

# Платформа Microsoft .NET и язык программирования C#



# Урок №11

## Работа с XML-файлами

### Содержание

<b>1. Парсеры XML .....</b>	<b>3</b>
Что такое парсер? .....	3
Цели и задачи парсера? .....	4
DOM- и SAX-парсеры. ....	5
<b>2. Примеры создания XML-документов. ....</b>	<b>7</b>
<b>3. XML-документация. ....</b>	<b>22</b>
Что такое XML-документация? .....	22
Зачем использовать XML-документацию .....	24
Примеры использования. ....	24
<b>Домашнее задание .....</b>	<b>31</b>

# 1. Парсеры XML

Как вы уже, наверное, знаете что XML (*eXtensible Markup Language*) это расширяемый язык разметки, предназначенный для создания специальных текстовых файлов. Содержимое этих файлов представляет собой фрагменты информации, которые связаны между собой иерархически. Благодаря простоте создания, удобству чтения, а также расширяемости, данный тип файлов довольно широко используется в разнообразных программных средствах, в частности при создании распределенных приложений.

Однако особенная структура XML-формата вызывает определенные неудобства при извлечении данных из файлов, так как необходимо осуществить синтаксический анализ текста, что при выполнении этой процедуры вручную требует немало усилий. Поэтому для упрощения жизни разработчиков используются специальные механизмы — синтаксические анализаторы (парсеры).

## Что такое парсер?

**Парсер** — это программа, которая предназначена для синтаксического анализа определенной текстовой информации и преобразования ее к необходимому виду, который требуется для дальнейшей обработки. Одним из применений парсеров является разбор содержимого файла, соответствующего спецификации XML.

## Цели и задачи парсера?

Любой парсер используется для извлечения из входных данных необходимой информации, в нашем случае мы будем рассматривать их использование при работе с файлами в формате XML.

Парсеры XML-документов используются для решения следующих задач:

- разбор содержимого документа по определенному алгоритму (парсинг), то есть получение данных для обработки более удобных, чем исходный текст. В результате парсинга мы получаем набор специализированных объектов для хранения узлов XML-документа в виде бинарного дерева.
- поиск необходимой информации по критериям, которая относится к определенному объекту. Для этих целей был разработан специальный язык — XPath.
- проверка на соблюдение определенных синтаксических правил заполнения XML-документа: использование только определенных тегов, правильное расположение тегов, корректные значения полей и атрибутов. Однако, самым важным является правильное формирование самого XML-документа, и эти правила определяются спецификацией XML Schema рекомендованной консорциумом W3C.
- преобразование XML-документа в другое представление. Формат XML является хорошим решением для хранения и передачи информации, но обработка этой информации связана с определенными трудностями. Поэтому был разработан язык преобразования

XML-документов XSLT (*eXtensible Stylesheet Language Transformations*), который позволяет преобразовать исходный XML-файл в документы различных форматов.

При разборе XML-документов в парсерах применяется один из двух способов парсинга: метод дерева или метод потока.

## DOM- и SAX-парсеры

Наиболее популярными API, которые реализованы большинством XML-парсеров являются (*Document Object Model*) DOM и (*Simple API for XML*) SAX.

DOM-парсер работает по методу дерева и является стандартом W3C, данный парсер совместим с различными языками программирования, что принесло ему заслуженную популярность. DOM-парсер загружает в память весь XML-документ, формируя дерево объектов, которое в точности повторяет дерево элементов оригинального файла, при этом каждый элемент XML-файла соответствует узлу этого дерева, все узлы связаны между собой отношением «родитель-потомок». К достоинствам этого парсера можно отнести отсутствие ограничений на структуру документа, однако построение дерева всего XML-документа требует большого объема памяти и ресурсов процессора.

Корпорация Microsoft предоставляет DOM-парсер *Microsoft XML Core Services* (MSXML), который входит в пакет поставки операционной системы Windows.

SAX-парсер является потоковым парсером, принцип действия, которого заключается в анализе XML-документа

в потоке, то есть в определенный момент времени осуществляется парсинг отдельного узла, по окончании которого полученная информация в памяти не сохраняется. По мере просмотра XML-документа анализатором происходят различные события (начало и конец документа, начало и конец элемента XML-разметки и т.д.), при возникновении которых SAX-парсер связывается с приложением при помощи вызова одного из определенных методов обратного вызова.

SAX-парсер удобно использовать при разборе больших документов, так как не требуется считывания в память всего документа сразу. Основной недостаток этого парсера состоит в том, что просмотр входного потока осуществляется только в прямом направлении, произвольное перемещение по документу невозможно. Кроме того, нельзя легко определить отношения между его элементами, так как обратный вызов SAX-парсера предоставляет крайне мало информации о контексте, в котором этот вызов происходит.

