**№ 14 Сериализация**

**Курносенко Софья**

**Введение в сериализацию объектов**

Ранее мы посмотрели, как сохранять информацию в текстовые файлы, а также затронули сохранение несложных структур в бинарные данные. Но нередко подобных механизмов оказывается недостаточно особенно для сохранения сложных объектов. С этой проблемой призван справится механизм сериализации. **Сериализация** представляет процесс преобразования какого-либо объекта в поток байтов. После преобразования мы можем этот поток байтов или записать на диск или сохранить его временно в памяти. А при необходимости можно выполнить обратный процесс - **десериализацию**, то есть получить из потока байтов ранее сохраненный объект.

### Атрибут Serializable

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

[Serializable]

class Person

{

public string Name { get; set; }

public int Year { get; set; }

public Person(string name, int year)

{

Name = name;

Year = year;

}

}

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**:

[Serializable]

class Person

{

public string Name { get; set; }

public int Year { get; set; }

[NonSerialized]

public string accNumber;

public Person(string name, int year, string acc)

{

Name = name;

Year = year;

accNumber = acc;

}

}

Атрибут [NonSerialized] можно применить только к полю!

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]

class Worker : Person

### Формат сериализации

Хотя сериализация представляет собой преобразование объекта в некоторый набор байтов, но в действительности только бинарным форматом она не ограничивается. Итак, в .NET можно использовать следующие форматы:

* бинарный
* SOAP
* xml
* JSON

Для каждого формата предусмотрен свой класс: для сериализации в бинарный формат - класс **BinaryFormatter**, для формата SOAP - класс **SoapFormatter**, для xml - **XmlSerializer**, для json - **DataContractJsonSerializer**.

Для классов BinaryFormatter и SoapFormatter сам функционал сериализации определен в интерфейсе **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);

}

Хотя классы BinaryFormatter и SoapFormatter по-разному реализуют данный интерфейс, но общий функционал будет тот же: для сериализации будет использоваться метод Serialize, который в качестве параметров принимает поток, куда помещает сериализованные данные (например, бинарный файл), и объект, который надо сериализовать. А для десериализации будет применяться метод Deserialize, который в качестве параметра принимает поток с сериализованными данными.

Класс XmlSerializer не реализует интерфейс IFormatter и по функциональности в целом несколько отличается от BinaryFormatter и SoapFormatter, но и он также предоставляет для сериализации метод Serialize, а для десериализации Deserialize. И в этом плане очень легко при необходимости перейти от одного способа сериализации к другому.

## Бинарная сериализация. BinaryFormatter

Для бинарной сериализации применяется класс **BinaryFormatter**:

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

namespace Serialization

{

[Serializable]

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

class Program

{

static void Main(string[] args)

{

// объект для сериализации

Person person = new Person("Tom", 29);

Console.WriteLine("Объект создан");

// создаем объект BinaryFormatter

BinaryFormatter formatter = new BinaryFormatter();

// получаем поток, куда будем записывать сериализованный объект

using (FileStream fs = new FileStream(@"D:\ООП\_2к\_1с\OOP\_Course2\_Term1\people.txt", FileMode.OpenOrCreate))

{

formatter.Serialize(fs, person);

Console.WriteLine("Объект сериализован");

}

// десериализация из файла people.dat

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

Person newPerson = (Person)formatter.Deserialize(fs);

Console.WriteLine("Объект десериализован");

Console.WriteLine($"Имя: {newPerson.Name} --- Возраст: {newPerson.Age}");

}

Console.ReadLine();

}

}

}

Так как класс BinaryFormatter определен в пространстве имен System.Runtime.Serialization.Formatters.Binary, то в самом начале подключаем его.

У нас есть простенький класс Person, который объявлен с атрибутом Serializable. Благодаря этому его объекты будут доступны для сериализации.

Далее создаем объект BinaryFormatter: BinaryFormatter formatter = new BinaryFormatter();

Затем последовательно выполняем сериализацию и десериализацию. Для обоих операций нам нужен поток, в который либо сохранять, либо из которого считывать данные. Данный поток представляет объект FileStream, который записывает нужный нам объект Person в файл people.dat.

Сериализация одним методом formatter.Serialize(fs, person) добавляет все данные об объекте Person в файл people.dat.

При десериализации нам нужно еще преобразовать объект, возвращаемый функцией Deserialize, к типу Person: (Person)formatter.Deserialize(fs).

Как вы видите, сериализация значительно упрощает процесс сохранения объектов в бинарную форму по сравнению, например, с использованием связки классов BinaryWriter/BinaryReader.

