

ГЛАВА 13. РАБОТА С ПОТОКАМИ И ФАЙЛОВОЙ СИСТЕМОЙ

Оглавление

§13.7 XML-документы.....	1
§13.8 Сериализация и десериализация.....	12

§13.7 XML-документы

На сегодняшний день XML является одним из распространенных стандартов документов, который позволяет в удобной форме сохранять сложные по структуре данные. Поэтому разработчики платформы .NET включили в фреймворк широкие возможности для работы с XML.

Прежде чем перейти непосредственно к работе с XML-файлами, сначала рассмотрим, что представляет собой xml-документ и как он может хранить объекты, используемые в программе на C#.

Например, у нас есть следующий класс:

Листинг 13.17. Класс «Пользователь»

1	<code>class User</code>
2	<code>{</code>
3	<code> public string Name { get; set; }</code>
4	<code> public int Age { get; set; }</code>
5	<code> public string Company { get; set; }</code>
6	<code>}</code>

В программе на C# мы можем создать список объектов класса User:

Листинг 13.18. Список List объектов пользователей

1	<code>User user1 = new User { Name = "Bill Gates", Age = 48, Company = "Microsoft" }</code>
2	<code>User user2 = new User { Name = "Larry Page", Age = 42, Company = "Google" }</code>
3	<code>List<User> users = new List<User> { user1, user2 }</code>

Чтобы сохранить список в формате xml мы могли бы использовать следующий xml-файл:

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <users>
3   <user name="Bill Gates">
4     <company>Microsoft</company>
5     <age>48</age>
6   </user>
7   <user name="Larry Page">
8     <company>Google</company>
9     <age>48</age>
10  </user>
11 </users>
```

Рисунок 13.1 – Представление списка пользователей через xml-документ

XML-документ объявляет строка `<?xml version="1.0" encoding="utf-8" ?>`. Она задает версию (1.0) и кодировку (utf-8) xml. Далее идет собственно содержимое документа.

XML-документ должен иметь один единственный корневой элемент, внутри которого помещаются все остальные элементы. В данном случае таким элементом является элемент `<users>`. Внутри корневого элемента `<users>` задан набор элементов `<user>`. Вне корневого элемента мы не можем разместить элементы `user`.

Каждый элемент определяется с помощью открывающего и закрывающего тегов, например, `<user>` и `</user>`, внутри которых помещается значение или содержимое элементов. Также элемент может иметь сокращенное объявление: `<user/>` - в конце элемента помещается слеш.

Элемент может иметь вложенные элементы и атрибуты. В данном случае каждый элемент `user` имеет два вложенных элемента `company` и `age` и атрибут `name`.

Атрибуты определяются в теле элемента и имеют следующую форму: **название="значение"**. Например, `<user name="Bill Gates">`, в данном случае атрибут называется `name` и имеет значение `Bill Gates`

Внутри простых элементов помещается их значение. Например,

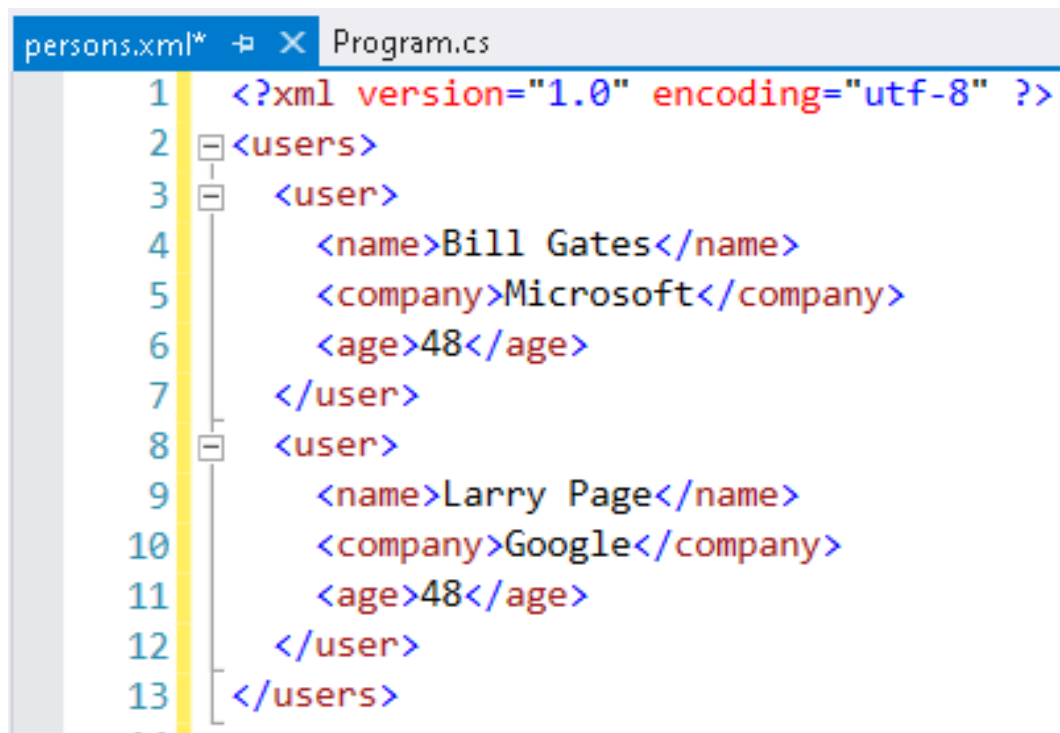
`<company>Google</company>`

- элемент `company` имеет значение `Google`.

Названия элементов являются регистрозависимыми, поэтому `<company>` и `<COMPANY>` будут представлять разные элементы.

Таким образом, весь список `Users` из кода `C#` сопоставляется с корневым элементом `<users>`, каждый объект `User` - с элементом `<user>`, а каждое свойство объекта `User` - с атрибутом или вложенным элементом элемента `<user>`.

Что использовать для свойств - вложенные элементы или атрибуты? Это вопрос предпочтений - мы можем использовать как атрибуты, так и вложенные элементы. Так, в предыдущем примере вполне можно использовать вместо атрибута вложенный элемент:



