

Платформа Microsoft .NET и язык программирования C#



Урок №10

Взаимодействие с файловой системой. Сериализация

Содержание

1.	Модель потоков в C#. Пространство System.IO	4
2.	Класс Stream	5
3.	Анализ байтовых классов потоков	7
4.	Анализ символьных классов потоков.	9
5.	Анализ двоичных классов потоков.	10
6.	Использование класса FileStream для файловых операций.	11
7.	Использование класса StreamWriter для файловых операций.	16
8.	Использование класса StreamReader для файловых операций.	18
9.	Использование класса BinaryWriter для файловых операций.	20

10. Использование класса BinaryReader для файловых операций.	23
11. Использование классов Directory, DirectoryInfo, File и FileInfo для файловых операций.	25
12. Регулярные выражения	33
13. Понятие атрибутов	38
14. Что такое сериализация?	43
15. Отношения между объектами	44
16. Графы отношений объектов	45
17. Атрибуты для сериализации [Serializable] и [NonSerialized].	47
18. Форматы сериализации	49
Пространство System.Runtime.Serialization.Formatters	50
Двоичное форматирование. Класс BinaryFormatter	51
Soap форматирование. Класс SoapFormatter	54
Сериализация Xml. Класс XmlSerializer	57
Примеры использования сериализации	60
Создание пользовательского формата сериализации. Интерфейс ISerializable	66
19. Домашнее задание	70

1. Модель потоков в C#. Пространство System.IO

Практически все операции, связанные с вводом/выводом любой последовательности байтов (запись/чтение файлов, использование устройств ввода-вывода, потока шифрования и др.) в .NET Framework осуществляются с помощью потоков.

Поток — это абстракция, которая используется как для чтения, так и для записи информации. Хотя потоки связаны с различными физическими устройствами, их поведение во многом аналогично и не зависит от конкретного устройства, поэтому классы ввода-вывода могут использоваться с устройствами различных типов.

Основным пространством имен в .NET Framework, связанным с вводом-выводом информации, является `System.IO`, содержащее необходимый набор классов, структур, делегатов и перечислений, большинство из которых будет рассмотрено в данном уроке.

2. Класс Stream

Класс `Stream` из пространства имен `System.IO` является базовым классом для всех потоков, он обеспечивает универсальное представление различных типов ввода и вывода, изолируя программиста от отдельных сведений операционной системы и базовых устройств. Однако в зависимости от базового источника или хранилища данных потоки могут поддерживать только некоторые возможности, предоставляемые классом `Stream`.

Для определения возможностей потока осуществлять чтение и запись используются свойства `CanRead` и `CanWrite` соответственно. Методы `Read()` и `Write()` позволяют выполнять чтение и запись данных в различных форматах. Если поток поддерживает возможность позиционирования (значение свойства `CanSeek` равно `true`), то для установки позиции в конкретном потоке можно использовать метод `Seek()` либо свойство `Position`, а для изменения длины потока метод `SetLength()`. В противном случае для определения длины потока необходимо прочитать его до конца.

Некоторые реализации потоков выполняют локальную буферизацию основных данных для улучшения производительности. В таких потоках для удаления внутренних буферов и обеспечения принудительной записи всех данных в основной источник данных или хранилище объектов используется метод `Flush()`.

Вызов метода `Close()` класса `Stream` освобождает такие ресурсы операционной системы, как дескрипторы файлов, сетевые подключения или память, используемую для внутренней буферизации.

3. Анализ байтовых классов потоков

Классы `FileStream`, `MemoryStream` и `BufferedStream` являются наследниками класса `Stream`, которые предназначены для работы с различными потоками байтов.

Класс `FileStream` используется для выполнения различных операций с файлами (чтение, запись, открытие, закрытие и т.д.), обеспечивая доступ на уровне байтов. Он позволяет работать как с бинарными, так и с текстовыми файлами. Однако в случае взаимодействия с последними необходимо использовать специальные методы преобразования из байтов в строки, потому что класс `FileStream` позволяет работать с файлами только на уровне байтов. Поэтому имеет смысл в этом случае использовать специальные классы для работы с текстовыми файлами (`StreamReader`, `StreamWriter`).

Интенсивное использование данных, которые хранятся в файлах, может привести к снижению производительности вашего приложения из-за выполнения большого количества операций, связанных с открытием и закрытием файлов. В такой ситуации имеет смысл использовать класс `MemoryStream`, который позволяет сохранять временные данные в оперативной памяти, в связи с чем работа с этими данными осуществляется очень быстро. В этом виде потоков все данные сохраняются в виде массива байтов, размер которого невозможно изменить.

Класс `BufferedStream` обеспечивает механизм буферизации при осуществлении операций чтения и записи в различных потоках. Данный механизм предусматривает использование зарезервированной области памяти, которая используется для временного хранения данных — буфера. Механизм буферизации обеспечивает повышение производительности при чтении и записи данных и может использоваться для синхронизации передачи данных между устройствами, которые работают с различной скоростью.

Байтовые классы потоков позволяют получить информацию в виде массива байтов, и при необходимости записать эти данные в файл, иначе программистам придется бы самостоятельно преобразовывать байты в значения требуемого типа. Поэтому в пространстве `System.IO` находятся классы-оболочки, которые предназначены для автоматического преобразования байтового потока либо в бинарный, либо в символьный потоки.

4. Анализ символьных классов потоков

Базовыми классами для работы с символьными потоками являются абстрактные классы `TextReader` и `TextWriter`.

Класс `TextReader` является базовым классом для класса `StringReader`,читывающего данные из строки, и класса `StreamReader`, который используется для чтения символов из байтового потока в определенной кодировке. Кодировка задается при помощи класса `Encoding` из пространства имен `System.Text`, по умолчанию используется кодировка UTF-8.

Класс `TextWriter` — базовый класс для классов, осуществляющих запись символов: класс `StringWriter` позволяет осуществить запись символов в строку и класс `StreamWriter`, который записывает символы в поток байт в необходимой кодировке (по умолчанию UTF-8).