Хотя мы взяли лишь один объект Person, но равным образом мы можем использовать и массив подобных объектов, список или иную коллекцию, к которой применяется атрибут Serializable. Посмотрим на примере массива:

using System;

using System.IO;

using System.Runtime.Serialization.Formatters.Binary;

namespace Serialization

{

[Serializable]

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

class Program

{

static void Main(string[] args)

{

Person person1 = new Person("Tom", 29);

Person person2 = new Person("Bill", 25);

// массив для сериализации

Person[] people = new Person[] { person1, person2 };

BinaryFormatter formatter = new BinaryFormatter();

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

// сериализуем весь массив people

formatter.Serialize(fs, people);

Console.WriteLine("Объект сериализован");

}

// десериализация

using (FileStream fs = new FileStream("people.dat", FileMode.OpenOrCreate))

{

Person[] deserilizePeople = (Person[])formatter.Deserialize(fs);

foreach (Person p in deserilizePeople)

{

Console.WriteLine($"Имя: {p.Name} --- Возраст: {p.Age}");

}

}

}

}

}

## Сериализация в формат SOAP. SoapFormatter

SOAP (Simple Object Access Protocol) – это стандартизированный протокол для передачи сообщений между клиентом и веб-сервисом, а там, где есть стандарты, там есть достаточная простота проверок. Основной целью разработки этого языка/словаря было обеспечение совместимости между платформами. Итак, Java вызывает COM, вызывает .Net, вызывает J2EE и rest, оставаясь расширяемым в обозримом будущем. Вместо того, чтобы строить еще один мост между объектом COM и классом Java, мы можем просто использовать SOAP для раскрытия функциональности нашего объекта, чтобы другие платформы могли использовать их без необходимости в мостах.

При такой сериализации данные упакуются в конверт SOAP, данные в котором имеют вид xml-подобного документа.

# Класс Soap Formatter

Namespace: [System.Runtime.Serialization.Formatters.Soap](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.serialization.formatters.soap?view=netframework-4.8)

**Примечание**

Начиная с .NET Framework 2.0, этот класс устарел.

# Работа с JSON

## Сериализация в JSON. JsonSerializer

Обозначение объектов JavaScript (JSON - JavaScript Object Notation) - стандартный текстовый формат для представления структурированных данных на основе синтаксиса объекта JavaScript. Он обычно используется для передачи данных в веб-приложениях (например, отправка некоторых данных с сервера клиенту, таким образом, чтобы это могло отображаться на веб-странице или наоборот).

Основная функциональность по работе с JSON сосредоточена в пространстве имен **System.Text.Json**.

Ключевым типом является класс **JsonSerializer**, который и позволяет сериализовать объект в json и, наоборот, десериализовать код json в объект C#.

Для сохранения объекта в json в классе JsonSerializer определен статический метод **Serialize()**, который имеет ряд перегруженных версий. Некоторые из них:

*Serialize(Object, Type, JsonSerializerOptions)* преобразует объект определенного типа в json строку.

*Serialize(Stream, Object, Type, JsonSerializerOptions)* преобразует предоставленное значение в текст JSON в кодировке UTF-8 и записывает его в Stream.

*SerializeAsync(Stream, Object, Type, JsonSerializerOptions, CancellationToken)* асинхронно преобразует значение указанного типа в текст JSON в кодировке UTF-8 и записывает его в указанный поток.

*Deserialize(Stream, Type, JsonSerializerOptions)* считывает текст в кодировке UTF-8, представляющий одно значение JSON, в returnType. Поток будет прочитан до конца.

*DeserializeAsync(Stream, Type, JsonSerializerOptions, CancellationToken)* асинхронно считывает текст в кодировке UTF-8, представляющий одно значение JSON, в экземпляр указанного типа. Поток будет прочитан до конца.

Рассмотрим применение класса на простом примере. Сериализуем и десериализуем простейший объект:

using System;

using System.Text.Json;

namespace HelloApp

{

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

Person tom = new Person { Name = "Tom", Age = 35 };

string json = JsonSerializer.Serialize<Person>(tom);

Console.WriteLine(json);

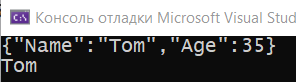
Person restoredPerson = JsonSerializer.Deserialize<Person>(json);

Console.WriteLine(restoredPerson.Name);

}

}

}



Здесь вначале сериализуем с помощью метода JsonSerializer.Serialize() объект типа Person в стоку типа json. Затем обратно получаем из этой строки объект Person посредством метода JsonSerializer.Deserialize().

### Некоторые замечания по сериализации/десериализации

**Объект, который подвергается десериализации, должен иметь конструктор без параметров.** Например, в примере выше это конструктор по умолчанию. Но можно также явным образом определить подобный конструктор в классе.