## 2. Примеры создания XML-документов

Пространство имен `System.Xml` предоставляет разнообразные классы для чтения и генерации XML-документов. Если Вам необходимо работать с DOM, то Вы можете использовать класс `XmlDocument`, для чтения XML-документов как потока данных, можете задействовать классы `XmlTextReader`, дополняющий класс `XmlTextWriter` упрощает процесс создания XML-документов. Рассмотрим эти классы подробнее.

Для создания XML-документов предназначен класс `XmlTextWriter`. У этого класса существует несколько методов, начинающихся на `Write`, которые позволяют генерировать различные фрагменты XML, в том числе элементы, атрибуты, комментарии и т.д. В следующем примере некоторые из этих методов используются для создания XML-файла с названием `Cars.xml` (Рисунок 2.1).

```
using System;
using System.Xml;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
```

```

        XmlTextWriter writer = null;

        try
        {
            writer = new XmlTextWriter("Cars.xml",
                                     System.Text.Encoding.Unicode);
            writer.Formatting = Formatting.Indented;
            writer.WriteStartDocument();
            writer.WriteStartElement("Cars");
            writer.WriteStartElement("Car");
            writer.WriteAttributeString("Image",
                                       "MyCar.jpeg");
            writer.WriteElementString("Manufactured",
                                       "La Hispano Suiza de Automovils");
            writer.WriteElementString("Model",
                                       "Alfonso");
            writer.WriteElementString("Year", "1912");
            writer.WriteElementString("Color", "Black");
            writer.WriteElementString("Speed", "130");
            writer.WriteEndElement();
            writer.WriteEndElement();
            WriteLine("The Cars.xml file is generated!");
        }

        catch (Exception ex)
        {
            WriteLine(ex.Message);
        }
        finally
        {
            if (writer != null)
                writer.Close();
        }
    }
}

```



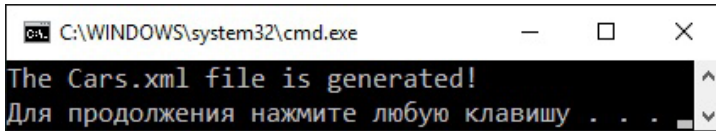


Рисунок 2.1. Создание XML-документа

Полученный файл находится в текущей папке проекта, и будет использоваться нами для примеров этого раздела, его содержимое представлено на рисунке 2.2.

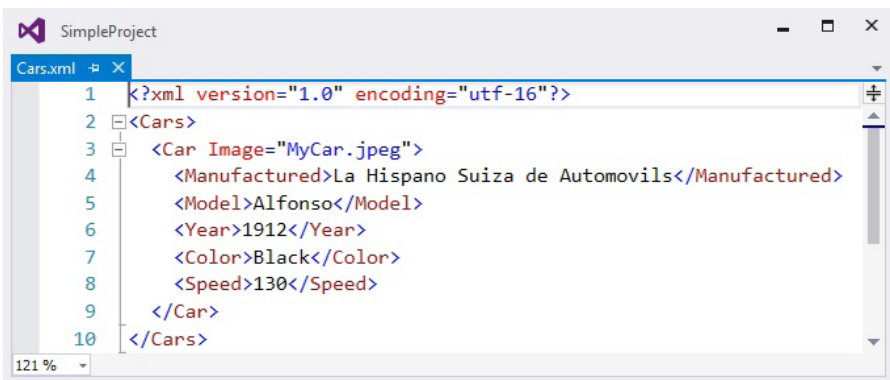


Рисунок 2.2. Содержимое файл Cars.xml

Установка свойству `Formatting` значения `Formatting.Indented` перед началом записи порождает отступы, которые Вы видите в полученном файле. Если этого не сделать, то не генерируются ни отступы, ни переводы строки. По умолчанию количество символов отступа равно двум, а символ, используемый для отступа — пробел. Свойства `Indentation` и `IndentChar` класса `XmlTextWriter` позволяют изменить количество отступов и символ отступа, соответственно.

Класс `XmlDocument` реализует программный интерфейс к XML-документам, которые соответствуют спецификации DOM Level 2 Core, и представляет документ

в виде перевернутого дерева узлов с корневым элементом наверху.

Каждый узел является экземпляром класса `XmlNode`, реализующим методы и свойства для обхода DOM-деревьев, считывания и изменения содержимого узлов, добавления и удаления узлов. `XmlDocument` является производным от `XmlNode` и добавляет собственные методы и свойства, которые поддерживают загрузку и сохранение документов, создание новых узлов и другие операции.