5. Анализ двоичных классов потоков

Для чтения и записи данных в двоичном формате используются классы-оболочки `BinaryReader` и `BinaryWriter`.

Класс `BinaryReader` предназначен для чтения двоичных данных из потока, при этом существует возможность указать требуемую кодировку, по умолчанию используется кодировка UTF-8. У этого класса существует ряд методов, которые позволяют считывать различные стандартные типы данных из текущего потока байт.

Класс `BinaryWriter` используется для записи в поток стандартных типов данных в двоичном формате, с возможностью указания кодировки (по умолчанию UTF-8). В классе существуют перегруженные методы для записи в поток любого стандартного типа данных.

6. Использование класса FileStream для файловых операций

Класс `FileStream` предоставляет методы `Read()` и `Write()`, которые обеспечивают чтение и запись потока байт в файл, соответственно. Однако прежде чем их использовать, необходимо создать экземпляр класса `FileStream` для чего можно воспользоваться одним из 15 перегруженных конструкторов. Наиболее универсальным является конструктор, который принимает четыре параметра: путь к файлу, режим создания, режим доступа и режим совместного использования.

Путь к файлу указывается либо в виде литерала, либо в виде переменной типа `string` и обязательно должен содержать имя файла с расширением, при этом может быть указан как абсолютный, так и относительный путь к файлу (папка `Debug` текущего проекта).

Режим создания файла задается при помощи перечисления `FileMode`, путем указания одного из перечисленных ниже значений:

- `FileMode.Append` — открывает файл, если он существует, и перемещает курсор в конец файла, если файл не существует — создает новый файл, можно использовать только вместе с `FileAccess.Write`;
- `FileMode.Create` — создает новый файл, если файл уже существует, то он будет переписан;

- `FileMode.CreateNew` — создает новый файл, если файл уже существует, то будет вызвано исключение `IOException`;
- `FileMode.Open` — открывает существующий файл, если файла не существует, то будет вызвано исключение `FileNotFoundException`;
- `FileMode.OpenOrCreate` — открывает существующий файл, если файла не существует, то будет создан новый файл;
- `FileMode.Truncate` — открывает существующий файл и усекает его размер до нуля.

Режим доступа к файлу задается с использованием значений перечисления `FileAccess` и определяет поведение потока:

- `FileAccess.Write` — данные можно только записать в файл;
- `FileAccess.Read` — данные можно только прочитать из файла;
- `FileAccess.ReadWrite` — данные можно записать в файл и прочитать из файла.

Режим совместного использования указывает на то, каким образом этот файл будет доступен другим объектам и задается при помощи значений перечисления `FileShare`:

- `FileShare.Delete` — разрешает удаление файла;
- `FileShare.Inheritable` — разрешает наследование дескриптора файла дочерними процессами;

- `FileShare.None` — отклоняет совместное использование текущего файла, любой запрос на открытие файла не выполняется до тех пор, пока файл не будет закрыт;
- `FileShare.Read` — разрешает последующее открытие файла для чтения, если этот флаг не задан, любой запрос на открытие файла для чтения не выполняется до тех пор, пока файл не будет закрыт;
- `FileShare.ReadWrite` — разрешает последующее открытие файла для чтения или записи, если этот флаг не задан, любой запрос на открытие файла для записи или чтения не выполняется до тех пор, пока файл не будет закрыт;
- `FileShare.Write` — разрешает последующее открытие файла для записи, если этот флаг не задан, любой запрос на открытие файла для записи не выполняется до тех пор, пока файл не будет.

Перечисления `FileAccess` и `FileShare` поддерживают побитовое соединение значений, то есть при помощи оператора побитовое ИЛИ (`|`) можно соединить несколько значений соответствующего перечисления.

Пример использования класса `FileStream` представлен на рисунке 6.1.

```
using System.IO;
using System.Text;
using static System.Console;

namespace SimpleProject
{
```

```
class Program
{
    static void WriteFile(string filePath)
    {
        using (FileStream fs = new FileStream(filePath,
            FileMode.Create, FileAccess.Write,
            FileShare.None))
        {
            // получаем данные для записи в файл
            WriteLine("Enter the data to write
                      to the file:");
            string writeText = ReadLine();
            // преобразуем строку в массив байт
            byte[] writeBytes = Encoding.Default.
                GetBytes(writeText);
            // записываем данные в файл
            fs.Write(writeBytes, 0, writeBytes.Length);
            WriteLine("Information recorded!");
        }
    }
    static string ReadFile(string filePath)
    {
        using (FileStream fs = new FileStream(filePath,
            FileMode.Open, FileAccess.Read,
            FileShare.Read))
        {
            byte[] readBytes = new byte[(int)fs.Length];
            // считываем данные из файла
            fs.Read(readBytes, 0, readBytes.Length);
            // преобразуем байты в строку
            return Encoding.Default.
                GetString(readBytes);
        }
    }
    static void Main(string[] args)
    {
        string filePath = "test.bin";
```

6. Использование класса FileStream для файловых операций

```
        WriteFile(filePath);

        // выводим результат на консоль
        WriteLine($"\\nData read from the file:
{ReadFile(filePath)}");
    }
}

}
```

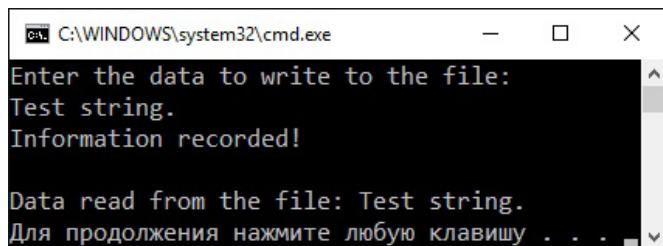


Рисунок 6.1 Пример использования класса FileStream

Для обеспечения стабильной работы вашей программы при работе с потоками всегда следует помнить о необходимости вызова метода `Close()` для освобождения ресурсов операционной системы, выделенных этому потоку, после окончания работы с ним. Однако, как вы помните из седьмого урока, для классов реализующих интерфейс `IDisposable` существует возможность применения специального синтаксиса с использованием ключевого слова `using`, который гарантирует автоматическую очистку ресурсов. Данный подход вы видели в предыдущем примере при использовании класса `FileStream`.