```
persons.xml* X Program.cs
1  <?xml version="1.0" encoding="utf-8" ?>
2  <users>
3    <user>
4      <name>Bill Gates</name>
5      <company>Microsoft</company>
6      <age>48</age>
7    </user>
8    <user>
9      <name>Larry Page</name>
10     <company>Google</company>
11     <age>48</age>
12   </user>
13 </users>
```

Рисунок 13.2 - Представление списка пользователей через *xml*-документ

Работа с XML с помощью классов System.Xml

Для работы с XML в `C#` можно использовать несколько подходов. В первых версиях фреймворка основной функционал работы с XML

предоставляло пространство имен `System.Xml`. В нем определен ряд классов, которые позволяют манипулировать xml-документом:

Таблица 13.1 – Классы пространства имен System.Xml

Класс	Описание класса
<code>XmlNode</code>	представляет узел xml. В качестве узла может использоваться весь документ, так и отдельный элемент
<code>XmlDocument</code>	представляет весь xml-документ
<code>XmlElement</code>	представляет отдельный элемент. Наследуется от класса <code>XmlNode</code>
<code>XmlAttribute</code>	представляет атрибут элемента
<code>XmlText</code>	представляет значение элемента в виде текста, то есть тот текст, который находится в элементе между его открывающим и закрывающим тегами
<code>XmlComment</code>	представляет комментарий в xml
<code>XmlNodeList</code>	используется для работы со списком узлов

Ключевым классом, который позволяет манипулировать содержимым xml, является `XmlNode`, поэтому рассмотрим некоторые его основные методы и свойства:

Таблица 13.2 – Свойства класса XmlNode

Свойства	Описание
Свойство <code>Attributes</code>	возвращает объект <code>XmlAttributeCollection</code> , который представляет коллекцию атрибутов
Свойство <code>ChildNodes</code>	возвращает коллекцию дочерних узлов для данного узла
Свойство <code>HasChildNodes</code>	возвращает <code>true</code> , если текущий узел имеет дочерние узлы
Свойство <code>FirstChild</code>	возвращает первый дочерний узел
Свойство <code>LastChild</code>	возвращает последний дочерний узел

Свойство InnerText	возвращает текстовое значение узла
Свойство InnerXml	возвращает всю внутреннюю разметку xml узла
Свойство Name	возвращает название узла. Например, <user> - значение свойства Name равно "user"
Свойство ParentNode	возвращает родительский узел у текущего узла

Применим эти классы и их функционал. И вначале для работы с xml создадим новый файл. Назовем его *persons.xml* и определим в нем следующее содержание:

```

persons.xml*  Program.cs
1  <?xml version="1.0" encoding="utf-8" ?>
2  <users>
3    <user name="Bill Gates">
4      <company>Microsoft</company>
5      <age>48</age>
6    </user>
7    <user name="Larry Page">
8      <company>Google</company>
9      <age>42</age>
10   </user>
11 </users>

```

Рисунок 13.3 – Файл *users.xml*

Теперь пройдемся по этому документу и выведем его данные на консоль:

Листинг 13.19. Чтение данных из xml-файла

1	using System.Xml;
2	class Program
3	{
4	static void Main(string[] args)
5	{
6	XmlDocument xDoc = new XmlDocument();
7	xDoc.Load("persons.xml");
8	// получим корневой элемент
9	XmlElement xRoot = xDoc.DocumentElement;
10	// обход всех узлов в корневом элементе

11	foreach(XmlNode xnode in xRoot)
12	{
13	// получаем атрибут name
14	if(xnode.Attributes.Count>0)
15	{
16	XmlNode attr = xnode.Attributes.GetNamedItem("name");
17	if (attr!=null)
18	Console.WriteLine(attr.Value);
19	}
20	// обходим все дочерние узлы элемента user
21	foreach(XmlNode childnode in xnode.ChildNodes)
22	{
23	// если узел - company
24	if(childnode.Name=="company")
25	{
26	Console.WriteLine(\$"Компания: {childnode.InnerText}");
27	}
28	// если узел age
29	if (childnode.Name == "age")
30	{
31	Console.WriteLine(\$"Возраст: {childnode.InnerText}");
32	}
33	}
34	Console.WriteLine();
35	}
36	Console.Read();
37	}
38	}

Чтобы начать работу с документом xml, нам надо создать объект `XmlDocument` и затем загрузить в него xml-файл:

```
xDoc.Load("persons.xml");
```

При разборе xml для начала мы получаем корневой элемент документа с помощью свойства `xDoc.DocumentElement`. Далее уже происходит собственно разбор узлов документа.

В цикле

```
foreach(XmlNode xnode in xRoot)
```

пробегаемся по всем дочерним узлам корневого элемента. Так как дочерние узлы представляют элементы `<user>`, то мы можем получить их атрибуты:

```
XmlNode attr = xnode.Attributes.GetNamedItem("name");
```

и вложенные элементы:

```
foreach(XmlNode childnode in xnode.ChildNodes)
```

Чтобы определить, что за узел перед нами, мы можем сравнить его название:

```
if(childnode.Name=="company")
```

Подобным образом мы можем создать объекты User по данным из xml:

Листинг 13.20. Запись данных в объект из xml-файл

1	using System;
2	using System.Collections.Generic;
3	using System.Xml;
4	namespace HelloApp
5	{
6	class User
7	{
8	public string Name { get; set; }
9	public int Age { get; set; }
10	public string Company { get; set; }
11	}
12	class Program
13	{
14	static void Main(string[] args)
15	{
16	List<User> users = new List<User>();
17	XmlDocument xDoc = new XmlDocument();
18	xDoc.Load("persons.xml");
19	XmlElement xRoot = xDoc.DocumentElement;
20	foreach (XmlElement xnode in xRoot)
21	{
22	User user = new User();
23	XmlNode attr = xnode.Attributes.GetNamedItem("name");
24	if (attr != null)
25	user.Name = attr.Value;
26	foreach (XmlNode childnode in xnode.ChildNodes)
27	{
28	if (childnode.Name == "company")
29	user.Company = childnode.InnerText;
30	if (childnode.Name == "age")

31	user.Age = Int32.Parse(childnode.InnerText);
32	}
33	users.Add(user);
34	}
35	foreach (User u in users)
36	Console.WriteLine(\$"{u.Name} ({u.Company}) - {u.Age}");
37	Console.Read();
38	}
39	}
40	}

Изменение XML-документа

Для редактирования xml-документа (изменения, добавления, удаления элементов) мы можем воспользоваться методами класса XmlNode:

Таблица 13.3 – Методы класса XmlNode

Метод	Описание
AppendChild	добавляет в конец текущего узла новый дочерний узел
InsertAfter	добавляет новый узел после определенного узла
InsertBefore	добавляет новый узел до определенного узла
RemoveAll	удаляет все дочерние узлы текущего узла
RemoveChild	удаляет у текущего узла один дочерний узел и возвращает его

Класс XmlDocument добавляет еще ряд методов, которые позволяют создавать новые узлы:

Таблица 13.4 – Методы класса XmlDocument

Метод	Описание
CreateNode	создает узел любого типа
CreateElement	создает узел типа XmlDocument

CreateAttribute	создает узел типа XmlAttribute
CreateTextNode	создает узел типа XmlTextNode
CreateComment	создает комментарий

Возьмем xml-документ из прошлой темы и добавим в него новый элемент:

Листинг 13.21.