Метод `Load()` класса `XmlDocument` разбирает указанный XML-документ и строит его представление в памяти. Если документ не является правильно оформленным, генерируется `XmlException`. За успешным вызовом метода `Load()` часто следует чтение значения свойства `DocumentElement` объекта `XmlDocument`. `DocumentElement` возвращает ссылку на `XmlNode` для корневого элемента, который является начальной точкой обхода DOM-дерева.

Свойство `HasChildNodes` класса `XmlDocument` позволяет определить, есть ли у данного узла (в том числе у корневого) потомки. Для доступа к потомкам узла служит свойство `ChildNodes`, которое возвращает набор узлов `XmlNodeList`. В следующем примере использование свойств `HasChildNodes` и `ChildNodes` позволяет выполнить рекурсивный просмотр всех узлов дерева (Рисунок 2.3).

```
using System;
using System.Xml;
using static System.Console;

namespace SimpleProject
{
```

```
class Program
{
    static void OutputNode(XmlNode node)
    {
        WriteLine($"Type={node.NodeType}\tName=
                    {node.Name}\tValue={node.Value}");

        if (node.HasChildNodes)
        {
            foreach (XmlNode child in node.ChildNodes)
                OutputNode(child);
        }
    }

    static void Main(string[] args)
    {
        try
        {
            XmlDocument doc = new XmlDocument();
            doc.Load("Cars.xml");
            OutputNode(doc.DocumentElement);
        }

        catch (Exception ex)
        {
            WriteLine(ex.Message);
        }
    }
}
```

В данном примере выводится информация по каждому узлу XML-документа: тип, имя и значение. Для чего используются свойства `NodeType`, `Name` и `Value` объекта `XmlNode`, соответственно.

```

C:\WINDOWS\system32\cmd.exe
Type=Element      Name=Cars      Value=
Type=Element      Name=Car       Value=
Type=Element      Name=Manufactured Value=
Type=Text         Name=#text    Value=La Hispano Suiza de Automovils
Type=Element      Name=Model     Value=
Type=Text         Name=#text    Value=Alfonso
Type=Element      Name=Year      Value=
Type=Text         Name=#text    Value=1912
Type=Element      Name=Color     Value=
Type=Text         Name=#text    Value=Black
Type=Element      Name=Speed     Value=
Type=Text         Name=#text    Value=130
Для продолжения нажмите любую клавишу . . .

```

**Рисунок 2.3.** Рекурсивный просмотр всех узлов дерева

Для узлов некоторых типов (например, элементов) свойство `Name` имеет смысл, а `Value` – нет. В других случаях (в частности, для текстовых узлов) имеет смысл `Value`, а не `Name`. Бывает также (пример тому атрибуты), что имеет смысл оба свойства `Name` и `Value`.

Приведенные выше результаты работы программы не содержат узлов атрибутов, хотя в документе есть два элемента с атрибутами. Дело в том, что атрибуты обрабатываются особым образом, они присутствуют не в списке, возвращаемом свойством `ChildNodes`, а в списке, который возвращается свойством `Attributes`. Внесем изменения в метод `OutputNode()`, чтобы атрибуты выводились наравне с другими типами узлов (Рисунок 2.4).

```

using System;
using System.Xml;
using static System.Console;

namespace SimpleProject
{

```

```

class Program
{
    static void OutputNode(XmlNode node)
    {
        WriteLine($"Type={node.NodeType}\tName=
                    {node.Name}\tValue={node.Value}");

        if (node.Attributes != null)
        {
            foreach (XmlAttribute attr in node.Attributes)
                WriteLine($"Type= {attr.NodeType}\tName=
                            {attr.Name}\tValue={attr.Value}");
        }

        if (node.HasChildNodes)
        {
            foreach (XmlNode child in node.ChildNodes)
                OutputNode(child);
        }
    }

    static void Main(string[] args)
    {
        try
        {
            XmlDocument doc = new XmlDocument();
            doc.Load("Cars.xml");
            OutputNode(doc.DocumentElement);
        }

        catch (Exception ex)
        {
            WriteLine(ex.Message);
        }
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
Type=Element      Name=Cars      Value=
Type=Element      Name=Car       Value=
Type= Attribute   Name=Image     Value=MyCar.jpeg
Type=Element      Name=Manufactured Value=
Type=Text         Name=#text     Value=La Hispano Suiza de Automovils
Type=Element      Name=Model     Value=
Type=Text         Name=#text     Value=Alfonso
Type=Element      Name=Year      Value=
Type=Text         Name=#text     Value=1912
Type=Element      Name=Color     Value=
Type=Text         Name=#text     Value=Black
Type=Element      Name=Speed     Value=
Type=Text         Name=#text     Value=130
Для продолжения нажмите любую клавишу . . .