7. Использование класса StreamWriter для файловых операций

Как уже говорилось ранее, класс `StreamWriter` позволяет осуществлять запись в поток символьных данных, то есть по факту обеспечивает запись информации в текстовые файлы. Методы `Write()` и `WriteLine()`, позволяющие записывать данные в текстовый файл, вы уже встречали при работе с классом `Console`. В этом нет ничего удивительного, ведь свойство `Out` класса `Console` возвращает тип `TextWriter` — стандартный выходной поток (консоль). Приведем пример записи текста в файл (Рисунок 7.1).

```
using System.IO;
using System.Text;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string filePath = "test.txt";
            using (FileStream fs = new FileStream(filePath,
                FileMode.Create))
            {
                using (StreamWriter sw = new StreamWriter(fs,
                    Encoding.Unicode))
                {

```

```
// получаем данные для записи в файл
WriteLine("Enter the data to write
          to the file:");
string writeText = ReadLine();
// записываем данные в файл
sw.WriteLine(writeText);
foreach (var item in writeText)
{
    sw.Write($"{item} ");
}
WriteLine("\nData recorded!");
}
}
```

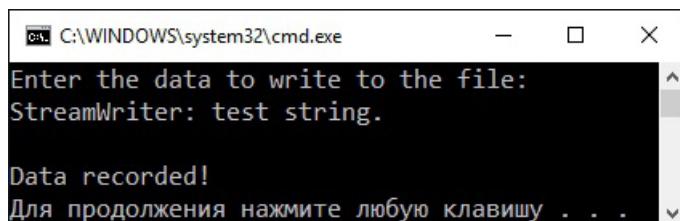


Рисунок 7.1. Запись данных в текстовый файл

Откроем полученный файл в Блокноте, чтобы проверить правильность записи данных (Рисунок 7.2).

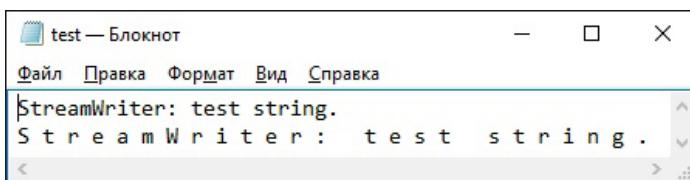


Рисунок 7.2. Содержимое записанного текстового файла

8. Использование класса StreamReader для файловых операций

Класс `StreamReader` используется для чтения информации из текстовых файлов при помощи соответствующих методов:

- `Read()` — читает следующий символ, либо массив символов;
- `ReadBlock()` — читает массив символов;
- `ReadLine()` — читает строку от текущей позиции до символа перехода на новую строчку.
- `ReadToEnd()` — считывает все символы из потока, начиная от текущей позиции до конца.

Опять-таки некоторые из методов вам уже знакомы по классу `Console`, потому что свойство `In` класса `Console` возвращает тип `TextReader` — стандартный входной поток (клавиатура).

Также у класса `StreamReader` существует три свойства:

- `BaseStream` — возвращает основной поток;
- `CurrentEncoding` — позволяет определить текущую кодировку потока;
- `EndOfStream` — показывает, не достигли ли мы конца потока.

Приведем пример чтения текста из файла (Рисунок 8.1).

```
using System.IO;
using System.Text;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string filePath = "test.txt";
            using (FileStream fs = new FileStream(filePath,
                FileMode.Open))
            {
                using (StreamReader sr =
                    new StreamReader(fs,
                        Encoding.Unicode))
                {
                    // выводим результат на консоль
                    WriteLine($"Data read from the file:\n");
                    WriteLine(sr.ReadToEnd()); // получаем
                    // все данные из файла
                }
            }
        }
    }
}
```

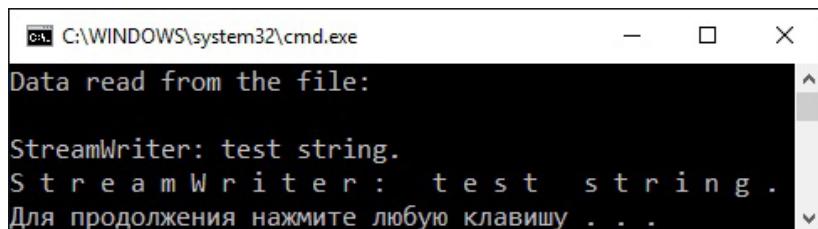


Рисунок 8.1. Чтение данных из текстового файла

9. Использование класса BinaryWriter для файловых операций

Запись информации в файлы в бинарном виде осуществляется при помощи класса `BinaryWriter`, для чего существует ряд перегруженных методов `Write()`, принимающих различные типы данных. Пример записи бинарной информации в файл приведен ниже (Рисунок 9.1)

```
using System.IO;
using System.Text;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string filePath = "test.dat";
            using (FileStream fs = new FileStream(filePath,
                FileMode.Create))
            {
                using (BinaryWriter bw =
                    new BinaryWriter(fs,
                        Encoding.Unicode))
                {
                    WriteLine("Enter the data to write
                            to the file:");
                    string writeText = ReadLine();
                    double pi = 3.1415926;
```

```
        int number = 1256;

        // запись данных в файл
        bw.Write(writeText);
        bw.Write(pi);
        bw.Write(number);

        WriteLine("\nData recorded!");
    }

}

}

}
```

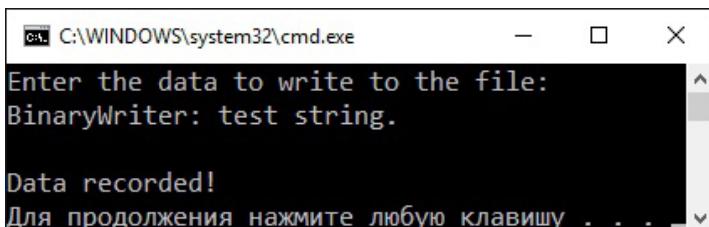


Рисунок 9.1. Запись в файл бинарных данных

Если открыть полученный файл в Блокноте, то мы увидим, что текстовую информацию на английском языке мы можем прочитать, а остальные данные нечитабельны, так как сохранены в бинарном виде (Рисунок 9.2).