```

1 XmlDocument xDoc = new XmlDocument();
2 xDoc.Load(@"persons.xml");
3 XmlElement xRoot = xDoc.DocumentElement;
4 // создаем новый элемент user
5 XmlElement userElem = xDoc.CreateElement("user");
6 // создаем атрибут name
7 XmlAttribute nameAttr = xDoc.CreateAttribute("name");
8 // создаем элементы company и age
9 XmlElement companyElem = xDoc.CreateElement("company");
10 XmlElement ageElem = xDoc.CreateElement("age");
11 // создаем текстовые значения для элементов и атрибута
12 XmlText nameText = xDoc.CreateTextNode("Mark Zuckerberg");
13 XmlText companyText = xDoc.CreateTextNode("Facebook");
14 XmlText ageText = xDoc.CreateTextNode("30");
15 //добавляем узлы
16 nameAttr.AppendChild(nameText);
17 companyElem.AppendChild(companyText);
18 ageElem.AppendChild(ageText);
19 userElem.Attributes.Append(nameAttr);
20 userElem.AppendChild(companyElem);
21 userElem.AppendChild(ageElem);
22 xRoot.AppendChild(userElem);
23 xDoc.Save(@"persons.xml");

```

Добавление элементов происходит по одной схеме. Сначала создаем элемент

(xDoc.CreateElement("user")).

Если элемент сложный, то есть содержит в себе другие элементы, то создаем эти элементы. Если элемент простой, содержащий внутри себя некоторое текстовое значение, то создаем этот текст

```
(XmlText companyText = xDoc.CreateTextNode("Facebook"));).
```

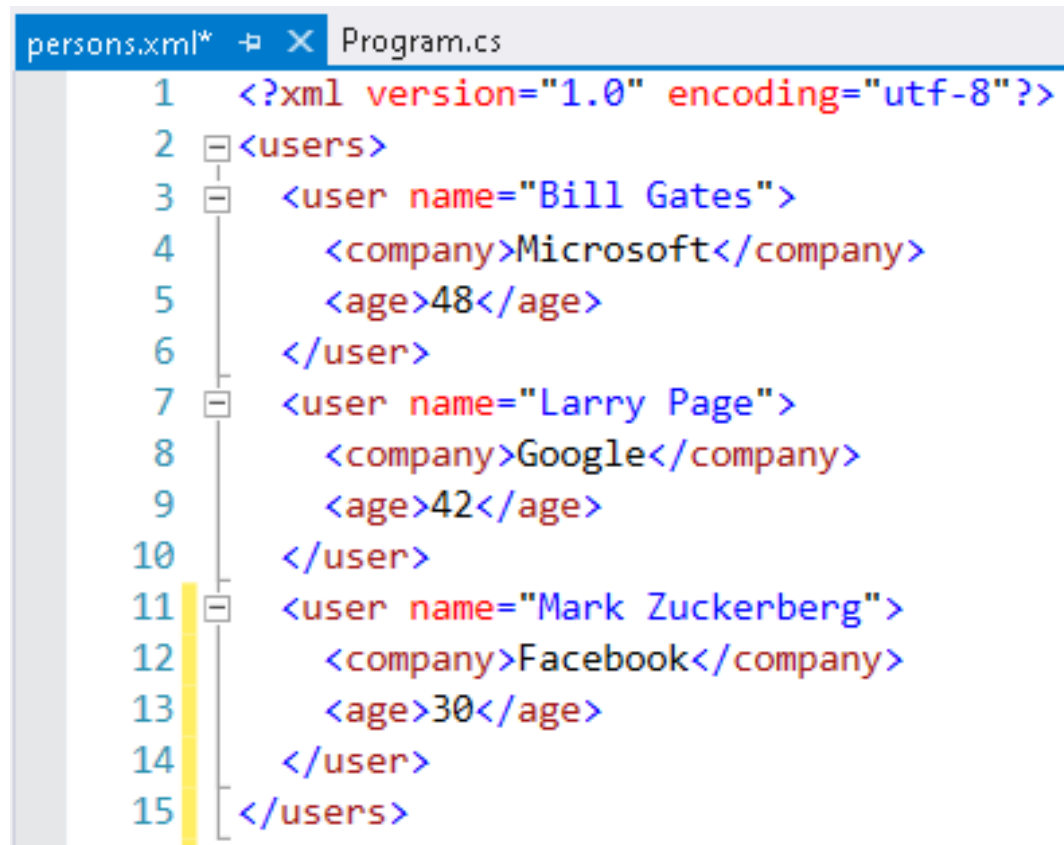
Затем все элементы добавляются в основной элемент user, а тот добавляется в корневой элемент

