

**УО «МОГИЛЕВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ  
ИМЕНИ А.А. КУЛЕШОВА»  
СОЦИАЛЬНО-ГУМАНИТАРНЫЙ КОЛЛЕДЖ**



**Дисциплина  
«Конструирование программ и языки программирования»**

**Разработка программ обработки XML-документов  
(4 часа)**

Методические рекомендации к лабораторной работе № 18

Могилев 2018

Составитель: Шлапаков А.В.

Понятие «XML-документ». Методические указания по лабораторной работе № 18 по дисциплине «Конструирование программ и языки программирования». Для учащихся очной формы обучения специальности 1–40 01 01 «Программное обеспечение информационных технологий».

## Оглавление

1 Цель работы.....	4
2 Краткие теоретические сведения .....	5
2.1 Теги языка комментариев XML .....	5
2.2 Создание XML-файла .....	7
2.2.1. Как создать XML-файл в Visual Studio вручную .....	7
2.2.2. Как создать XML-файл в блокноте .....	8
2.3 Компиляция XML-документа .....	9
2.4 Работа с XML с помощью классов System.Xml .....	11
2.4.1. Изменение XML-документа .....	14
2.5 Класс XmlReader и XmlWriter .....	15
3 Задания.....	29
4 Контрольные вопросы.....	31

## **1 Цель работы**

выработать умение разрабатывать программы обработки XML-документов.

## 2 Краткие теоретические сведения

XML – это так называемый язык разметки, который используется для представления иерархических данных и структурирования передаваемой информации в удобной для программ, приложений и различных систем форме. XML универсален и не зависит от платформы, на которой используется. Как и у любого языка, у него есть свой собственный (но довольно несложный) синтаксис, представленный в основном в виде различных тегов. Внешне он несколько похож на язык гипертекстовой разметки HTML, но если HTML используется для разметки страниц и отображения данных на них, то XML используется для структурирования данных, их сортировки, фильтрации и т.п. Можно даже приписать XML к аналогу баз данных, который не требует СУБД. В особенности XML-файлы хороши в представлении древовидных (иерархических) БД с произвольной длиной ветвей. В большинстве современных реляционных БД реализация подобной структуры далеко не всегда бывает эффективной и удобной, в особенности, когда длина ветвей достигает до десятков элементов.

Ко всему прочему, XML-документ довольно легко создать, для этого не нужно иметь никаких особенных программ и специфического ПО. В самом простом случае можно обойтись даже стандартной программой “Блокнот”. А при наличии Visual Studio, предпочтительнее воспользоваться им, так как работа с XML-файлами в нём очень удобна и проста. Кроме того, есть возможность заполнить XML-файл программно, используя, например, консольное приложение Visual Studio.

Язык C# поддерживает три типа комментариев. Первые два характеризуются символами // и /\* \*/. Третий тип основан на XML-тегах и называется XML-комментарием. (XML-комментарии также называются комментариями документации.) Каждая строка XML-комментария начинается с символов ///, XML-комментариями обычно предваряется определение классов, пространств имен, методов, свойств и событий. Используя XML-комментарии, можно встраивать информацию о программе в саму программу. При компиляции такой программы вы получите XML-комментарии собранными в XML-файле. XML-комментарии можно утилизировать с помощью средства IntelliSense системы Visual Studio.

### 2.1 Теги языка комментариев XML

C# поддерживает теги XML-документации, представленные в табл. 1. Назначение большинства из них можно понять по описанию, да и работают они подобно другим XML-тегам, с которыми многие программисты уже знакомы. Однако тег <list> (<список>) несколько сложнее других. Список содержит два компонента: заголовок списка и элементы списка. Базовый формат заголовка списка такой:

```
<listheader>
<term> имя </term>
<description> текст </description>
</listheader>
```

Здесь текст описывает имя. Для таблицы элемент текст не используется. Базовый формат элемента списка таков:

```
<item>
```

```

<term> имя_элемента </term>
<description> текст </description>
</item>

```

Здесь текст описывает имя\_элемента. Для маркированных или нумерованных списков или таблиц элемент имя\_элемента не используется. Можно использовать несколько тегов <item>.

Таблица 1. Теги языка комментариев XML

Тег	Описание
<c> код </c>	Определяет текст, заданный элементом код, как программный код
<code> код </code>	Определяет несколько строк текста, заданных элементом код, как программный код
<example> трактовка </example>	Текст, соответствующий элементу трактовка, описывает пример кода
<exception cref="имя"> Explanation </exception>	Описывает исключительную ситуацию, заданную элементом имя
include file = 'имя_файла' path ='путь [@имя_тега = "ID_тега"]' />	Определяет файл, содержащий XML-комментарии для текущего файла. Файл задается элементом имя_файла. Каталогический путь к тегу, имя тега и ID тега задаются элементами путь, имя_тега, и ID_тега, соответственно
<list type = "тип"> заголовок_списка элементы_списка </list>	Определяет список. Тип списка задается элементом тип, который может принимать одно из следующих значений: bullet (маркированный), number (нумерованный) или table (таблица)
<para> текст </para>	Определяет абзац текста в другой тег
<param name = 'имя_параметра' > Трактовка </param>	Описывает параметр, заданный элементом имя_параметра. Описание содержится в тексте, соответствующем элементу трактовка
<paramref name = "имя_параметра" />	Означает, что элемент имя_параметра представляет собой имя параметра
<permission cref = "идентификатор"> трактовка </permission>	Описывает параметр разрешения, связанный с членами класса, заданными элементом идентификатор. Параметры разрешения описаны в тексте, соответствующем элементу трактовка
<remarks>трактовка </remarks>	Текст, заданный элементом трактовка, представляет собой общие комментарии, которые часто используются для описания класса или структуры