```

**Рисунок 2.4.** Вывод всей информации об узлах дерева

Класс `XmlDocument` позволяет не только читать, но и изменять XML-документы. В следующем примере демонстрируется возможность формирования нового XML-файла на основе существующего (Рисунок 2.5).

```

using System;
using System.Xml;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                XmlDocument doc = new XmlDocument();
                doc.Load("Cars.xml");

                XmlNode root = doc.DocumentElement;
            }
        }
    }
}

```

```

// удалить первый элемент Cars
root.RemoveChild(root.FirstChild);
// создать узлы элементов.
XmlNode bike =
    doc.CreateElement("Motorcycle");
XmlNode elem1 =
    doc.CreateElement("Manufactured");
XmlNode elem2 = doc.CreateElement("Model");
XmlNode elem3 = doc.CreateElement("Year");
XmlNode elem4 = doc.CreateElement("Color");
XmlNode elem5 =
    doc.CreateElement("Engine");
// создать текстовые узлы
XmlNode text1 =
    doc.CreateTextNode("Harley-Davidson
    Motor Co. Inc.");
XmlNode text2 =
    doc.CreateTextNode("Harley 20J");
XmlNode text3 = doc.CreateTextNode("1920");
XmlNode text4 = doc.CreateTextNode("Olive");
XmlNode text5 = doc.CreateTextNode("37 HP");
// присоединить текстовые узлы
// к узлам элементов
elem1.AppendChild(text1);
elem2.AppendChild(text2);
elem3.AppendChild(text3);
elem4.AppendChild(text4);
elem5.AppendChild(text5);
// присоединить узлы элементов к узлу bike
bike.AppendChild(elem1);
bike.AppendChild(elem2);
bike.AppendChild(elem3);
bike.AppendChild(elem4);
bike.AppendChild(elem5);
// присоединить узел bike к корневому узлу
root.AppendChild(bike);

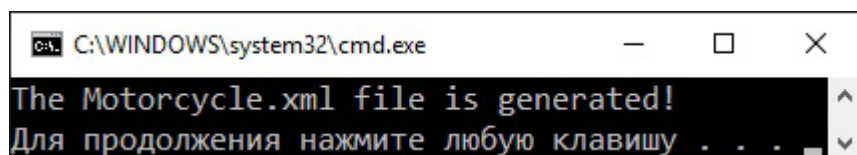
```

```

        doc.Save("Motorcycle.xml"); // сохранить
        //измененный документ

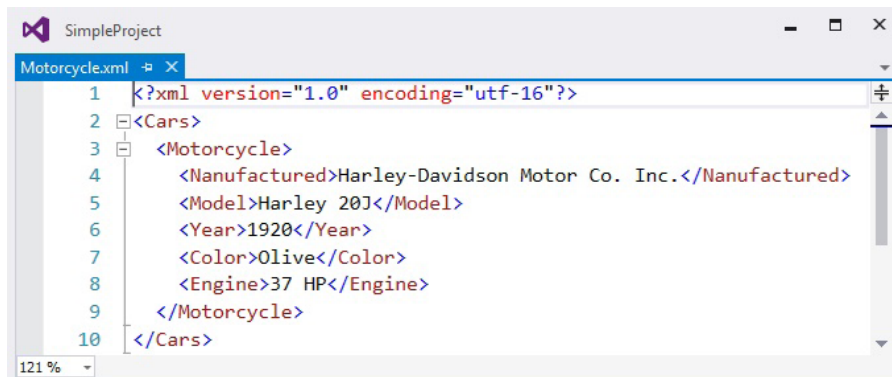
        WriteLine("The Motorcycle.xml file
                    is generated!");
    }
    catch (Exception ex)
    {
        WriteLine(ex.Message);
    }
}
}
}

```



**Рисунок 2.5.** Формирования нового XML-файла

Содержание полученного файла Motorcycle.xml изображено на рисунке 2.6.



**Рисунок 2.6.** Содержание полученного файла



Если нужно просто считать XML и его структура интересует Вас меньше, чем содержимое, то можно использовать класс `XmlTextReader`. Класс `XmlTextReader`, как и `XmlDocument`, находится в пространстве имен `System.Xml` и предоставляет быстрый способ последовательного просмотра XML-документа, он основан на понятии потока данных с возможностью чтения только вперед. Этот класс эффективнее `XmlDocument` по затратам памяти, особенно для больших документов, так как не использует кеширование. Кроме того, он позволяет еще проще, чем `XmlDocument`, сканировать документ в поиске элементов, атрибутов и т.п.