Подобное вызовет ошибку:

using System;

using System.Text.Json;

namespace HelloApp

{

class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Person(string name)

{

Name = name;

}

}

class Program

{

static void Main(string[] args)

{

Person tom = new Person("Franny");

string json = JsonSerializer.Serialize<Person>(tom);

Console.WriteLine(json);

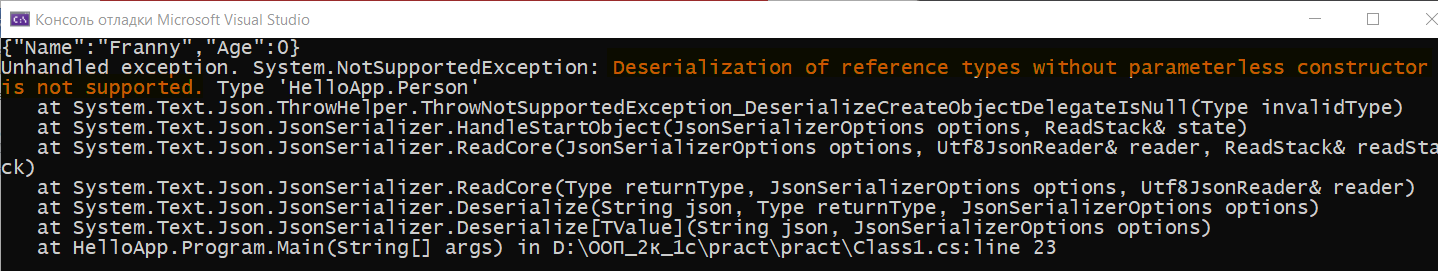
Person restoredPerson = JsonSerializer.Deserialize<Person>(json);

Console.WriteLine(restoredPerson.Name);

}

}

}



**Сериализации подлежат только публичные свойства объекта** (с модификатором public).

### Запись и чтение файла json

Поскольку методы SerializeAsyc/DeserializeAsync могут принимать поток типа Stream, то соответственно мы можем использовать файловый поток для сохранения и последующего извлечения данных:

using System;

using System.IO;

using System.Text.Json;

using System.Threading.Tasks;

namespace HelloApp

{

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

class Program

{

static async Task Main(string[] args)

{

            // сохранение данных

            using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))

{

Person tom = new Person() { Name = "Tom", Age = 35 };

await JsonSerializer.SerializeAsync<Person>(fs, tom);

Console.WriteLine("Data has been saved to file");

}

// чтение данных

using (FileStream fs = new FileStream("user.json", FileMode.OpenOrCreate))

{

Person restoredPerson = await JsonSerializer.DeserializeAsync<Person>(fs);

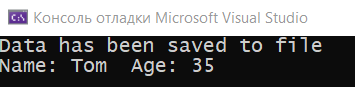
Console.WriteLine($"Name: {restoredPerson.Name}  Age: {restoredPerson.Age}");

}

}

}

}



**Асинхронность** позволяет вынести отдельные задачи из основного потока в специальные асинхронные методы или блоки кода. Особенно это актуально в графических программах, где продолжительные задачи могу блокировать интерфейс пользователя. И чтобы этого не произошло, нужно задействовать асинхронность. Также асинхронность несет выгоды в веб-приложениях при обработке запросов от пользователей, при обращении к базам данных или сетевым ресурсам. При больших запросах к базе данных асинхронный метод просто уснет на время, пока не получит данные от БД, а основной поток сможет продолжить свою работу. В синхронном же приложении, если бы код получения данных находился в основном потоке, этот поток просто бы блокировался на время получения данных.

Ключевыми для работы с асинхронными вызовами в C# являются два ключевых слова: **async** и **await**, цель которых - упростить написание асинхронного кода. Они используются вместе для создания асинхронного метода.

**Асинхонный метод** обладает следующими признаками:

* В заголовке метода используется модификатор **async**
* Метод содержит одно или несколько выражений **await**
* В качестве возвращаемого типа используется один из следующих:
  + void
  + Task
  + Task<T>
  + ValueTask<T>

Асинхронный метод, как и обычный, может использовать любое количество параметров или не использовать их вообще. Однако асинхронный метод не может определять параметры с модификаторами **out** и **ref**.

Также стоит отметить, что слово **async**, которое указывается в определении метода, не делает автоматически метод асинхронным. Оно лишь указывает, что данный метод может содержать одно или несколько выражений **await**.

В данном случае вначале данные сохраняются в файл user.json и затем считываются из него.