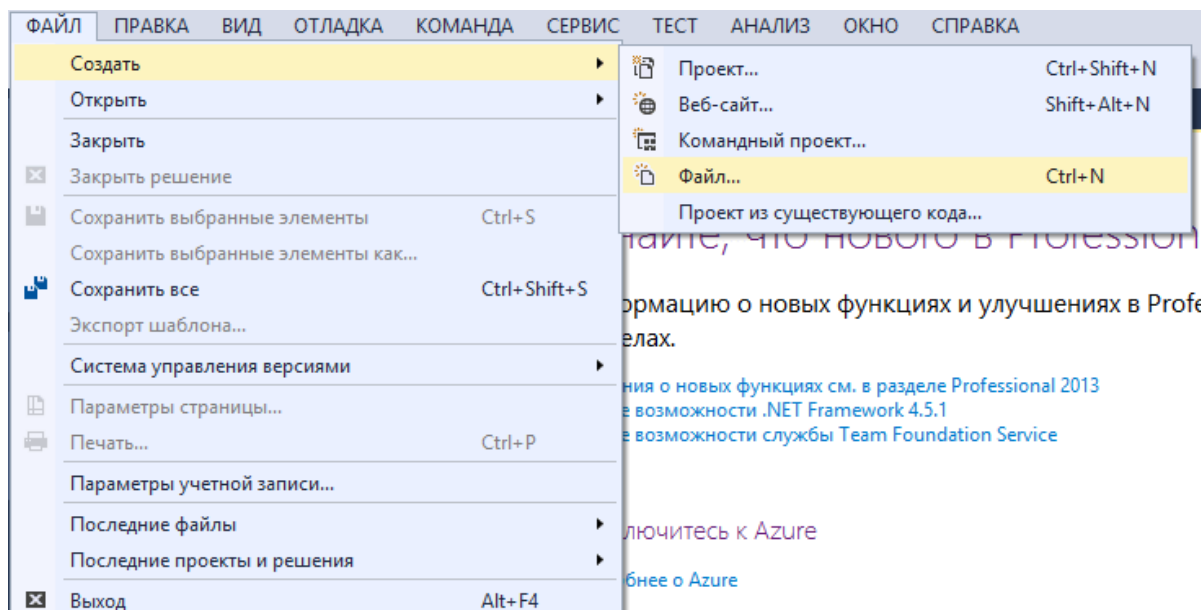
<returns> трактовка </returns>	Текст, заданный элементом трактовка, описывает значение, возвращаемое методом
<see cref = "идентификатор" />	Объявляет ссылку на другой элемент, заданный элементом идентификатор
<seealso cref = "идентификатор" />	Объявляет перекрестную ссылку типа "см. также" на идентификатор
<summary> трактовка </summary>	Текст, заданный элементом трактовка, представляет собой общие комментарии, которые часто используются
<value> трактовка </value>	Текст, заданный элементом трактовка, описывает свойство

## 2.2 Создание XML-файла

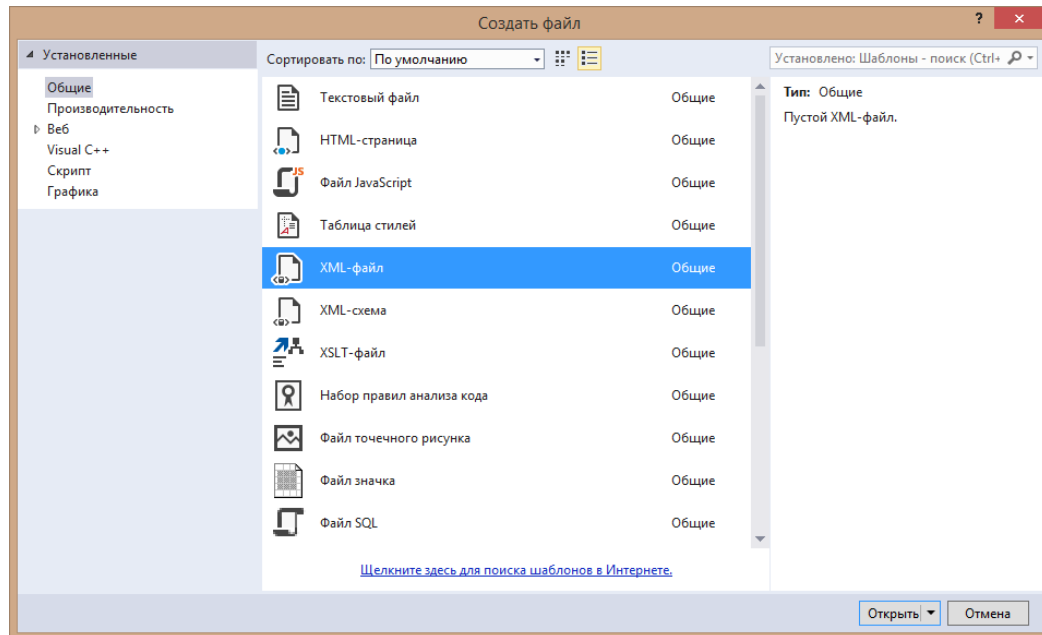
### 2.2.1. Как создать XML-файл в Visual Studio вручную

Для начала попробуем создать простенький XML-файл вручную в Visual Studio и в блокноте.