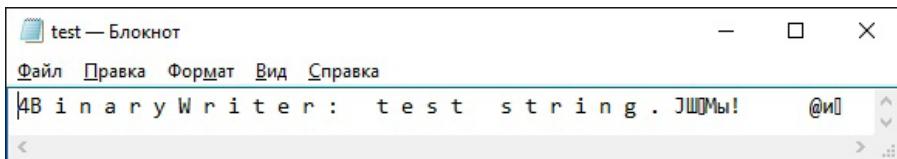


Рисунок 9.2. Содержимое записанного бинарного файла

Текст, использующий латинские буквы, не кодируется, какой бы тип кодировки при записи вы не указали. Это связано с тем, что основой для всех текстовых кодировок является кодировка ASCII (*American Standard Code for Information Interchange*), в которой описываются 128 символов — знаки препинания, арабские цифры и латинские буквы. Если вы хотите получить полностью нечитабельный бинарный файл, тогда введите текст на каком-нибудь другом языке, например русском (предлагаем вам проверить это самостоятельно).

10. Использование класса BinaryReader для файловых операций

Класс `BinaryReader` предназначен для считывания информации, которая хранится в файлах в двоичном формате. Для чего используется ряд методов, которые позволяют считать из файла различные типы данных, все они начинаются на слово `Read`, после которого следует название конкретного типа данных. Следует учитывать, что считывать данные вы должны в том же порядке, в котором они были записаны, продемонстрируем это на примере (Рисунок 10.1).

```
using System.IO;
using System.Text;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string filePath = "test.dat";
            using (FileStream fs = new FileStream(filePath,
                FileMode.Open))
            {
                using (BinaryReader br =
                    new BinaryReader(fs,
                        Encoding.Unicode))
```

```
{  
    // получаем данные из файла  
    WriteLine($"Data read from the file:\n");  
    WriteLine(br.ReadString());  
    WriteLine(br.ReadDouble());  
    WriteLine(br.ReadInt32());  
}  
}  
}  
}  
}
```

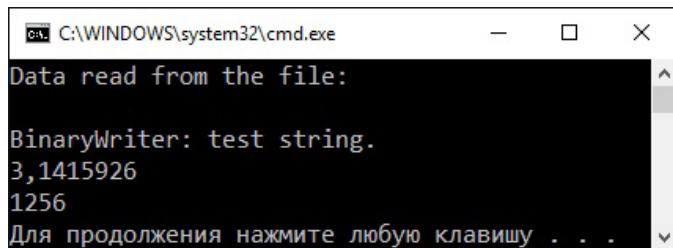


Рисунок 10.1. Чтение данных из бинарного файла

11. Использование классов Directory, DirectoryInfo, File и FileInfo для файловых операций

В пространстве имен System.IO находятся классы, которые позволяют осуществлять различные операции с каталогами — `Directory` и `DirectoryInfo`, а также с файлами `File` и `FileInfo`. Эти классы имеют подобную функциональность, но отличаются способом использования. Для того чтобы вызвать методы классов `DirectoryInfo` и `FileInfo` необходимо создать экземпляры этих классов, а классы `Directory` и `File` являются статическими и предоставляют соответственно статические методы. Поэтому если вам нужен объект для многократного использования при работе с папками или файлами имеет смысл использовать классы `DirectoryInfo` или `FileInfo`, соответственно, а для выполнения одного конкретного действия (создание, открытие, копирование и т.д.) наиболее эффективно будет использовать методы классов `Directory` или `File`. Для начала приведем пример получения информации о каталоге с использованием класса `DirectoryInfo` (Рисунок 11.1).

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

namespace SimpleProject
{
```

```
class Program
{
    static void Main(string[] args)
    {
        // текущий каталог
        DirectoryInfo dir = new DirectoryInfo(".");
        WriteLine($"Full path to the
                  directory:\n{dir.FullName}");
        WriteLine($"Time of creation:
                  {dir.CreationTime}");

        WriteLine("\n\tAll directory files:");
        FileInfo[] files = dir.GetFiles(); // все файлы
                                         // в каталоге
        foreach (FileInfo f in files)
        {
            WriteLine(f.Name);
        }
        WriteLine();
    }
}
```

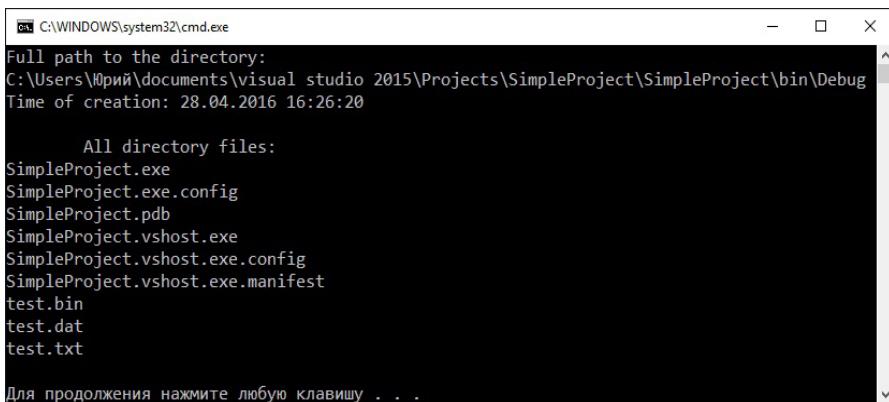


Рисунок 11.1. Получение информации о текущем каталоге

Следующий пример демонстрирует возможность создания каталога и подкаталога при помощи методов класса `DirectoryInfo` и использование методов класса `FileInfo` для создания файла и записи в него информации (Рисунок 11.2).

