo.ToUpper();
    }

    [OnDeserialized]
    private void OnDeserialized(StreamingContext context)
    {
        // Вызывается по завершении процесса десериализации.
        dataItemOne = dataItemOne.ToLower();
        dataItemTwo = dataItemTwo.ToLower();
    }
}
```

Выполнив сериализацию этого нового типа, вы снова обнаружите, что данные сохраняются в верхнем регистре, а десериализируются — в нижнем.

Дякую за увагу!