### Настройка сериализации с помощью JsonSerializerOptions

По умолчанию JsonSerializer сериализует объекты в минимифицированный код. С помощью дополнительного параметра типа **JsonSerializerOptions** можно настроить механизм сериализации/десериализации, используя свойства JsonSerializerOptions. Некоторые из его свойств:

* **AllowTrailingCommas**: устанавливает, надо ли добавлять после последнего элемента в json запятую. Если равно true, запятая добавляется
* **IgnoreNullValues**: устанавливает, будут ли сериализоваться/десериализоваться в json объекты и их свойства со значением null
* **IgnoreReadOnlyProperties**: аналогично устанавливает, будут ли сериализоваться свойства, предназначенные только для чтения
* **WriteIndented**: устанавливает, будут ли добавляться в json пробелы (условно говоря, для красоты). Если равно true устанавливаются дополнительные пробелы

Применение:

using System;

using System.Text.Json;

namespace HelloApp

{

class Person

{

public string Name { get; set; }

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

// сохранение данных

var options = new JsonSerializerOptions

{

WriteIndented = true

};

Person tom = new Person { Name = "Tom", Age = 35 };

string json = JsonSerializer.Serialize<Person>(tom, options);

Console.WriteLine(json);

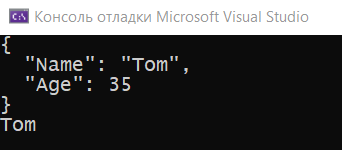
Person restoredPerson = JsonSerializer.Deserialize<Person>(json);

Console.WriteLine(restoredPerson.Name);

}

}

}



### Настройка сериализации с помощью атрибутов

По умолчанию сериализации подлежат все публичные свойства. Кроме того, в выходном объекте json все названия свойств соответствуют названиям свойств объекта C#.

Атрибут **JsonIgnore** позволяет исключить из сериализации определенное свойство. А **JsonPropertyName** позволяет замещать оригинальное название свойства. Пример использования:

using System;

using System.Text.Json;

using System.Text.Json.Serialization;

namespace HelloApp

{

class Person

{

[JsonPropertyName("firstname")]

public string Name { get; set; }

[JsonIgnore]

public int Age { get; set; }

}

class Program

{

static void Main(string[] args)

{

Person tom = new Person() { Name = "Tom", Age = 35 };

string json = JsonSerializer.Serialize<Person>(tom);

Console.WriteLine(json);

Person restoredPerson = JsonSerializer.Deserialize<Person>(json);

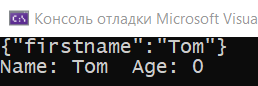
Console.WriteLine($"Name: {restoredPerson.Name} Age: {restoredPerson.Age}");

}

}

}

В данном случае свойство Age будет игнорироваться, а для свойства Name будет использоваться псевдоним "firstname".



Обратите внимание, что, поскольку свойство Age не было сериализовано, то при десериализации для него используется значение по умолчанию.

Без обобщенных методов:

Person tom = new Person() { Name = "Tom", Age = 35 };

string json = JsonSerializer.Serialize(tom, typeof(Person));

Console.WriteLine(json);

Person restoredPerson = (Person)JsonSerializer.Deserialize(json, typeof(Person));

Console.WriteLine($"Name: {restoredPerson.Name} Age: {restoredPerson.Age}");

# Работа с XML в C#

## 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:

class User

{

public string Name { get; set; }

public int Age { get; set; }

public string Company { get; set; }

}

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

User user1 = new User { Name = "Bill Gates", Age = 48, Company = "Microsoft" };

User user2 = new User { Name = "Larry Page", Age = 42, Company = "Google" };

List<User> users = new List<User> { user1, user2 };

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

<?xml version="1.0" encoding="utf-8" ?>

<users>

<user name="Bill Gates">

<company>Microsoft</company>

<age>48</age>

</user>

<user name="Larry Page">

<company>Google</company>

<age>48</age>

</user>

</users>

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, внутрь 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:

<?xml version="1.0" encoding="utf-8" ?>

<users>

<user>

<name>Bill Gates</name>

<company>Microsoft</company>

<age>48</age>

</user>

<user>

<name>Larry Page</name>

<company>Google</company>

<age>48</age>

</user>

</users>

## 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:

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

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

* Свойство **Attributes** возвращает объект XmlAttributeCollection, который представляет коллекцию атрибутов
* Свойство **ChildNodes** возвращает коллекцию дочерних узлов для данного узла
* Свойство **HasChildNodes** возвращает true, если текущий узел имеет дочерние узлы
* Свойство **FirstChild** возвращает первый дочерний узел
* Свойство **LastChild** 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 *users.xml* и определим в нем следующее содержание:

<?xml version="1.0" encoding="utf-8" ?>

<users>

<user name="Bill Gates">

<company>Microsoft</company>

<age>48</age>

</user>

<user name="Larry Page">

<company>Google</company>

<age>48</age>

</user>

</users>

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

using System;

using System.Xml;

namespace HelloApp

{

class Program

{

static void Main(string[] args)

{

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D:\\users.xml");

// получим корневой элемент

XmlElement? xRoot = xDoc.DocumentElement;

if (xRoot != null)

{

// обход всех узлов в корневом элементе

foreach (XmlElement xnode in xRoot)

{

// получаем атрибут name

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

Console.WriteLine(attr?.Value);

// обходим все дочерние узлы элемента user

foreach (XmlNode childnode in xnode.ChildNodes)

{

// если узел - company

if (childnode.Name == "company")