```
(xRoot.AppendChild(userElem);).
```

Чтобы сохранить измененный документ на диск, используем метод Save:

```
xDoc.Save("persons.xml")
```

Результат выполнения программы:



```
1 <?xml version="1.0" encoding="utf-8"?>
2 <users>
3   <user name="Bill Gates">
4     <company>Microsoft</company>
5     <age>48</age>
6   </user>
7   <user name="Larry Page">
8     <company>Google</company>
9     <age>42</age>
10  </user>
11  <user name="Mark Zuckerberg">
12    <company>Facebook</company>
13    <age>30</age>
14  </user>
15 </users>
```

Рисунок 13.3

После этого в xml-файле появится следующий элемент:

```
<user name="Mark Zuckerberg">
  <company>Facebook</company>
  <age>30</age>
</user>
```

Удаление первого узла xml-документа будет выглядеть следующим образом:

Листинг 13.22.

1	<code>XmlDocument xDoc = new XmlDocument()</code>
2	<code>xDoc.Load(@"persons.xml")</code>
3	<code>XmlElement xRoot = xDoc.DocumentElement</code>
4	<code>XmlNode firstNode = xRoot.FirstChild</code>
5	<code>xRoot.RemoveChild(firstNode)</code>
6	<code>xDoc.Save(@"persons.xml")</code>

§13.8 Сериализация и десериализация

Ранее мы рассмотрели, как сохранять информацию в текстовые файлы. Но нередко подобных механизмов оказывается недостаточно особенно для сохранения сложных объектов. С этой проблемой призван справиться механизм сериализации.

Сериализация — это процесс преобразования объекта в поток байтов для сохранения или передачи в память, базу данных или файл. Эта операция предназначена для того, чтобы сохранить состояния объекта для последующего воссоздания при необходимости. Обратный процесс называется **десериализацией**.

На этом рисунке показан общий процесс сериализации.

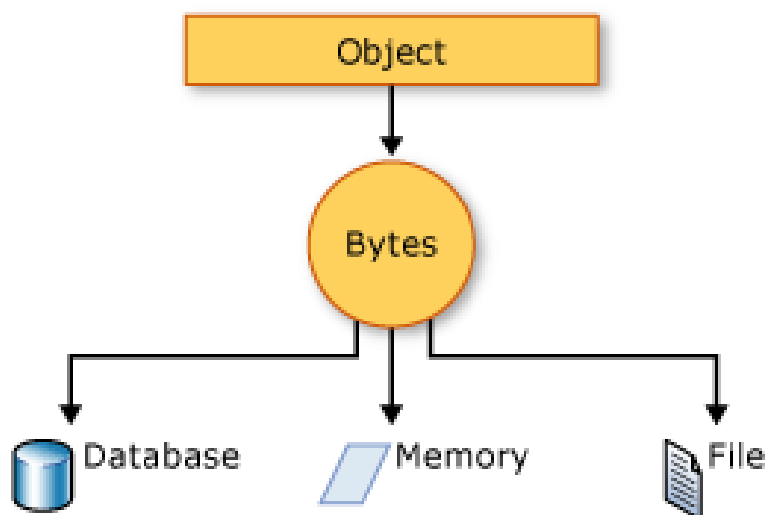


Рисунок 13.4 –Демонстрация процесса сериализации

Объект сериализуется в поток, который служит для передачи данных. Поток также может содержать сведения о типе объекта, в том числе о его версии, языке и региональных параметрах, а также имени сборки. В этом формате потока объект можно сохранить в базе данных, файле или памяти.

Сериализация позволяет разработчику сохранять состояние объекта и воссоздавать его при необходимости. Это полезно для длительного хранения объектов или для обмена данными. Посредством сериализации разработчик может выполнять следующие действия:

- Отправка объекта в удаленное приложение с помощью веб-службы;
- Передача объекта из одного домена в другой;
- Передача объекта через брандмауэр в виде строки JSON или XML;
- Хранение сведений о безопасности и пользователях между приложениями.

.NET Framework предоставляет пространства имен `System.Runtime.Serialization` и `System.Xml.Serialization`, которые помогут с сериализацией и десериализацией вашего объекта.

.NET Framework предоставляет три механизма сериализации, которые вы можете использовать по умолчанию:

- `XmlSerializer`
- `DataContractSerializer`
- `BinaryFormatter`

Атрибут Serializable

Чтобы объект определенного класса можно было сериализовать, надо этот класс пометить атрибутом `Serializable`:

Листинг 13.23.

1	[Serializable]
2	<code>class Person</code>
3	<code>{</code>
4	<code>public string Name { get; set; }</code>
5	<code>public int Year { get; set; }</code>
6	<code>public Person(string name, int year)</code>
7	<code>{</code>
8	<code> Name = name;</code>
9	<code> Year = year;</code>