Использовать `XmlTextReader` очень просто, сначала из файла, URL или другого источника данных создается объект `XmlTextReader`, затем последовательно вызывается метод `Read()`, пока не будет найдено нужное содержимое или не будет достигнут конец документа. Каждый вызов метода `Read()` продвигает воображаемый курсор на следующий узел документа. Свойства `XmlTextReader`, такие как `NodeType`, `Name`, `Value` и `AttributeCount`, возвращают информацию о текущем узле. Методы `GetAttribute()`, `MoveToFirstAttribute()` и `MoveToNextAttribute()` позволяют обращаться к атрибутам текущего узла, если таковые имеются. Следующий пример демонстрирует использование класса `XmlTextReader` для считывания всей информации из XML-файла (Рисунок 2.7).

```
using System;
using System.Xml;
using static System.Console;

namespace SimpleProject
{
```

```

class Program
{
    static void Main(string[] args)
    {
        XmlTextReader reader = null;
        try
        {
            reader = new XmlTextReader("Cars.xml");
            reader.WhitespaceHandling =
                WhitespaceHandling.None;
            while (reader.Read())
            {
                WriteLine($"Type={reader.NodeType}\t\
                           tName={reader.Name}\t\
                           tValue={reader.Value}");
                if (reader.AttributeCount > 0)
                {
                    while (reader.MoveToNextAttribute())
                    {
                        WriteLine($"Type={reader.
                                NodeType}\tName={reader.
                                Name}\tValue={reader.Value}");
                    }
                }
            }
        }
        catch (Exception ex)
        {
            WriteLine(ex.Message);
        }
        finally
        {
            if (reader != null)
                reader.Close();
        }
    }
}

```

```

C:\WINDOWS\system32\cmd.exe
Type=XmlDeclaration      Name=xml      Value=version="1.0" encoding="utf-16"
Type=Attribute   Name=version   Value=1.0
Type=Attribute   Name=encoding   Value=utf-16
Type=Element     Name=Cars      Value=
Type=Element     Name=Car      Value=
Type=Attribute   Name=Image    Value=MyCar.jpeg
Type=Element     Name=Manufactured      Value=
Type=Text        Name=      Value=La Hispano Suiza de Automovils
Type=EndElement  Name=Manufactured      Value=
Type=Element     Name=Model      Value=
Type=Text        Name=      Value=Alfonso
Type=EndElement  Name=Model      Value=
Type=Element     Name=Year      Value=
Type=Text        Name=      Value=1912
Type=EndElement  Name=Year      Value=
Type=Element     Name=Color      Value=
Type=Text        Name=      Value=Black
Type=EndElement  Name=Color      Value=
Type=Element     Name=Speed      Value=
Type=Text        Name=      Value=130
Type=EndElement  Name=Speed      Value=
Type=EndElement  Name=Car      Value=
Type=EndElement  Name=Cars      Value=
Для продолжения нажмите любую клавишу . . .

```

**Рисунок 2.7.** Считывания информации из Xml-файла с использованием класса XmlTextReader

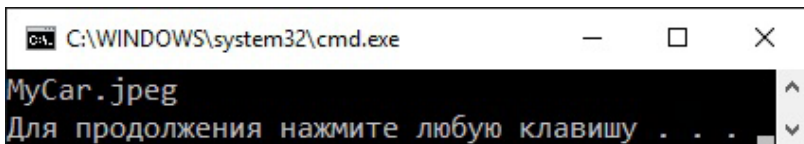
Обратите внимание на узлы `EndElement`, класс `XmlTextReader` в отличие от `XmlDocument` считает начальные и завершающие тэги элемента отдельными элементами. `XmlTextReader` возвращает узлы пустых промежутков, если не указано иное, установка его свойства `WhitespaceHandling` в `WhitespaceHandling.None` подавляет выдачу пустых промежутков. Класс `XmlTextReader` также как и `XmlDocument` не возвращает атрибуты как часть нормального процесса просмотра, просматривать узлы атрибутов нужно отдельно.

Класс `XmlTextReader` часто применяют для извлечения из XML-документов значений заданных узлов. В следующем примере осуществляется поиск всех элементов `Car` с атрибутом `Image` и выводит на консоль значения этих атрибутов (Рисунок 2.8).

```
using System;
using System.Xml;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            XmlTextReader reader = null;
            try
            {
                reader = new XmlTextReader("Cars.xml");
                reader.WhitespaceHandling =
                    WhitespaceHandling.None;
                while (reader.Read())
                {
                    if (reader.NodeType ==
                        XmlNodeType.Element && reader.Name ==
                        "Car" && reader.AttributeCount > 0)
                    {
                        while (reader.MoveToNextAttribute())
                        {
                            if (reader.Name == "Image")
                            {
                                WriteLine(reader.Value);
                                break;
                            }
                        }
                    }
                }
            }
            catch (Exception ex)
            {
                WriteLine(ex.Message);
            }
        }
    }
}
```

```
        finally
        {
            if (reader != null)
                reader.Close();
        }
    }
}
```



**Рисунок 2.8.** Вывод информации  
об определенном элементе

При завершении работы с `XmlTextReader` важно его закрыть, чтобы он в свою очередь мог закрыть источник данных, поэтому для объекта `XmlTextReader` вызывается метод `Close()` в блоках `finally`.