{

Console.WriteLine($"Компания: {childnode.InnerText}");

}

// если узел age

if (childnode.Name == "age")

{

Console.WriteLine($"Возраст: {childnode.InnerText}");

}

}

Console.WriteLine();

}

}

}

}

}

В итоге я получу следующий вывод на консоли:

Bill Gates

Компания: Microsoft

Возраст: 48

Larry Page

Компания: Google

Возраст: 42

Чтобы начать работу с документом xml, нам надо создать объект XmlDocument и затем загрузить в него xml-файл: xDoc.Load("users.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 классов и структур по данным из xml:

using System;

using System.Xml;

using System.Collections.Generic;

namespace HelloApp

{

class User

{

public string? Name { get; set; }

public int Age { get; set; }

public string? Company { get; set; }

}

class Program

{

static void Main(string[] args)

{

List<User> users = new List<User>();

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D:\\users.xml");

XmlElement? xRoot = xDoc.DocumentElement;

if (xRoot != null)

{

foreach (XmlElement xnode in xRoot)

{

User user = new User();

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

user.Name = attr?.Value;

foreach (XmlNode childnode in xnode.ChildNodes)

{

if (childnode.Name == "company")

user.Company = childnode.InnerText;

if (childnode.Name == "age")

user.Age = int.Parse(childnode.InnerText);

}

users.Add(user);

}

foreach (User u in users)

Console.WriteLine($"{u.Name} ({u.Company}) - {u.Age}");

}

}

}

}

В данном случае определен класс User с тремя свойствами. При переборе узлов файла xml значения элементов и их атрибутов передается объекту класса User.

Консольный вывод программы:

Bill Gates (Microsoft) - 48

Larry Page (Google) -42

## XPath

**XPath** представляет язык запросов в XML. Он позволяет выбирать элементы, соответствующие определенному селектору.

Рассмотрим некоторые наиболее распространенные селекторы:

.

выбор текущего узла

..

выбор родительского узла

\*

выбор всех дочерних узлов текущего узла

user

выбор всех узлов с определенным именем, в данном случае с именем "user"

@name

выбор атрибута текущего узла, после знака @ указывается название атрибута (в данном случае "name")

@+

выбор всех атрибутов текущего узла

element[3]

выбор определенного дочернего узла по индексу, в данном случае третьего узла

//user

выбор в документе всех узлов с именем "user"

user[@name='Bill Gates']

выбор элементов с определенным значением атрибута. В данном случае выбираются все элементы "user" с атрибутом name='Bill Gates'

user[company='Microsoft']

выбор элементов с определенным значением вложенного элемента. В данном случае выбираются все элементы "user", у которых дочерний элемент "company" имеет значение 'Microsoft'

//user/company

выбор в документе всех узлов с именем "company", которые находятся в элементах "user"

Действие запросов XPath основано на применении двух методов класса **XmlElement**:

* **SelectSingleNode()**: выбор единственного узла из выборки. Если выборка по запросу содержит несколько узлов, то выбирается первый
* **SelectNodes()**: выборка по запросу коллекции узлов в виде объекта XmlNodeList

Для запросов возьмем xml-документ:

<?xml version="1.0" encoding="utf-8" ?>

<users>

<user name="Bill Gates">

<company>Microsoft</company>

<age>48</age>

</user>

<user name="Larry Page">

<company>Google</company>

<age>48</age>

</user>

</users>

Теперь выберем все узлы корневого элемента, то есть все элементы user:

XmlDocument xDoc = new XmlDocument();

xDoc.Load("D://users.xml");

XmlElement xRoot = xDoc.DocumentElement;

// выбор всех дочерних узлов

XmlNodeList childnodes = xRoot.SelectNodes("\*");

foreach (XmlNode n in childnodes)

Console.WriteLine(n.OuterXml);

Выберем все узлы <user>:

XmlNodeList childnodes = xRoot.SelectNodes("user");

Выведем на консоль значения атрибутов name у элементов user:

XmlNodeList childnodes = xRoot.SelectNodes("user");

foreach (XmlNode n in childnodes)

Console.WriteLine(n.SelectSingleNode("@name").Value);

Результатом выполнения будет следующий вывод:

Bill Gates

Larry Page

Выберем узел, у которого атрибут name имеет значение "Bill Gates":

XmlNode childnode = xRoot.SelectSingleNode("user[@name='Bill Gates']");

if (childnode != null)

Console.WriteLine(childnode.OuterXml);

Выберем узел, у которого вложенный элемент "company" имеет значение "Microsoft":

XmlNode childnode = xRoot.SelectSingleNode("user[company='Microsoft']");

if (childnode != null)

Console.WriteLine(childnode.OuterXml);

Допустим, нам надо получить только компании. Для этого надо осуществить выборку вниз по иерархии элементов:

XmlNodeList childnodes = xRoot.SelectNodes("//user/company");

foreach (XmlNode n in childnodes)

Console.WriteLine(n.InnerText);

## Сериализация в XML. 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. Помимо этого **сам класс должен быть объявлен с модификатором public**, иначе при сериализации возникнет исключение.