Чтобы создать пустой XML-документ в Visual Studio, следует в окне программы нажать вкладку “Файл”, а затем выбрать строки “Создать” и “Файл”, как показано на скриншоте:



Затем в появившемся окне следует выбрать “XML-файл” и нажать кнопку “Открыть”.



Вот и всё, XML-документ создан. Изначально в нём записана лишь одна строка:

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

Как легко понять, она означает, что созданный файл поддерживает язык XML версии 1.0 и работает с кодировкой UTF-8.

Теперь давайте запишем в него какие-нибудь данные. Например, такие:

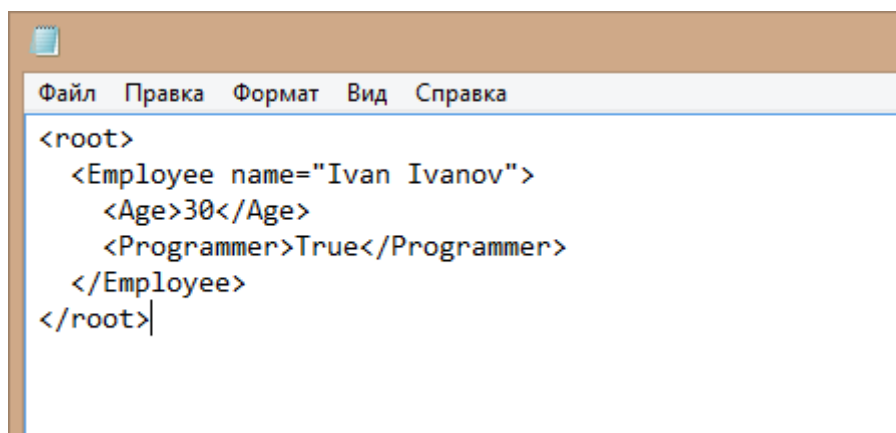
```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <Employee name="Ivan Ivanov">
    <Age>30</Age>
    <Programmer>True</Programmer>
  </Employee>
</root>
```

Теги в XML можно именовать так, как удобно пользователю, что сильно упрощает визуализацию XML-файлов для людей. В данном примере у нас имеются иерархические данные в три ступени. Сначала идёт структура каталога, обозначенная тегами `<root></root>`. В неё вложен каталог с названием `Employee` и атрибутом `name`, которому присвоено значение “Ivan Ivanov”. А в этот каталог вложены данные `Age` и `Programmer`. Иными словами, у нас теперь в корне структуры данных имеется некоторый сотрудник Иван Иванов, которому 30 лет и который является программистом. Таких сотрудников может быть очень много, как может быть много и атрибутов этих сотрудников, а также атрибутов у атрибутов этих сотрудников и даже атрибутов у атрибутов у атрибутов этих сотрудников. Примерно так и выглядят XML-файлы. Данный способ создания XML-файла довольно прост и удобен в применении. Если же под рукой не оказалось Visual Studio или другой специализированной программы, то можно обойтись и простым блокнотом.

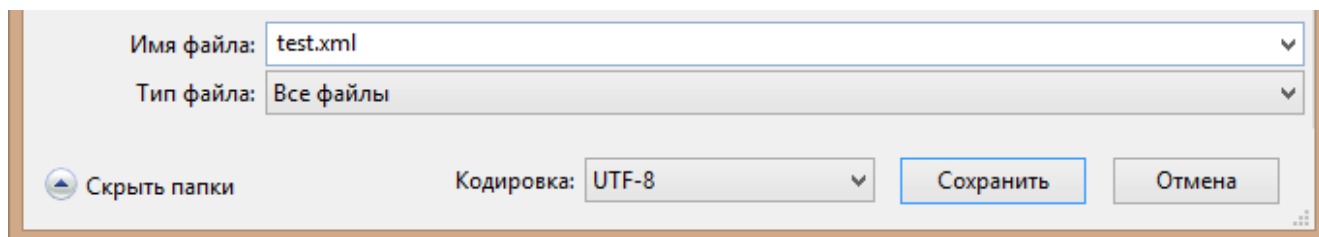
### 2.2.2. Как создать XML-файл в блокноте



Создавать какие-либо большие и объемные XML-файлы в блокноте будет довольно затруднительно, есть возможность запутаться и ошибиться во всём обилии тегов и данных, однако создать простенькую базу вполне возможно. Для этого надо лишь открыть Блокнот и просто начать писать. Например, ту же структуру, что и выше:



В принципе, можно обойтись и без строки о версии XML-языка и без присваивания кодировки, которая создавалась автоматически в примере с Visual Studio выше, но лучше будет записать и её. Чтобы сохранить данный файл правильно, надо при сохранении выбрать “Тип файла” – Все файлы, а к имени файла прибавить расширение .xml. Кроме того, стоит выбрать кодировку “UTF-8”.



Таким образом и создаются XML-файлы вручную.

### 2.3 Компиляция XML-документа

Чтобы создать XML-файл, который содержит комментарии к документу, задайте опцию /doc. Например, чтобы скомпилировать файл DocTest.cs, содержащий XML-комментарии, используйте такую командную строку:

```
csc DocTest.cs /doc:DocTest.xml
```

Чтобы создать выходной XML-файл при использовании интегрированной среды Visual Studio IDE, необходимо использовать диалоговое окно Property Pages (Страницы свойств), которое активизируется при выборе команды View, Property Pages (Вид, Страницы свойств). Затем выберите команду Configuration Properties, Build (Свойства конфигурации, Построить). После этого укажите имя XML-файла в свойстве XML Documentation File (XML-файл документации).