## 3. XML-документация

Документирование исходного кода должно быть стандартной частью процесса разработки. При создании программных компонентов разработчики должны учитывать, что в дальнейшем эти компоненты будут использоваться другими разработчиками. Поэтому наличие документации к компоненту облегчает его использование и снижает затраты на поддержку.

### Что такое XML-документация?

Как Вы знаете из предыдущих уроков, для комментирования различных участков кода в языке C# существует два типа комментариев: однострочные (`//`) и многострочные (`/**/`). Однако существует еще один тип комментариев — XML комментарии (`///`), которые используются для добавления встроенных тегов документации XML. Теги XML могут быть размещены в исходных файлах для документирования кода и используются для создания внешней исходной документации, что отличает их от традиционных комментариев кода.

Теги документации XML используются для документирования классов и их элементов, таких как конструкторы, методы, свойства, перечисления, делегаты, события и т.д. Рассмотрим наиболее используемые теги:

- `summary` — предоставляет краткое описание типа, текст, написанный между тегами, отображается IntelliSense в Visual Studio;

- `param` — предоставляет описание связанного параметра, текст, написанный между тегами, отображается при написании клиентского кода;
- `paramref` — позволяет связать содержащееся слово с названием параметра с таким же именем в теге `<param>`;
- `returns` — предоставляет описание значения, возвращаемого методом;
- `remarks` — предоставляет более подробную информацию о типе или элементе;
- `para` — позволяет управлять форматом вывода документации, вставляя или перенося сегменты внутри абзацев;
- `c` — иллюстрирует короткий сегмент кода;
- `code` — иллюстрирует большой пример кода;
- `example` — предназначен для иллюстрации примеров кода, как правило, включает в себя теги `<c>` и `<code>`;
- `see` — позволяет создать ссылку на любой тип, который используется внутри класса, ссылки интегрируются непосредственно в справочную систему Visual Studio;
- `exception` — позволяет документировать типы исключений, которые будет генерировать данный элемент;
- `include` — связывает внешний файл документации с Вашим исходным кодом;
- `value` — предоставляет описание значения свойства.

Если Вы введете тройной слеш над необходимым элементом, то Visual Studio автоматически сформирует необходимые теги одним из, которых будет тег `<summary>`.

## Зачем использовать XML-документацию

Как уже говорилось выше, одной из причин использования XML-документации является стандартизация проектов, написанных на языке C#, что значительно облегчает их дальнейшее сопровождение.

Второй не менее важной причиной является автоматическое отображение информации о задокументированных элементах Вашего класса в IntelliSense Visual Studio, что осуществляется таким же образом, как и для стандартных элементов .NET Framework. Как Вы понимаете, это значительно облегчит работу с Вашим классом и сэкономит время, как Вам, так и другим разработчикам.

И последняя причина — возможность создания XML-файла, в котором будут находиться все теги XML-документации Вашего класса, и именно этот файл IntelliSense будет использовать для генерирования всплывающих подсказок при работе с вашим классом. Вы же можете использовать полученный файл для различных целей.

## Примеры использования

Для того чтобы продемонстрировать использование XML-документации, мы создадим в отдельном файле простой класс `Distance`, в котором разместим два метода, позволяющих осуществить перевод километров в мили и обратно. В этом классе мы создадим всю необходимую XML-документацию, описывающую класс, методы, параметры, исключения и возвращаемые значения.



```

using System;

namespace SimpleProject
{
    /// <summary>
    /// The <c>Distance</c> class provides methods
    /// for converting kilometers to and from miles.
    /// </summary>
    public class Distance
    {
        /// <summary>
        /// Converts miles to miles.
        /// </summary>
        /// <param name="kilometers">
        /// Used to indicate kilometers.
        /// A <see cref="double"/>
        /// type representing a value.
        /// </param>
        /// <exception cref="ArgumentException">
        /// If <paramref name="kilometers"/>
        /// is negative.
        /// </exception>
        /// <returns>Returns the distance in miles.
        /// </returns>
        public static double
            KilometersToMiles(double kilometers)
        {
            if (kilometers < 0)
            {
                throw new ArgumentException("The value
                    must be positive.");
            }
            return kilometers * 0.621371;
        }

        /// <summary>
        /// Converts miles to kilometers.
    }
}

```

```

    /// </summary>
    /// <param name="miles">
    /// Used to indicate miles.
    /// A <see cref="double"/> type representing a value.
    /// </param>
    /// <exception cref="ArgumentException">
    /// If <paramref name="miles"/> is negative.
    /// </exception>
    /// <returns>Returns the distance in
    /// kilometers.</returns>
    public static double MilesToKilometers(double miles)
    {
        if (miles < 0)
        {
            throw new ArgumentException("The value
                must be positive.");
        }
        return miles * 1.60934;
    }
}