```
using System.IO;
using System.Text;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void WriteFile(FileInfo f)
        {
            using (FileStream fs = f.Open(FileMode.Create,
                FileAccess.Write, FileMode.None))
            {
                WriteLine("\nEnter the data to write to the file:");
                string writeText = ReadLine();
                byte[] writeBytes =
                    Encoding.Default.GetBytes(writeText);
                fs.Write(writeBytes, 0, writeBytes.Length);
                WriteLine("\nData recorded!\n");
            }
        }

        static string ReadFile(FileInfo f)
        {
            using (FileStream fs = f.OpenRead())
            {
                byte[] readBytes = new byte[(int)fs.Length];
                fs.Read(readBytes, 0, readBytes.Length);
                return Encoding.Default.GetString(readBytes);
            }
        }
    }
}
```

```
static void Main(string[] args)
{
    DirectoryInfo dir =
        new DirectoryInfo(@"D:\Test");

    if (!dir.Exists) // если каталог не существует
    {
        dir.Create(); // создаем каталог
    }
    WriteLine($"Last access time to the directory:
{dir.LastAccessTime}");

    // создаем подкаталог
    DirectoryInfo dir1 =
        dir.CreateSubdirectory("Subdir1");
    WriteLine($"Full path to
the directory:\n{dir1.FullName}");

    FileInfo fInfo =
        new FileInfo(dir1 + @"\Test.bin");
    WriteFile(fInfo);
    WriteLine(ReadFile(fInfo));

    WriteLine($"\\n\\tOnly files with
the extension '.bin':");
    FileInfo[] files = dir1.GetFiles("*.bin");

    foreach (FileInfo f in files)
    {
        WriteLine(f.Name);
    }
    WriteLine();

}

}
```

The screenshot shows a command prompt window titled 'cmd' with the path 'C:\WINDOWS\system32\cmd.exe'. The output of the program is as follows:

```
Last access time to the directory: 29.03.2017 0:00:00
Full path to the directory:
D:\Test\Subdir1

Enter the data to write to the file:
FileInfo: test string.

Data recorded!

FileInfo: test string.

Only files with the extension '.bin':
Test.bin

Для продолжения нажмите любую клавишу . . .
```

Рисунок 11.2. Пример использования
классов DirectoryInfo и FileInfo

Следующий пример подобен предыдущему, только в этот раз мы создадим текстовый файл, и для этого будем использовать класс `File` (Рисунок 11.3).

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

namespace SimpleProject
{
    class Program
    {
        static void WriteFile(string path)
        {
            using (StreamWriter sw = File.CreateText(path))
            {
                WriteLine("Enter the data to write to the file:");
                string writeText = ReadLine();
```

```
        sw.WriteLine(writeText);
        foreach (var item in writeText)
        {
            sw.Write(${item} );
        }
        WriteLine("\nData recorded!");
    }

static string ReadFile(string path)
{
    using (StreamReader sr = File.OpenText(path))
    {
        return sr.ReadToEnd();
    }
}

static void Main(string[] args)
{
    string path = @"D:/Test/Subdir1/Test.txt";

    try
    {
        WriteFile(path);
        WriteLine($"\\nData read from the file:\\n");
        WriteLine(ReadFile(path));
    }

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

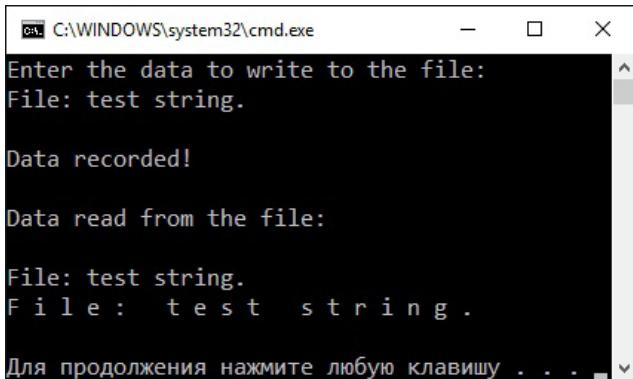


Рисунок 11.3. Пример использования класса File

В последнем примере этого раздела демонстрируется использование класса `Directory` для работы с каталогами (Рисунок 11.4).

```

using System.IO;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string path = @"D:\Test";
            if(Directory.Exists(path))
            {
                WriteLine($"Date and time of catalog creation:
                    {Directory.GetCreationTime(path)}");
                WriteLine($"\\nSubdirectories
                    in the specified
                    directory:");
                foreach (string item in
                    Directory.GetDirectories(path))
                {

```