Пример XML-документа

Рассмотрим пример использования XML-комментариев:

```
// Пример XML-документа.
using System;
/// <remark>
/// Это пример XML-документа.
/// В классе Test показано использование ряда тегов.
/// </remark>

class Test {
/// <summary>
/// Выполнение начинается с метода Main().
/// </summary>
public static void Main() { int sum;
sum = Summation(5);
Console.WriteLine("Сумма последовательных " + 5 + " чисел равна " +
sum);
}
/// <summary>
/// Метод Summation() возвращает сумму ряда чисел.
/// <param name = "val">
/// Последнее слагаемое передается в параметре val.
/// </param>
/// <see cref="int"> </see>
/// <returns>
/// Результат возвращается как int-значение.
/// </returns>
/// </summary>
static int Summation(int val) { int result = 0;
for(int i=1; i <= val; i++) result += i;
return result;
}
}
```

Предположим, что приведенная выше программа называется `XmlTest.cs`. с помощью следующей строки

```
csc XmlTest.cs /doc:XmlTest.xml
```

эту программу можно скомпилировать, а в результате компиляции будет создан файл `XmlTest.xml`, который должен содержать следующие комментарии:

```
<?xml version="1.0"?>
<doc>
<assembly>
<name>t</name>
</assembly>
<members>
```

```
<member name="T:Test">  
<remark>
```

Это пример XML-документа.

В классе `Test` показано использование ряда тегов.

```
</remark>  
</member>  
<member name="M:Test.Main">  
<summary>
```

Выполнение начинается с метода `Main()`.

```
</summary>  
</member>  
<member name="M:Test.Summation(System. Int32) ">  
<summary>
```

Метод `Summation()` возвращает сумму ряда чисел.

```
<param name="val">
```

Последнее слагаемое передается в параметре `val`.

```
</param>  
<see cref="T:System.Int32"> </see>  
<returns>
```

Результат возвращается как `int`-значение.

```
</returns>  
</summary>  
</member>  
</members>  
</doc>
```

Обратите внимание на то, что каждому документированному элементу присваивается уникальный идентификатор. Эти идентификаторы могут использовать другие программы, которые включают XML-комментарии.

## 2.4 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>42</age>
  </user>
</users>
```

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

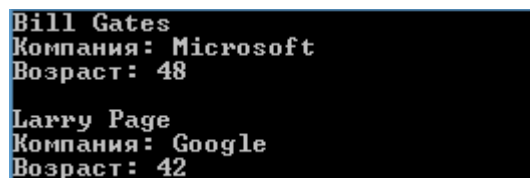
```
using System;
using System.Xml;
class Program
{
    static void Main(string[] args)
    {
        XmlDocument xDoc = new XmlDocument();
        xDoc.Load("users.xml");
        // получим корневой элемент
```

```

XmlElement xRoot = xDoc.DocumentElement;
// обход всех узлов в корневом элементе
foreach(XmlNode xnode in xRoot)
{
    // получаем атрибут name
    if(xnode.Attributes.Count>0)
    {
        XmlNode attr =
xnode.Attributes.GetNamedItem("name");
        if (attr!=null)
            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: {0}", child-
node.InnerText);
        }
        // если узел age
        if (childnode.Name == "age")
        {
            Console.WriteLine("Возраст: {0}", child-
node.InnerText);
        }
    }
    Console.WriteLine();
}
Console.Read();
}
}

```

В итоге получим следующий вывод на консоли:



```

Bill Gates
Компания: Microsoft
Возраст: 48

Larry Page
Компания: Google
Возраст: 42

```

Чтобы начать работу с документом xml, необходимо создать объект XmlDocument и затем загрузить в него xml-файл: xDoc.Load("users.xml");

При разборе xml для начала получим 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 кор-

нового 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:

```
List<User> users = new List<User>();
XmlDocument xDoc = new XmlDocument();
xDoc.Load("users.xml");
XmlElement xRoot = xDoc.DocumentElement;
foreach(XmlElement xnode in xRoot)
{
    User user = new User();
    XmlNode attr = xnode.Attributes.GetNamedItem("name");
    if (attr!=null)
        user.Name=attr.Value;

    foreach (XmlNode childnode in xnode.ChildNodes)
    {
        if (childnode.Name == "company")
            user.Company=childnode.InnerText;

        if (childnode.Name == "age")
            user.Age = Int32.Parse(childnode.InnerText);
    }
    users.Add(user);
}
foreach (User u in users)
    Console.WriteLine("{0} ({1}) - {2}", u.Name, u.Company,
u.Age);
```

### 2.4.1. Изменение XML-документа

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

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 XmlElement, унаследованный от XmlNode, добавляет еще ряд методов, которые позволяют создавать новые узлы:

CreateNode: создает узел любого типа

CreateElement: создает узел типа XmlDocument

CreateAttribute: создает узел типа XmlAttribute

CreateTextNode: создает узел типа XmlNode

CreateComment: создает комментарий

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