```

Далее необходимо создать XML-файл, в котором будет содержаться вся информация о XML-документации класса [Distance](#), информация из этого файла будет использоваться IntelliSense. Чтобы создать этот файл необходимо открыть окно свойств проекта (сочетание клавиш Alt+Enter) и на вкладке Build указать два свойства Output path и XML documentation file (свойство необходимо активировать). Свойство Output path сообщает компилятору, где создавать файл XML-документации с возможностью выбора места (кнопка Browse...), а в свойстве XML documentation file указывается полное имя Вашего файла документации XML с учетом значения свойства Output path (Рисунок 3.1).

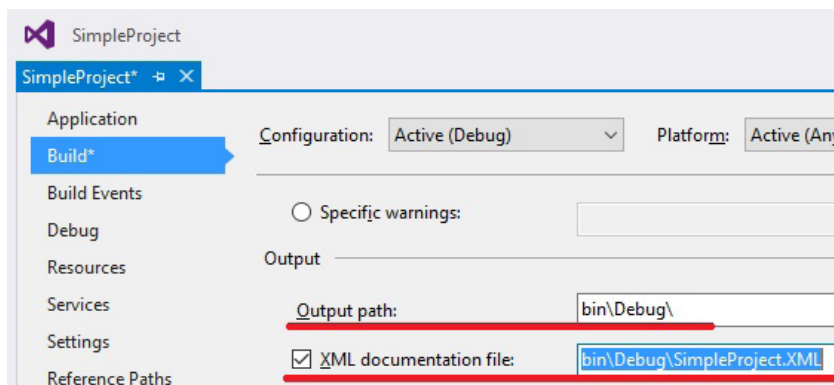


Рисунок 3.1. Установка свойств создание XML-файла

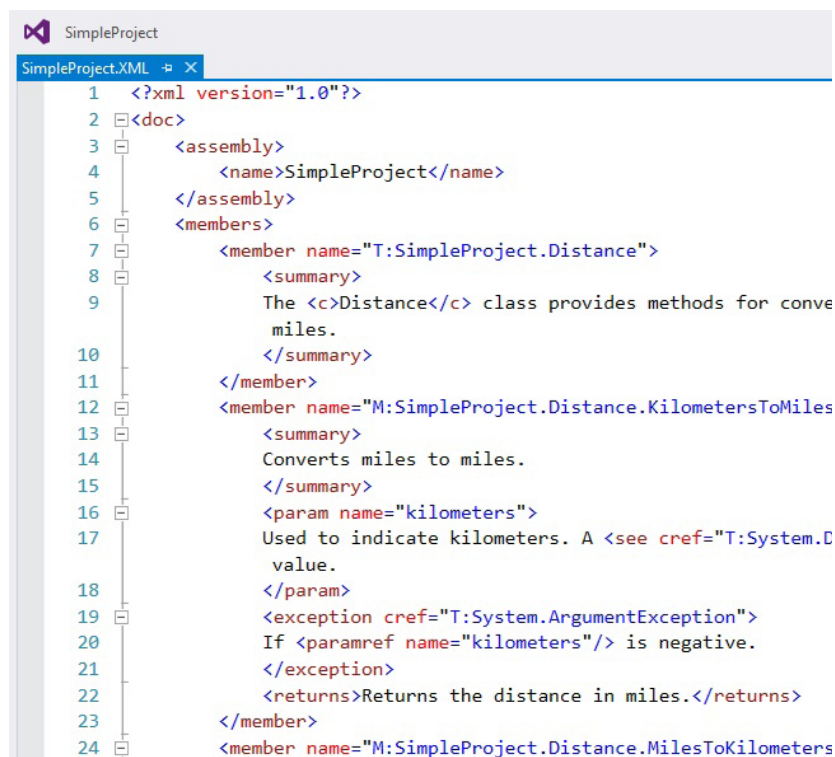
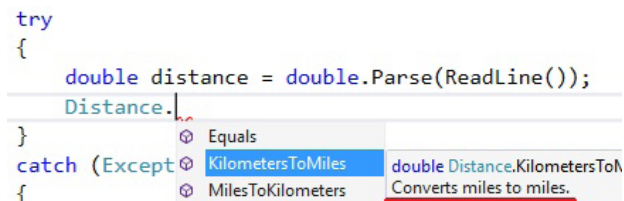