Во-вторых, **XmlSerializer требует указания типа**:

using System;

using System.IO;

using System.Xml.Serialization;

namespace Serialization

{

    // класс и его члены объявлены как public

    [Serializable]

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

// стандартный конструктор без параметров

public Person()

{ }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

class Program

{

static void Main(string[] args)

{

            // объект для сериализации

            Person person = new Person("Tom", 29);

Console.WriteLine("Объект создан");

// передаем в конструктор тип класса

XmlSerializer formatter = new XmlSerializer(typeof(Person));

// получаем поток, куда будем записывать сериализованный объект

using (FileStream fs = new FileStream(@"D:\ООП\_2к\_1с\pract\people.xml", FileMode.OpenOrCreate))

{

formatter.Serialize(fs, person);

Console.WriteLine("Объект сериализован");

}

// десериализация

using (FileStream fs = new FileStream(@"D:\ООП\_2к\_1с\pract\people.xml", FileMode.OpenOrCreate))

{

Person newPerson = (Person)formatter.Deserialize(fs);

Console.WriteLine("Объект десериализован");

Console.WriteLine($"Имя: {newPerson.Name} --- Возраст: {newPerson.Age}");

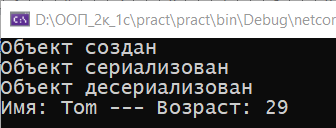
}

Console.ReadLine();

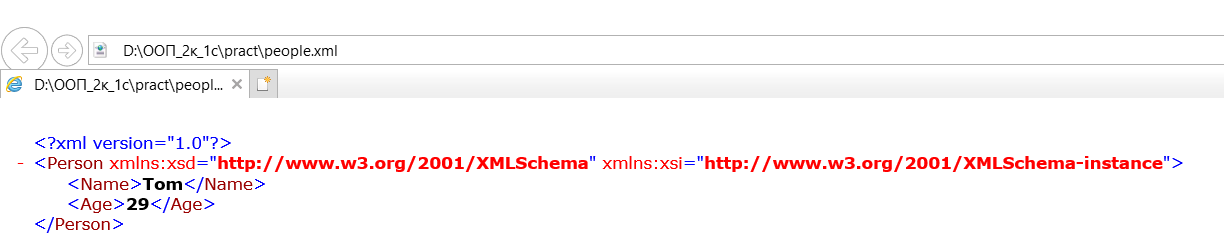
}

}

}

****

Файл people.xml получился таким:



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.

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

Person person1 = new Person("Tom", 29);

Person person2 = new Person("Bill", 25);

Person[] people = new Person[] { person1, person2 };

XmlSerializer formatter = new XmlSerializer(typeof(Person[]));

using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))

{

formatter.Serialize(fs, people);

}

using (FileStream fs = new FileStream("people.xml", FileMode.OpenOrCreate))

{

Person[] newpeople = (Person[])formatter.Deserialize(fs);

foreach (Person p in newpeople)

{

Console.WriteLine($"Имя: {p.Name} --- Возраст: {p.Age}");

}

}

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

using System;

using System.IO;

using System.Xml.Serialization;

namespace Serialization

{

[Serializable]

public class Person

{

public string Name { get; set; }

public int Age { get; set; }

public Company Company { get; set; }

public Person()

{ }

public Person(string name, int age, Company comp)

{

Name = name;

Age = age;

Company = comp;

}

}

[Serializable]

public class Company

{

public string Name { get; set; }

// стандартный конструктор без параметров

public Company() { }

public Company(string name)

{

Name = name;

}

}

class Program

{

static void Main(string[] args)

{

Person person1 = new Person("Tom", 29, new Company("Microsoft"));

Person person2 = new Person("Bill", 25, new Company("Apple"));

Person[] people = new Person[] { person1, person2 };

XmlSerializer formatter = new XmlSerializer(typeof(Person[]));

using (FileStream fs = new FileStream(@"D:\ООП\_2к\_1с\pract\people.xml", FileMode.OpenOrCreate))

{

formatter.Serialize(fs, people);

}

using (FileStream fs = new FileStream(@"D:\ООП\_2к\_1с\pract\people.xml", FileMode.OpenOrCreate))

{

Person[] newpeople = (Person[])formatter.Deserialize(fs);

foreach (Person p in newpeople)

{

Console.WriteLine($"Имя: {p.Name} --- Возраст: {p.Age} --- Компания: {p.Company.Name}");

}

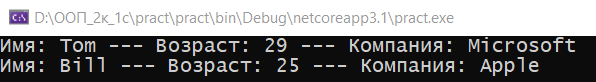
}

Console.ReadLine();