```
XmlDocument xDoc = new XmlDocument();
xDoc.Load("users.xml");
XmlElement xRoot = xDoc.DocumentElement;
// создаем новый элемент user
XmlElement userElem = xDoc.CreateElement("user");
// создаем атрибут name
XmlAttribute nameAttr = xDoc.CreateAttribute("name");
// создаем элементы company и age
XmlElement companyElem = xDoc.CreateElement("company");
XmlElement ageElem = xDoc.CreateElement("age");
// создаем текстовые значения для элементов и атрибута
XmlText nameText = xDoc.CreateTextNode("Mark Zuckerberg");
XmlText companyText = xDoc.CreateTextNode("Facebook");
XmlText ageText = xDoc.CreateTextNode("30");

//добавляем узлы
nameAttr.AppendChild(nameText);
companyElem.AppendChild(companyText);
ageElem.AppendChild(ageText);
userElem.Attributes.Append(nameAttr);
userElem.AppendChild(companyElem);
userElem.AppendChild(ageElem);
xRoot.AppendChild(userElem);
xDoc.Save("users.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("users.xml")

После этого в xml-файле появится следующий элемент:

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

## 2.5 Класс XmlReader и XmlWriter

Когда размер XML-данных доходит до нескольких сотен мегабайт, а то и гигабайт, классы `XmlDocument` или `XPathDocument` использовать просто неприлично, а в некоторых случаях и вовсе невозможно, ведь приложение будет требовать объем оперативной памяти равный размеру данных. Именно для таких случаев и предназначен класс `XmlReader`.

В отличие от своих собратьев по пространству имен, класс `XmlReader` не хранит данные в памяти. Доступ к данным осуществляется последовательно. Но из-за последовательности становится невозможным узнать, что содержит тот или иной XML-элемент, до того, как данные будут прочитаны. Этот недостаток делает класс `XmlReader` неудобным в использовании и даже может отпугнуть неопытных программистов. Но все не так страшно, как может показаться на первый взгляд.

Класс `XmlReader` не имеет конструктора. Создать новый экземпляр класса `XmlReader` можно при помощи метода `Create`, который принимает либо физический путь или URL к XML-документу, либо поток (`Stream`).

```
XmlReader xml = XmlReader.Create("XmlFile1.xml");
```

Для перехода от одного узла к другому предназначена функция `Read`, которая возвращает либо `true`, если удалось осуществить переход к узлу, либо, по достижению конца документа, `false`. Как правило, чтение всего документа производится циклом.

```
while (xml.Read())
{
    // ...
}
```

В зависимости от типа узла (`NodeType`), экземпляр класса `XmlReader` может содержать дополнительно имя узла (`Name`) и его текстовое значение (`Value`).

```
Console.WriteLine("Тип узла: {0}", xml.NodeType);
Console.WriteLine("Имя узла: {0}", xml.Name);
Console.WriteLine("Значение узла: {0}", xml.Value);
```

Для проверки наличия данных, предназначены вспомогательные свойства: `HasValue` – указывает, имеет узел текстовое значение (`Value`) или нет; `HasAttributes` – указывает, имеет узел атрибуты или нет; `IsEmptyElement` – содержит `true`, если элемент не имеет никаких данных.

Если текущий узел содержит атрибуты (`HasAttributes = true`), то получить их можно при помощи функции `MoveToNextAttribute`, которая, по аналогии с `Read`, будет возвращать `true`, до достижения конца определения элемента. Имя атрибута и значение также будут находиться в свойствах `Name` и `Value`.

```
if (xml.HasAttributes)
{
    while (xml.MoveToNextAttribute())
    {
        Console.WriteLine("Атрибут: {0}", xml.Name);
    }
}
```



```

        Console.WriteLine("Значение: {0}", xml.Value);
    }
}

```

После считывания атрибутов может потребоваться вернуться назад к элементу, для этого существует метод `MoveToElement`.

```

if (xml.MoveToNextAttribute())
{
    Console.WriteLine("Атрибут: {0}", xml.Name);
    xml.MoveToElement();
    Console.WriteLine("Элемент: {0}", xml.Name);
}

```

Для упрощения обработки документа есть несколько полезных функций. Методы `ReadOuterXml` и `ReadInnerXml` позволяют получить фрагменты XML в виде текста, как есть. Так, например, найдя в документе элемент `member`, при помощи функции `ReadOuterXml` можно получить его целиком и загрузить в `XmlDocument`, т.е. обработать данные на более высоком уровне, а не собирать их по крупинкам, как в продемонстрированном ранее примере.

```

if (xml.NodeType == XmlNodeType.Element)
{
    if (xml.Name == "member")
    {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(xml.ReadOuterXml());
        XmlNode n = xmlDoc.SelectSingleNode("member");
        Console.WriteLine("KUID: {0}",
n.Attributes["kuid"].Value);
        Console.WriteLine("Псевдоним:{0}",
n.SelectSingleNode("nickname").InnerText);
        Console.WriteLine("Имя:{0}",
n.SelectSingleNode("firstName").InnerText);
        Console.WriteLine("Фамилия:{0}",
n.SelectSingleNode("lastName").InnerText);
        Console.WriteLine("-----");
    }
}

```

А при желании, можно вообще сделать класс `member` и создать его экземпляр на основе полученной из xml информации при помощи механизмов десериализации .NET Framework.

```

[XmlRoot("member")]

```

```

public class Member
{
    [XmlAttribute("kuid")]
    public int KUID { get; set; }
    [XmlElement("nickname")]
    public string Nickname { get; set; }
    [XmlElement("firstName")]
    public string FirstName { get; set; }
    [XmlElement("lastName")]
    public string LastName { get; set; }
    public Member() { }
    public Member(string xml)
    {
        LoadXml(xml);
    }
    public void LoadXml(string source)
    {
        XmlSerializer mySerializer = new XmlSerializ-
er(this.GetType());
        using (MemoryStream ms = new
MemoryStream(Encoding.UTF8.GetBytes(source)))
        {
            object obj = mySerializer.Deserialize(ms);
            foreach (PropertyInfo p in
obj.GetType().GetProperties())
            {
                PropertyInfo p2 =
this.GetType().GetProperty(p.Name);
                if (p2 != null && p2.CanWrite)
                {
                    p2.SetValue(this, p.GetValue(obj, null),
null);
                }
            }
        }
    }
}

```

Примечание. Для использования классов XML-сериализации необходимо импортировать в проект пространство имен **System.Xml.Serialization**. Помимо этого, для работы функции **LoadXml** может потребоваться импортировать пространства имен **System.IO** и **System.Reflection**.