Рисунок 3.2. Содержимое XML-файла документации

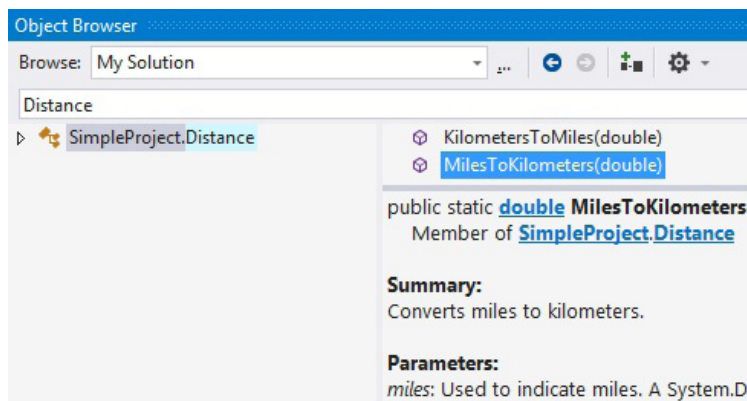
После того как Вы установите необходимые значения свойств, необходимо перекомпилировать Ваш проект, после чего по указанному Вами пути будет создан XML-файл (Рисунок 3.2).

После этого IntelliSense будет выводить информацию о ваших методах во всплывающих подсказках (Рисунок 3.3).



**Рисунок 3.3.** Информация о методе во всплывающей подсказке

Еще одним улучшением является возможность увидеть в окне ObjectBrowser всю документацию, связанную с Вашим классом (Рисунок 3.4).



**Рисунок 3.4.** Окно ObjectBrowser с информацией о классе Distance

Ну, и напоследок приведем пример использования, созданного нами класса.

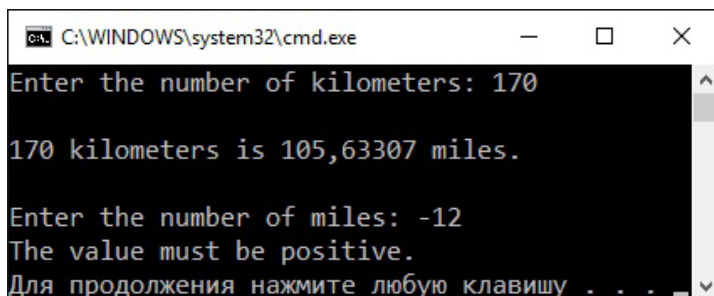
```
using System;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            try
            {
                Write("Enter the number of kilometers: ");
                double distance = double.Parse(ReadLine());
                WriteLine($"{distance} kilometers is
                           {Distance.KilometersToMiles(distance)}
                           miles.\n");

                Write("Enter the number of miles: ");
                distance = double.Parse(ReadLine());
                WriteLine($"{distance} miles is
                           {Distance.MilesToKilometers(distance)}
                           kilometers.");
            }

            catch (Exception ex)
            {
                WriteLine(ex.Message);
            }
        }
    }
}
```

Возможный вариант работы программы показан на рисунке 3.5.



```
C:\WINDOWS\system32\cmd.exe
Enter the number of kilometers: 170
170 kilometers is 105,63307 miles.
Enter the number of miles: -12
The value must be positive.
Для продолжения нажмите любую клавишу . . .
```

**Рисунок 3.5.** Пример использования класса Distance

# Домашнее задание

---

С помощью класса `XmlTextWriter` напишите приложение, сохраняющее в XML-файл информацию о заказах. Каждый заказ представляет собой несколько товаров. Информацию, характеризующую заказы и товары необходимо разработать самостоятельно.

Считайте информацию из XML-документа, полученного в первом задании с помощью классов `XmlDocument` и `XmlTextReader` и выведите полученную информацию на экран.



## Урок №11

# Работа с XML-файлами

© Юрий Задерей.

© Компьютерная Академия «Шаг»

[www.itstep.org](http://www.itstep.org).

Все права на охраняемые авторским правом фото-, аудио- и видеопроизведения, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объёме, оправданном поставленной задачей, в рамках учебного процесса и в учебных целях, в соответствии со ст. 1274 ч. 4 ГК РФ и ст. 21 и 23 Закона Украины «Про авторське право і суміжні права». Объём и способ цитируемых произведений соответствует принятым нормам, не наносит ущерба нормальному использованию объектов авторского права и не ущемляет законные интересы автора и правообладателей. Цитируемые фрагменты произведений на момент использования не могут быть заменены альтернативными, не охраняемыми авторским правом аналогами, и как таковые соответствуют критериям добросовестного использования и честного использования.

Все права защищены. Полное или частичное копирование материалов запрещено. Согласование использования произведений или их фрагментов производится с авторами и правообладателями. Согласованное использование материалов возможно только при указании источника.

Ответственность за несанкционированное копирование и коммерческое использование материалов определяется действующим законодательством Украины.