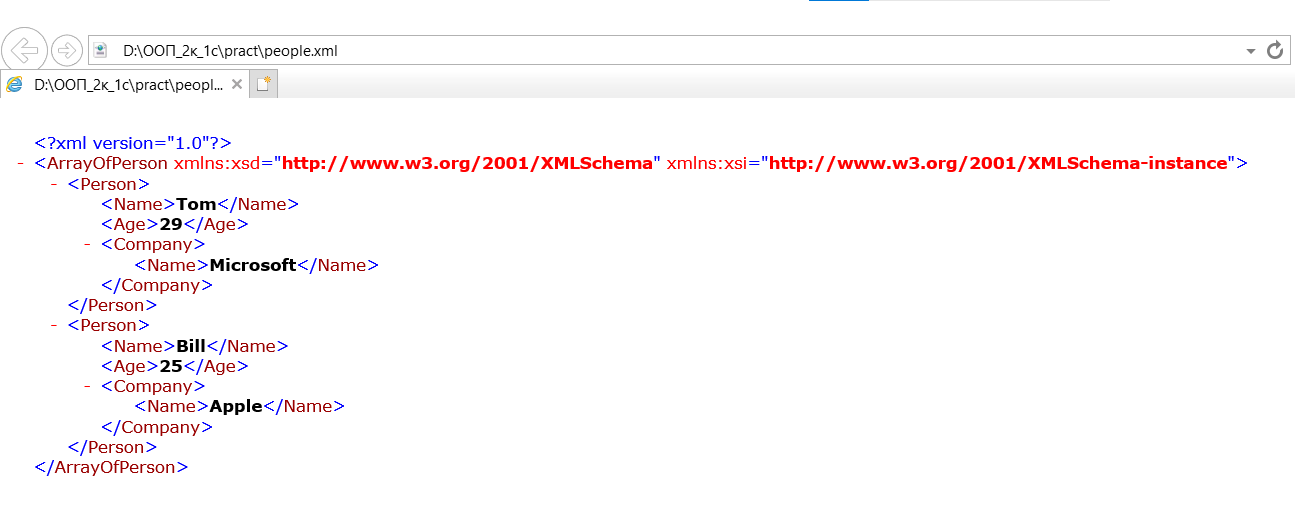
}

}

}

****

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:

****

## Linq to Xml. Создание документа XML

Еще один подход к работе с Xml представляет технология **LINQ to XML**. Вся функциональность LINQ to XML содержится в пространстве имен **System.Xml.Linq**. Рассмотрим основные классы этого пространства имен:

* **XAttribute**: представляет атрибут xml-элемента
* **XComment**: представляет комментарий
* **XDocument**: представляет весь xml-документ
* **XElement**: представляет отдельный xml-элемент

Ключевым классом является XElement, который позволяет получать вложенные элементы и управлять ими. Среди его методов можно отметить следующие:

* **Add()**: добавляет новый атрибут или элемент
* **Attributes()**: возвращает коллекцию атрибутов для данного элемента
* **Elements()**: возвращает все дочерние элементы данного элемента
* **Remove()**: удаляет данный элемент из родительского объекта
* **RemoveAll()**: удаляет все дочерние элементы и атрибуты у данного элемента

Итак, используем функциональность LINQ to XML и создадим новый XML-документ:

using System.Xml.Linq;

namespace lingToXml

{

class Program

{

static void Main()

{

XDocument xdoc = new XDocument();

// создаем первый элемент

XElement iphone6 = new XElement("phone");

// создаем атрибут

XAttribute iphoneNameAttr = new XAttribute("name", "iPhone 6");

XElement iphoneCompanyElem = new XElement("company", "Apple");

XElement iphonePriceElem = new XElement("price", "40000");

// добавляем атрибут и элементы в первый элемент

iphone6.Add(iphoneNameAttr);

iphone6.Add(iphoneCompanyElem);

iphone6.Add(iphonePriceElem);

// создаем второй элемент

XElement galaxys5 = new XElement("phone");

XAttribute galaxysNameAttr = new XAttribute("name", "Samsung Galaxy S5");

XElement galaxysCompanyElem = new XElement("company", "Samsung");

XElement galaxysPriceElem = new XElement("price", "33000");

galaxys5.Add(galaxysNameAttr);

galaxys5.Add(galaxysCompanyElem);

galaxys5.Add(galaxysPriceElem);

// создаем корневой элемент

XElement phones = new XElement("phones");

// добавляем в корневой элемент

phones.Add(iphone6);

phones.Add(galaxys5);

// добавляем корневой элемент в документ

xdoc.Add(phones);

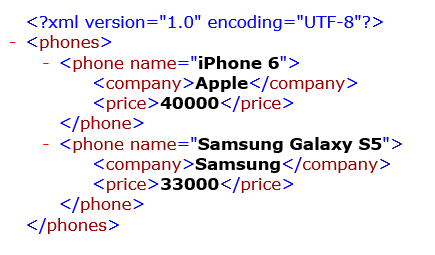
//сохраняем документ

xdoc.Save(@"D:\ООП\_2к\_1с\pract\phones.xml");

}

}

}

****

Чтобы создать документ, нам нужно создать объект класса **XDocument**. Это объект самого верхнего уровня в хml-документе.