```
        WriteLine($"\\t{item}");  
    }  
    WriteLine($"\\nLogical devices of  
            this computer:");  
    foreach (string item in  
            Directory.GetLogicalDrives())  
    {  
        WriteLine($"\\t{item}");  
    }  
    // удаляем каталог, все подкаталоги и файлы  
    Directory.Delete(path, true);  
}  
if (!Directory.Exists(path))  
{  
    WriteLine($"\\nSpecified directory  
          does not exist.\\n");  
}  
}  
}  
}
```

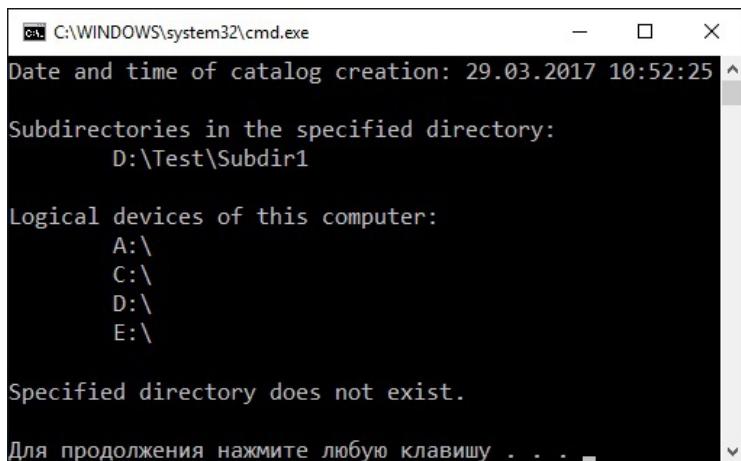


Рисунок 11.4. Пример использования класса Directory

12. Регулярные выражения

Во многих языках программирования для осуществления поиска в тексте определенной информации можно использовать регулярные выражения, смысл которых заключается в нахождении строк, удовлетворяющих заданному шаблону.

Платформа .NET Framework не является исключением, в ней для обеспечения работы с регулярными выражениями используется ряд классов из пространства имен `System.Text.RegularExpressions`, подробную информацию о них вы можете получить, обратившись в MSDN.

Основным классом, который всегда используется при работе с регулярными выражениями, является класс `Regex`. Обычно, при создании экземпляров этого класса в конструктор передается строка, представляющая собой регулярное выражение, с помощью которого и будет осуществляться поиск в тексте.

При написании регулярных выражений используются различные специальные символы, наличие каждого из них имеет определенное значение, а все они вместе определяют синтаксис регулярных выражений. В таблице 12.1 специальные символы представлены частично, за более подробной информацией рекомендуем обратиться в MSDN.

Таблица 12.1. Символы регулярных выражений (частично)

Символ	Значение
.	Любой одиночный символ, кроме символа новой строки (\n)
\w	Соответствует любому алфавитно-цифровому символу
\W	Соответствует любому не алфавитно-цифровому символу
\s	Соответствует любому пробельному символу
\S	Соответствует любому символу, не являющемуся пробелом
\d	Соответствует любой десятичной цифре
\D	Соответствует любому символу, не являющемуся десятичной цифрой
[символы]	Соответствует любому символу из множества, заданного в скобках
[^символы]	Соответствует любому символу, за исключением символов, заданных в скобках
*	Предшествующий шаблон повторяется 0 или более раз
+	Предшествующий шаблон повторяется 1 или более раз
?	Предшествующий шаблон повторяется 0 или 1 раз
{n, m}	Предшествующий шаблон повторяется не менее n и не более m раз
{n, }	Предшествующий шаблон повторяется n и более раз
{n}	Предшествующий шаблон повторяется ровно n раз
^	Шаблон должен находиться в начале текста
\$	Шаблон должен находиться в конце текста

Как вы могли заметить, при написании специальных символов регулярного выражения существует необходимость использовать символ обратная косая черта (\), который является символом экранирования. Поэтому для корректной записи строки необходимо либо экранировать

сам символ, либо ставить символ '@' перед всей строкой, создавая дословную строку, обычно используется второй подход.

Также существует возможность передать в конструктор несколько значений перечисления `RegexOptions`, соединенных при помощи оператора побитовое ИЛИ (|) :

- `RegexOptions.Compiled` — указывает, что регулярное выражение будет скомпилировано в сборку, что приводит к более быстрой работе, однако увеличивает время запуска;
- `RegexOptions.CultureInvariant` — указывает на игнорирование региональных языковых различий;
- `RegexOptions.ESCMAScript` — включает ECMAScript-совместимое поведение для регулярного выражения. Это значение может быть использовано только вместе со значениями `IgnoreCase`, `Multiline` и `Compiled`. Использование этого значения вместе с любыми другими параметрами приводит к исключению;
- `RegexOptions.ExplicitCapture` — указывает, что единственны допустимые записи являются явно поименованными или пронумерованными группами в форме (?<name>...);
- `RegexOptions.IgnoreCase` — указывает, что регулярное выражение не будет учитывать регистр;
- `RegexOptions.IgnorePatternWhitespace` — устраняет из шаблона неизбежные пробелы и включает комментарии, помеченные "#";
- `RegexOptions.Multiline` — многострочный режим. Изменяет значение символов "^" и "\$" так, что они

совпадают, соответственно, в начале и конце любой строки, а не только в начале и конце целой строки;

- `RegexOptions.None` — указывает на отсутствие дополнительных параметров;
- `RegexOptions.RightToLeft` — указывает на направление поиска — справа налево;
- `RegexOptions.Singleline` — указывает на применение односторочного режима.

Теперь приведем несколько примеров использования регулярных выражений (Рисунок 12.1).

```
using System.Text.RegularExpressions;
using static System.Console;

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string emailPattern = @"^([a-zA-Z0-9_-]+\.)*[a-zA-Z0-9_-]+@[a-zA-Z0-9_-]+\.(.[a-zA-Z0-9_-]+)*\.[a-zA-Z]{2,6}$";
            Write("Enter e-mail: ");
            string email = ReadLine();
            Regex regex = new Regex(emailPattern);
            WriteLine(regex.IsMatch(email) ? "E-mail confirmed." : "Incorrect e-mail!");
            string phonePattern = @"^\+\d{2}\((\d{3})\)\d{3}-\d{2}-\d{2}$";
            Write("Enter phone: ");
        }
    }
}
```

```
string phone = ReadLine();
regex = new Regex(phonePattern);
WriteLine(regex.IsMatch(email) ? "Data received.":
           "Data is not correct!");

WriteLine("\nReplacement of words matching
          a pattern.");
string text = "I like Java. Java forever.";
string textPattern = "Java";