Таким образом, обработка данных выходит на еще более высокий уровень.

```

using (XmlReader xml = XmlReader.Create("XmlFile1.xml"))
{
    while (xml.Read())
    {
        switch (xml.NodeType)
        {
            case XmlNodeType.Element:
                // нашли элемент member
                if (xml.Name == "member")
                {
                    // передаем данные в класс Member
                    Member m = new Mem-
ber(xml.ReadOuterXml());

                    Console.WriteLine("KUID: {0}", m.KUID);
                    Console.WriteLine("Псевдоним: {0}",
m.Nickname);

                    Console.WriteLine("Имя: {0}",
m.FirstName);

                    Console.WriteLine("Фамилия: {0}",
m.LastName);

                    Console.WriteLine("-----
-----");
                }
                break;
            }
        }
    }
}

```

При этом уровень обработки данных повышается не для всего содержимого XML, объем которого может достигать многих гигабайт, а лишь для его небольшого фрагмента. Потребление памяти, да и в целом нагрузка на компьютер, незначительные. А вот что делать, если элемент `member` будет содержать большие объемы вложенной информации. Например, все сообщения пользователя с каких-либо форумов?

```

<shk>
<members>
<member kuid="1">
    <nickname>Алексей</nickname>
    <firstName>Алексей</firstName>
    <lastName>Немиро</lastName>
    <messages>
    <message id="1">
        <subject>Привет мир!</subject>
        <text>Это тестовое сообщение.</text>
    
```

```

</message>
<message id="2">
  <subject>Как работать с XmlReader?</subject>
  <text>Это текст о том, как работать с XmlReader.</text>
</message>
</messages>
</member>
<member kuid="288">
  <nickname>Игорь Голов</nickname>
  <firstName>Игорь</firstName>
  <lastName>Голов</lastName>
</member>
<member kuid="1858">
  <nickname>[i]Pro</nickname>
  <firstName>Арте́м</firstName>
  <lastName>Донцов</lastName>
  <messages>
    <message id="3">
      <subject>Еще одно сообщение</subject>
      <text>Это текст еще одного тестового сообщения.</text>
    </message>
  </messages>
</member>
<member kuid="2575">
  <nickname>Shark1</nickname>
  <firstName>Виталий</firstName>
  <lastName />
  <messages />
</member>
</members>
</shk>

```

В этом документе, каждый элемент `member` содержит один вложенный элемент `messages`, который в свою очередь может состоять из неограниченного числа элементов `message`. Если получать все содержимое узла `member`, то потребление оперативной памяти может быть высоким. Поэтому использование функции `ReadOuterXml` для этих целей не годится. В подобных случаях нужно разделить логику обработки элементов `member` и `messages`. Таким образом, чтобы элемент `member` содержал только основные данные о пользователе, как и в первых примерах, исключив из него элемент `messages`. Но и список сообщений никуда не пропадает, он просто будет обработан отдельно, причем ничто не заставляет использовать для этого метод `ReadOuterXml`.

Чтобы исключить объемные фрагменты XML, необходимо записывать в память только то, что нужно. Для этого потребуется создать экземпляр класса `XmlWriter`. Запись данных проще всего производить в `StringBuilder`, хотя можно и в поток (`Stream`). Также

потребуется две дополнительных булевых (логических) переменных, одна из которых будет разрешать запись данных в экземпляр `XmlWriter`, а вторая – информировать об исключении XML-элемента.

```
StringBuilder sb = null;
XmlWriter w = null;
XmlWriterSettings ws = new XmlWriterSettings() { Encoding = Encoding.UTF8 };
bool isMember = false, isSkipped = false;
```