Элементы создаются с помощью конструктора класса **XElement**. Конструктор имеет ряд перегруженных версий. Первый параметр конструктора передает название элемента, например, phone. Второй параметр передает значение этого элемента.

Создание атрибута аналогично созданию элемента. Затем все атрибуты и элементы добавляются в элементы phone с помощью метода Add().

Так как документ xml должен иметь один корневой элемент, то затем все элементы phone добавляются в один контейнер - элемент phones.

В конце корневой элемент добавляется в объект XDocument, и этот объект сохраняется на диске в xml-файл с помощью метода **Save()**.

Конструктор класса XElement позволяют задать набор объектов, которые будут входить в элемент. И предыдущий пример мы могли бы сократить следующим способом:

XDocument xdoc = new XDocument(new XElement("phones",

new XElement("phone",

new XAttribute("name", "iPhone 6"),

new XElement("company", "Apple"),

new XElement("price", "40000")),

new XElement("phone",

new XAttribute("name", "Samsung Galaxy S5"),

new XElement("company", "Samsung"),

new XElement("price", "33000"))));

xdoc.Save(@"D:\ООП\_2к\_1с\pract\phones.xml");

## Выборка элементов в LINQ to XML

Возьмем xml-файл, созданный в прошлой теме:

<?xml version="1.0" encoding="utf-8"?>

<phones>

<phone name="iPhone 6">

<company>Apple</company>

<price>40000</price>

</phone>

<phone name="Samsung Galaxy S5">

<company>Samsung</company>

<price>33000</price>

</phone>

</phones>

Переберем его элементы и выведем их значения на консоль:

using System.Xml.Linq;

using System;

namespace lingToXml

{

class Program

{

static void Main()

{

XDocument xdoc = XDocument.Load(@"D:\ООП\_2к\_1с\pract\phones.xml");

foreach (XElement phoneElement in xdoc.Element("phones").Elements("phone"))

{

XAttribute nameAttribute = phoneElement.Attribute("name");

XElement companyElement = phoneElement.Element("company");

XElement priceElement = phoneElement.Element("price");

if (nameAttribute != null && companyElement != null && priceElement != null)

{

Console.WriteLine($"Смартфон: {nameAttribute.Value}");

Console.WriteLine($"Компания: {companyElement.Value}");

Console.WriteLine($"Цена: {priceElement.Value}");

}

Console.WriteLine();

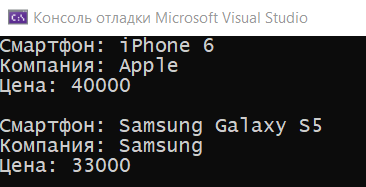
}

}

}

}

И мы получим следующий вывод:

****

Чтобы начать работу с имеющимся xml-файлом, надо сначала загрузить его с помощью статического метода XDocument.Load(), в который передается путь к файлу.

Поскольку xml хранит иерархически выстроенные элементы, то и для доступа к элементам надо идти начиная с высшего уровня в этой иерархии и далее вниз. Так, для получения элементов phone и доступа к ним надо сначала обратиться к корневому элементу, а через него уже к элементам phone: xdoc.Element("phones").Elements("phone")

Метод Element("имя\_элемента") возвращает первый найденный элемент с таким именем. Метод Elements("имя\_элемента") возвращает коллекцию одноименных элементов. В данном случае мы получаем коллекцию элементов phone и поэтому можем перебрать ее в цикле.

Спускаясь дальше по иерархии вниз, мы можем получить атрибуты или вложенные элементы, например, XElement companyElement = phoneElement.Element("company")

Значение простых элементов, которые содержат один текст, можно получить с помощью свойства **Value**: string company = phoneElement.Element("company").Value

Сочетая операторы Linq и LINQ to XML можно довольно просто извлечь из документа данные и затем обработать их. Создадим на основании данных в xml объекты класса Phone:

using System.Xml.Linq;

using System;

using System.Linq;

namespace lingToXml

{

class Phone

{

public string Name { get; set; }

public string Price { get; set; }

}

class Program

{

static void Main()

{

XDocument xdoc = XDocument.Load("phones.xml");

var items = from xe in xdoc.Element("phones").Elements("phone")

where xe.Element("company").Value == "Samsung"

select new Phone

{

Name = xe.Attribute("name").Value,

Price = xe.Element("price").Value

};

foreach (var item in items)

Console.WriteLine($"{item.Name} - {item.Price}");

}

}

}