WriteLine(text);
WriteLine(Regex.Replace(text, textPattern, "C#"));
}
}
}
```

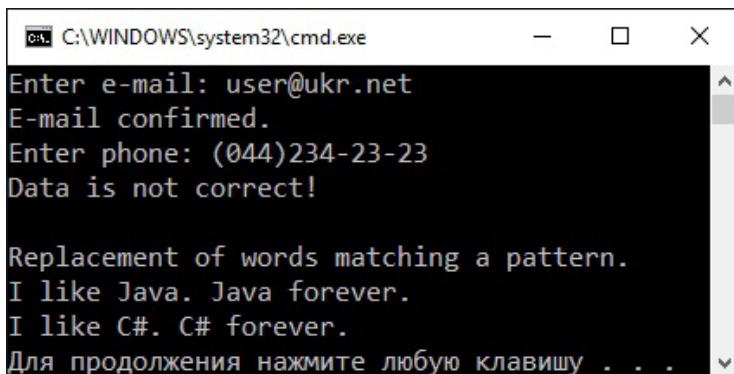


Рисунок 12.1. Пример использования регулярных выражений

13. Понятие атрибутов

Как вам известно, при компиляции написанного вами кода формируется исполняемый модуль (сборка), состоящий из: заголовка CLR, метаданных и CIL-кода. Так вот атрибуты в .NET Framework это специальные классы, при помощи которых можно добавить в сборку дополнительную информацию (метаданные). По умолчанию в любом приложении существуют атрибуты, описывающие текущую сборку, они находятся в файле AssemblyInfo.cs папки Properties. При использовании атрибутов в программном коде, метаданные остаются не используемыми до тех пор, пока другой фрагмент программного кода не будет явно использовать их для отображения данной информации.

Атрибуты указываются в квадратных скобках перед элементом приложения, дополнительную информацию о котором необходимо добавить в сборку. По договоренности все атрибуты оканчиваются на слово Attribute, например `ObsoleteAttribute`. Однако при написании атрибута допускается не указывать полное имя атрибута, приведем несколько примеров:

- [`Obsolete`] — указывает на то, что данный элемент является устаревшим, при его использовании в окне Error List появится соответствующее предупреждение;
- [`WebMethod`] — применяется только к методам и позволяет вызывать эти методы удаленным веб-клиентам;

- `[ServiceContract]` — применяется для интерфейсов или классов, которые определяют контракт службы в приложении WCF.

Атрибуты применяются к любому элементу приложения (сборка, класс, метод, делегат и т.д.) и у каждого из них есть свое предназначение. В .NET Framework существует большое количество стандартных атрибутов, но программист может создать атрибут для собственных нужд, так называемый пользовательский атрибут, и все они являются наследниками базового класса `System.Attribute`.

При создании пользовательского атрибута необходимо использовать атрибут `[AttributeUsage]`, конструктор которого может принимать до трех параметров.

Первым параметром всегда указывается значение перечисления `AttributeTargets`, при помощи которого указываются элементы приложения, к которым допустимо применять атрибут, допускается соединение нескольких значений при помощи оператора побитовое ИЛИ (`|`).

Второй и третий параметры имеют тип `bool` и являются не обязательными, и могут указываться в любой последовательности. Если явно задать параметру `AllowMultiple` значение `true`, то атрибут можно будет применять многократно к одному элементу, значение по умолчанию `false`. Значение следующего параметра `Inherited`, по умолчанию позволяет наследовать атрибут классами наследниками (значение `true`), явное присвоение значения `false` запрещает наследование.

В качестве примера рассмотрим создание и использование пользовательского атрибута, в котором будет храниться информация о разработчике и времени создания элемента приложения. Данный атрибут может применяться только к классам и методам что, и продемонстрировано на примере класса `Employee`. В методе `Main()` мы просто выводим информацию обо всех атрибутах этого класса (Рисунок 13.1).

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

namespace SimpleProject
{
    [AttributeUsage(AttributeTargets.Method |
                    AttributeTargets.Class)]
    public class CoderAttribute : Attribute
    {
        string _name = "Yuriy";
        DateTime _date = DateTime.Now;

        public CoderAttribute() { }

        public CoderAttribute(string name, string date)
        {
            try
            {
                _name = name;
                _date = Convert.ToDateTime(date);
            }
            catch (Exception ex)
            {
                WriteLine(ex.Message);
            }
        }
    }
}
```

```
public override string ToString()
{
    return $"Coder: {_name}, Date: {_date}";
}

[Coder]
class Employee
{
    public int Id { get; set; }
    public string Name { get; set; }
    public double Salary { get; set; }

    [Coder("John", "2017-3-29")]
    public void IncreaseWages(double wage)
    {
        Salary += wage;
    }
}
class Program
{
    static void Main(string[] args)
    {
        WriteLine("\tAttributes of class Employee:");

        foreach (var attr in typeof(Employee).
            GetCustomAttributes())
        {
            WriteLine(attr);
        }

        WriteLine("\n\tAttributes of members
of class Employee:");

        foreach (MemberInfo info in
            typeof(Employee).GetMembers())
        {
    }
```

Урок №10

```
        foreach (var attr in info.  
            GetCustomAttributes(true))  
        {  
            WriteLine(attr);  
        }  
    }  
}
```

```
C:\WINDOWS\system32\cmd.exe  
Attributes of class Employee:  
Coder: Yuriy, Date: 30.03.2017 15:36:01  
  
Attributes of members of class Employee:  
System.Runtime.CompilerServices.CompilerGeneratedAttribute  
System.Runtime.CompilerServices.CompilerGeneratedAttribute  
System.Runtime.CompilerServices.CompilerGeneratedAttribute  
System.Runtime.CompilerServices.CompilerGeneratedAttribute  
System.Runtime.CompilerServices.CompilerGeneratedAttribute  
System.Runtime.CompilerServices.CompilerGeneratedAttribute  
System.Runtime.CompilerServices.CompilerGeneratedAttribute  
Coder: John, Date: 29.03.2017 0:00:00  
__DynamicallyInvokableAttribute  
__DynamicallyInvokableAttribute  
__DynamicallyInvokableAttribute  
System.Security.SecureCriticalAttribute  
__DynamicallyInvokableAttribute  
Для продолжения нажмите любую клавишу . . .
```

Рисунок 13.1. Использование пользовательского атрибута

14. Что такое сериализация?

Сериализация это процесс сохранения состояния объекта в любом потоке, с возможностью его последующего восстановления, при этом сохраненная последовательность байт содержит всю необходимую информацию для восстановления этого объекта. Использование этой технологии намного облегчает процесс сохранения и извлечения больших объемов данных приложения в самых разных форматах, по сравнению с аналогичными действиями с использованием стандартных классов пространства System.IO.

15. Отношения между объектами

Хотя сохранять объекты с помощью средств сериализации очень просто, следует понимать, что процессы, происходящие при этом в фоновом режиме, являются достаточно сложными. Например, когда объект сохраняется в потоке, все соответствующие данные (базовые классы, вложенные объекты и т.п.) тоже автоматически сохраняются. Таким образом, при попытке выполнить сериализацию производного класса в этом процессе будет задействована вся цепочка наследования.