Пропустить ненужный фрагмент xml можно при помощи метода `Skip`.

```
using (XmlReader xml = XmlReader.Create("XmlFile1.xml"))
{
    while (xml.Read())
    {
        switch (xml.NodeType)
        {
            case XmlNodeType.Element:
                // нашли элемент member
                if (xml.Name == "member")
                {
                    isMember = true;
                    // в памяти есть данные пользователя
                    if (sb != null)
                    {
                        w.Flush();
                        w.Close();
                        // обрабатываем их
                        XmlDocument xmlDoc = new XmlDocument();
                        xmlDoc.LoadXml(sb.ToString());
                        XmlNode n =
                        xmlDoc.SelectSingleNode("member");
                        Console.WriteLine("KUID: {0}",
                        n.Attributes["kuid"].Value);
                        Console.WriteLine("Псевдоним: {0}",
                        n.SelectSingleNode("nickname").InnerText);
                        Console.WriteLine("Имя: {0}",
                        n.SelectSingleNode("firstName").InnerText);
                        Console.WriteLine("Фамилия: {0}",
                        n.SelectSingleNode("lastName").InnerText);
                        Console.WriteLine("-----
                        -----");
                    }
                }
            }
}
```

```

// создаем новый XmlWriter, для записи
данных пользователя

sb = new StringBuilder();
w = XmlWriter.Create(sb, ws);
w.WriteProcessingInstruction("xml",
"version=\"1.0\" encoding=\"utf-8\"");
}
else if (xml.Name == "messages")
{
    // пропускаем фрагмент с сообщениями
    xml.Skip();
    // ставим отметку, что фрагмент пропущен
    isSkipped = true;
}
// это данные пользователя и не пропущенный
фрагмент

if (isMember && !isSkipped)
{
    w.WriteStartElement(xml.Name);
    // если есть атрибуты, записываем их
    if (xml.HasAttributes)
    {
        while (xml.MoveToNextAttribute())
        {
            w.WriteAttributeString(xml.Name,
xml.Value);
        }
    }
}
// убираем отметку о пропущенном фрагменте
isSkipped = false;
break;
case XmlNodeType.Text:
    if (isMember)
    {
        w.WriteString(xml.Value);
    }
    break;
case XmlNodeType.EndElement:
    if (isMember)
    {
        w.WriteFullEndElement();
    }
    if (xml.Name == "member")

```

```

        {
            isMember = false;
        }
        break;
    }
}
}

```

По функционалу приведенный листинг ничем не отличается от предыдущих. На выходе также будет получен фрагмент документа, содержащий только элемент `member`, включая вложенный элемент `messages`. Что касается обработки `messages`, то для этого проще всего сделать отдельный `XmlReader` при помощи функции `ReadSubtree`.

```

else if (xml.Name == "messages")
{
    XmlReader xml2 = xml.ReadSubtree();
    while (xml2.Read())
    {
        switch (xml2.NodeType)
        {
            case XmlNodeType.Element:
                // нашли элемент message
                if (xml2.Name == "message")
                {
                    XmlDocument xmlDoc2 = new XmlDocument();
                    xmlDoc2.LoadXml(xml2.ReadOuterXml());
                    XmlNode n =
xmlDoc2.SelectSingleNode("message");
                    Console.WriteLine("Сообщение # {0}",
n.Attributes["id"].Value);
                    Console.WriteLine("Тема: {0}",
n.SelectSingleNode("subject").InnerText);
                    Console.WriteLine("Текст: {0}",
n.SelectSingleNode("text").InnerText);
                    Console.WriteLine("-----");
                }
                break;
            }
        }
        // пропускаем фрагмент с сообщениями
        xml.Skip();
        // ставим отметку, что фрагмент пропущен
        isSkipped = true;
    }
}

```

Таким образом, у нас будет отдельно обработан каждый элемент **member** и каждый связанный с ним **message**. Это практически никак не отразится на потреблении ресурсов, каким бы большим не был размер xml-файла.

По завершению работы с документом, всегда необходимо закрывать поток при помощи метода **Close**.

Примечание. При использовании инструкции **using { }**, закрытие потока происходит автоматически. Отдельный вызов метода **Close** может привести к возникновению исключения.

Следующий пример демонстрирует чтение документа xml.

```
using System;
using System.Xml;
namespace ReadXml1
{
    class Class1
    {
        static void Main(string[] args)
        {

            XmlTextReader textReader = new XmlTextReader("D:\\my.xml");
            textReader.Read();
            // If the node has value
            while (textReader.Read())
            {
                // Move to first element
            textReader.MoveToElement();
            Console.WriteLine("XmlTextReader Properties Test");
            Console.WriteLine("=====");

            // Read this element's properties and display them on console

            Console.WriteLine("Name:" + textReader.Name);
            Console.WriteLine("Base URI:" + textReader.BaseURI);
            Console.WriteLine("Local Name:" +textReader.LocalName);
            Console.WriteLine("Attribute Count:" + textReader.AttributeCount.ToString());
            Console.WriteLine("Depth:" + textReader.Depth.ToString());
            Console.WriteLine("Line Number:" + textReader.LineNumber.ToString());
            Console.WriteLine("Node Type:" + textReader.NodeType.ToString());
            Console.WriteLine("Attribute Count:" + textReader.Value.ToString());
            }
        }
    }
}
```