10	<code>}</code>
11	<code>}</code>

При отсутствии данного атрибута объект `Person` не сможет быть сериализован, и при попытке сериализации будет выброшено исключение `SerializationException`.

Сериализация применяется к свойствам и полям класса. Если мы не хотим, чтобы какое-то поле класса сериализовалось, то мы его помечаем атрибутом `NonSerialized`:

Листинг 13.24.

1	<code>[Serializable]</code>
2	<code>class Person</code>
3	<code>{</code>
4	<code>public string Name { get; set; }</code>
5	<code>public int Year { get; set; }</code>
6	<code>[NonSerialized]</code>
7	<code>public string accNumber;</code>
8	<code>public Person(string name, int year, string acc)</code>
9	<code>{</code>
10	<code> Name = name;</code>
11	<code> Year = year;</code>
12	<code> accNumber = acc;</code>
13	<code>}</code>
14	<code>}</code>

При наследовании подобного класса, следует учитывать, что атрибут `Serializable` автоматически не наследуется. И если мы хотим, чтобы производный класс также мог бы быть сериализован, то опять же мы применяем к нему атрибут `Serializable`.

Использование XmlSerializer

Для удобного сохранения и извлечения объектов из файлов xml может использоваться класс `XmlSerializer`.

Во-первых, `XmlSerializer` предполагает некоторые ограничения. Например, класс, подлежащий сериализации, должен иметь стандартный конструктор без параметров. Также сериализации подлежат только открытые члены. Если в классе есть поля или свойства с модификатором `private`, то при сериализации они будут игнорироваться.

Во-вторых, `XmlSerializer` требует указания типа:

Листинг 13.25.

1	<code>using System;</code>
2	<code>using System.IO;</code>

3	using System.Xml.Serialization;
4	namespace Serialization
5	{
6	// класс и его члены объявлены как public
7	[Serializable]
8	public class Person
9	{
10	public string Name { get; set; }
11	public int Age { get; set; }
12	// стандартный конструктор без параметров
13	public Person()
14	{ }
15	public Person(string name, int age)
16	{
17	Name = name;
18	Age = age;
19	}
20	}
21	class Program
22	{
23	static void Main(string[] args)
24	{
25	// объект для сериализации
26	Person person = new Person("Tom", 29);
27	Console.WriteLine("Объект создан");
28	// передаем в конструктор тип класса
29	XmlSerializer formatter = new XmlSerializer(typeof(Person));
30	// получаем поток, куда будем записывать сериализованный объект
31	using (FileStream fs = new FileStream("persons.xml", FileStream.OpenOrCreate))
32	{
33	formatter.Serialize(fs, person);
34	Console.WriteLine("Объект сериализован");
35	}
36	// десериализация
37	using (FileStream fs = new FileStream("persons.xml", FileStream.OpenOrCreate))
38	{
39	Person newPerson = (Person)formatter.Deserialize(fs);
40	Console.WriteLine("Объект десериализован");

41	<code>Console.WriteLine(\$"Имя: {newPerson.Name} --- Возраст: {newPerson.Age}");</code>
42	<code>}</code>
43	<code>Console.ReadLine();</code>
44	<code>}</code>
45	<code>}</code>
46	<code>}</code>

Итак, класс `Person` общедоступный и имеет общедоступные свойства, поэтому он может сериализоваться. При создании объекта `XmlSerializer` передаем в конструктор тип класса. Метод `Serialize` добавляет данные в файл `persons.xml`. А метод `Deserialize` извлекает их оттуда.

Если мы откроем файл `persons.xml`, то увидим содержание нашего объекта:

```

persons.xml*  Program.cs
1  <?xml version="1.0"?>
2  <Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4      <Name>Tom</Name>
5      <Age>29</Age>
6  </Person>

```

Рисунок 13.5 – Содержание данных об объекте в xml-файле

Равным образом мы можем сериализовать массив или коллекцию объектов, но главное требование состоит в том, чтобы в них был определен стандартный конструктор:

Листинг 13.26.

1	<code>Person person1 = new Person("Tom", 29);</code>
2	<code>Person person2 = new Person("Bill", 25);</code>
3	<code>Person[] people = new Person[] { person1, person2 };</code>
4	<code>XmlSerializer formatter = new XmlSerializer(typeof(Person[]));</code>
5	<code>using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))</code>
6	<code>{</code>
7	<code>formatter.Serialize(fs, people);</code>
8	<code>}</code>
9	<code>using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))</code>

10	{
11	Person[] newpeople = (Person[])formatter.Deserialize(fs);
12	foreach (Person p in newpeople)
13	{
14	Console.WriteLine(\$"Имя: {p.Name} --- Возраст: {p.Age}");
15	}
16	}

Но это был простой объект. Однако с более сложными по составу объектами работать так же просто. Например:

Листинг 13.27.

1	using System;
2	using System.IO;
3	using System.Xml.Serialization;
4	namespace Serialization
5	{
6	[Serializable]
7	public class Person
8	{
9	public string Name { get; set; }
10	public int Age { get; set; }
11	public Company Company { get; set; }
12	public Person()
13	{ }
14	public Person(string name, int age, Company comp)
15	{
16	Name = name;
17	Age = age;
18	Company = comp;
19	}
20	}
21	[Serializable]
22	public class Company
23	{
24	public string Name { get; set; }
25	// стандартный конструктор без параметров
26	public Company() { }
27	public Company(string name)
28	{
29	Name = name;

30	}
31	}
32	class Program
33	{
34	static void Main(string[] args)
35	{
36	Person person1 = new Person("Tom", 29, new Company("Microsoft"));
37	Person person2 = new Person("Bill", 25, new Company("Apple"));
38	Person[] people = new Person[] { person1, person2 };
39	XmlSerializer formatter = new XmlSerializer(typeof(Person[]));
40	using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))
41	{
42	formatter.Serialize(fs, people);
43	}
44	using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))
45	{
46	Person[] newpeople = (Person[])formatter.Deserialize(fs);
47	foreach (Person p in newpeople)
48	{
49	Console.WriteLine(\$"Имя: {p.Name} --- Возраст: {p.Age} --- Компания: {p.Company.Name}");
50	}
51	}
52	Console.ReadLine();
53	}
54	}
55	}

Класс `Person` содержит свойство `Company`, которое будет хранить объект класса `Company`. Члены класса `Company` объявляются с модификатором `public`, кроме того также присутствует стандартный конструктор без параметров. В итоге после сериализации мы получим следующий xml-документ:



```
1  <?xml version="1.0"?>
2  <ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
4      <Person>
5          <Name>Tom</Name>
6          <Age>29</Age>
7          <Company>
8              <Name>Microsoft</Name>
9          </Company>
10     </Person>
11     <Person>
12         <Name>Bill</Name>
13         <Age>25</Age>
14         <Company>
15             <Name>Apple</Name>
16         </Company>
17     </Person>
18 </ArrayOfPerson>
```

Рисунок 13.6 – Результат выполнения сериализации