Следующий пример еще более подробный.

```
using System;
using System.Xml;
namespace ReadingXML2
{
    class Class1
    {
        static void Main(string[] args)
        {
            int ws = 0;
            int pi = 0;
            int dc = 0;
            int cc = 0;
            int ac = 0;
            int et = 0;
            int el = 0;
            int xd = 0;
            // Read a document
            XmlTextReader textReader = new XmlTextReader("D:\\my.xml");
            // Read until end of file
            while (textReader.Read())
            {
                XmlNodeType nType = textReader.NodeType;
                // If node type is a declaration
                if (nType == XmlNodeType.XmlDeclaration)
                {
                    Console.WriteLine("Declaration:" + textReader.Name.ToString());
                    xd = xd + 1;
                }
                // if node type is a comment
                if (nType == XmlNodeType.Comment)
                {
                    Console.WriteLine("Comment:" + textReader.Name.ToString());
                    cc = cc + 1;
                }
                // if node type is an attribute
                if (nType == XmlNodeType.Attribute)
                {
                    Console.WriteLine("Attribute:" + textReader.Name.ToString());
                    ac = ac + 1;
                }
                // if node type is an element
                if (nType == XmlNodeType.Element)
                {
                    Console.WriteLine("Element:" + textReader.Name.ToString());
                }
            }
        }
    }
}
```

```

        el = el + 1;
    }
    // if node type is an entity\
    if (nType == XmlNodeType.Entity)
    {
        Console.WriteLine("Entity:" + textReader.Name.ToString());
        et = et + 1;
    }
    // if node type is a Process Instruction
    if (nType == XmlNodeType.Entity)
    {
        Console.WriteLine("Entity:" + textReader.Name.ToString());
        pi = pi + 1;
    }
    // if node type a document
    if (nType == XmlNodeType.DocumentType)
    {
        Console.WriteLine("Document:" + textReader.Name.ToString());
        dc = dc + 1;
    }
    // if node type is white space
    if (nType == XmlNodeType.Whitespace)
    {
        Console.WriteLine("WhiteSpace:" + textReader.Name.ToString());
        ws = ws + 1;
    }
}
// Write the summary
Console.WriteLine("Total Comments:" + cc.ToString());
Console.WriteLine("Total Attributes:" + ac.ToString());
Console.WriteLine("Total Elements:" + el.ToString());
Console.WriteLine("Total Entity:" + et.ToString());
Console.WriteLine("Total Process Instructions:" + pi.ToString());
Console.WriteLine("Total Declaration:" + xd.ToString());
Console.WriteLine("Total DocumentType:" + dc.ToString());
Console.WriteLine("Total WhiteSpaces:" + ws.ToString());
    }
}
}

```

Создание xml-документа реализует следующий код. Изучите и выполните его.

```

using System;
using System.Xml;
namespace ReadingXML2
{

```

```

class Class1
{
    static void Main(string[] args)
    {
        // Create a new file in C:\\ dir
        XmlTextWriter textWriter;
        textWriter = new XmlTextWriter("C:\\myXmlFile.xml", null);
        // Opens the document
        textWriter.WriteStartDocument();
        // Write comments
        textWriter.WriteComment("First Comment XmlTextWriter Sam-
ple Example");
        textWriter.WriteComment("myXmlFile.xml in root dir");
        // Write first element
        textWriter.WriteStartElement("Student");
        textWriter.WriteStartElement("r", "RECORD", "urn:record");
        // Write next element
        textWriter.WriteStartElement("Name", "");
        textWriter.WriteString("Student");
        textWriter.WriteEndElement();
        // Write one more element
        textWriter.WriteStartElement("Address", "");
        textWriter.WriteString("Colony");
        textWriter.WriteEndElement();
        // WriteChars
        char[] ch = new char[3];
        ch[0] = 'a';
        ch[1] = 'r';
        ch[2] = 'c';
        textWriter.WriteStartElement("Char");
        textWriter.WriteChars(ch, 0, ch.Length);
        textWriter.WriteEndElement();
        // Ends the document.
        textWriter.WriteEndDocument();
        // close writer
        textWriter.Close();
    }
}

```

Наконец, последний пример показывает. Как превратить файл базы данных в документ XML.

```

using System;
using System.Xml;
using System.Data;
using System.Data.OleDb;
namespace ReadingXML2
{
class Class1

```

```
{
    static void Main(string[] args)
    {
        // create a connection
        OleDbConnection con = new OleDbConnection();
        con.ConnectionString = "Provider=Microsoft.Jet.OLEDB.4.0;Data
        Source=C:\\Northwind.mdb";
        // create a data adapter
        OleDbDataAdapter da = new OleDbDataAdapter("Select *
from Customers", con);
        // create a new dataset
        DataSet ds = new DataSet();
        // fill dataset
        da.Fill(ds, "Customers");
        // write dataset contents to an xml file by calling
        //WriteXml method
        ds.WriteXml("C:\\OutputXML.xml");
    }
}
```

### 3. Задания

1. Реализовать примеры из лабораторной работы.
2. Написать программу:
  - для создания документа XML;
  - для поиска в документе XML;
  - для добавления в документ XML новых узлов;
  - демонстрирующую работу с атрибутами в документе XML;
  - для перевода XML базу данных ACCESS;
  - для перевода базы данных ACCESS в XML;
  - демонстрирующую изменение тегов документа XML;
  - для удаления и замены узлов XML.

XML файл должен будет иметь следующее имя: № лабораторной работы\_Фамилия\_группа. Варианты предметных областей создаваемых XML-документов:

#### **Вариант 1**

Описание лекарственных препаратов.

#### **Вариант 2**

Описание фильмов видеотеки.

#### **Вариант 3**

Список сотрудников колледжа.

#### **Вариант 4**

Список моделей мобильных телефонов.

#### **Вариант 5**

Список учащихся специальности.

#### **Вариант 6**

Список изучаемых дисциплин.

#### **Вариант 7**

Описание жанров музыки.

#### **Вариант 8**

Список видов спорта.

#### **Вариант 9**

Список моделей автомобилей.

#### **Вариант 10**

Описание маршрутов автобусов города.

#### **Вариант 11**

Описание маршрутов маршрутного такси города.

#### **Вариант 12**

Список посетителей поликлиники.

#### **Вариант 13**

Описание факультетов университетов города.

#### **Вариант 14**

Описание изданий библиотечного фонда.

#### **Вариант 15**

Описание техники в текущем кабинете.

Составитель: Шлапаков А.В.

#### **4 Контрольные вопросы**

1. Что такое XML-документ? Какая область его применения?
2. Какие классы предназначены для работы с XML-документами? В чем их отличия?