Множество взаимосвязанных объектов при сериализации представляется в виде графа объектов. Службы сериализации позволяют сохранить полученный граф объектов в самых разных форматах: двоичном формате, формате SOAP, формате XML или в формате JSON.

16. Графы отношений объектов

Для того чтобы гарантировать корректное сохранение данных объекта при его сериализации, среда CLR учитывает все связанные с ним объекты. Это множество связанных объектов и называется графом объектов, который обеспечивает простой способ учета взаимоотношений между объектами. Каждому объекту в графе объектов присваивается произвольное, но уникальное числовое значение, не имеющее смысла за пределами графа, как только это произошло, график объектов записывает все наборы зависимостей каждого объекта, причем все это происходит автоматически в фоновом режиме.

Для примера предположим, что мы создали набор классов, моделирующих типы автомобилей. У нас есть базовый класс, названный Auto (автомобиль), который имеет поле типа радио (Radio). Другой класс, SportAuto (спортивный автомобиль), расширяет базовый тип Auto. На рисунке 16.1 показан возможный график объектов, моделирующий указанные взаимосвязи.

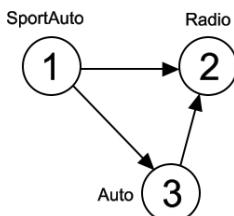


Рисунок 16.1. Пример графа объектов

Взаимосвязи, указанные в диаграмме, представляются средой CLR в виде формулы, которая выглядит примерно так:

```
[Auto 3, ref 2], [Radio 2],  
[SportAuto 1, ref 3, ref 2]
```

Из этой формулы видно, что объект 3 (Auto) имеет зависимость в отношении объекта 2 (Radio), объект 2 (Radio) является объектом, которому никто не требуется и наконец, объект 1 (SportAuto) имеет зависимость в отношении как объекта 3, так и объекта 2.

Когда будет выполняться сериализация или десериализация экземпляра SportAuto, объектный граф даст гарантию того, что типы Radio и Auto тоже будут участвовать в этих процессах.

Объектный график можно сохранить в любом производном от `System.IO.Stream` типе, главное, чтобы последовательность данных корректно представляла состояния объектов соответствующего графа.

17. Атрибуты для сериализации [Serializable] и [NonSerialized]

Для того чтобы ваш класс или структура могли участвовать в процессе сериализации, необходимо перед их объявлением указать атрибут `[Serializable]`. Если в вашем классе находятся поля, в сериализации которых отсутствует необходимость (случайные значения, фиксированные или кратковременные данные), то их следует пометить атрибутом `[NonSerialized]`. Эти поля не будут сохраняться при сериализации, что приведет к сокращению размера хранимых данных (графа объектов).

```
[Serializable]
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
    int _identNumber;

    [NonSerialized]
    const string Planet = "Earth";
}
```

Класс `Person` помечен атрибутом `[Serializable]`, поэтому все его элементы будут сериализованы, за исключением константного поля `Planet` (все люди живут на планете Земля), так как оно помечено атрибутом `[NonSerialized]`.

Атрибут `[Serializable]` не наследуется, поэтому если ваш класс наследуется от класса с атрибутом `[Serializable]`, то и у вашего класса тоже должен быть установлен атрибут `[Serializable]`, иначе он не сохранится в графе объектов при сериализации. При попытке сериализовать объект, который не поддерживает сериализацию, в среде выполнения будет сгенерировано исключение `SerializationException`.

18. Форматы сериализации

Платформа .NET Framework предоставляет возможность осуществлять сериализацию в четырех различных форматах:

- бинарный формат;
- формат XML;
- формат SOAP;
- формат JSON (будет рассмотрен в последующих курсах).

Между перечисленными выше форматами сериализации существуют определенные отличия.

При использовании бинарного форматирования будут сохраняться не только поля данных объектов из объектного графа, но и абсолютное имя каждого типа, а также полное имя определяющего тип компоновочного блока. Данный вид форматирования очень удобен когда необходимо использовать значение объектов в .NET приложениях, при этом форматировании сериализируются как открытые, так и закрытые поля типа.

Форматы SOAP и XML не являются форматами сериализации, которые сохраняют тип точно — они не записывают абсолютные имена типов и компоновочных блоков. Эти форматы сериализации предназначены для сохранения состояния объектов так, чтобы они могли использоваться в любой операционной системе, на любой платформе приложений (.NET, Java, QT и т.д.) или в любом языке программирования. Поэтому нет необходимости

поддерживать абсолютную точность для типа, поскольку нет гарантии, что все возможные получатели смогут понять типы данных, специфичные для .NET.

Пространство System.Runtime.Serialization.Formatters

Пространство `System.Runtime.Serialization.Formatters` является вложенным пространством имен и предоставляет интерфейсы и классы, которые используют различные классы форматеры. В частности содержит классы, предоставляющие информацию в случае возникновения ошибок при SOAP сериализации:

- `SoapFault` — предоставляет сведения об ошибках в SOAP сообщении;
- `ServerFault` — содержит сведения об ошибках со стороны сервера.

Это пространство имен является базовым для пространств имен, необходимых при бинарной и SOAP сериализациях (о них в следующих разделах).

Базовым для этого пространства имен является пространство `System.Runtime.Serialization`, которое в свою очередь содержит классы, структуры, интерфейсы и перечисления, используемые для сериализации и десериализации. Например, классы `OnSerializing`, `OnSerialized`, `OnDeserializing`, `OnDeserialized` при помощи которых можно управлять процессами сериализации и десериализации. В этом же пространстве имен находятся класс `SerializationException` — исключительная ситуация, которая генерируется в результате возникновения ошибок при сериализации и десериализации и интерфейс

[ISerializable](#), реализация которого классом позволяет ему управлять своими процессами сериализации и десериализации. Более подробную информацию об остальных членах этого пространства имен вы можете получить, обратившись в MSDN.

Двоичное форматирование. Класс [BinaryFormatter](#)

Двоичное форматирование осуществляется при помощи объектов класса [BinaryFormatter](#), который является единственным классом пространства имен `System.Runtime.Serialization.Formatters.Binary`. Этот класс реализует интерфейс [IFormatter](#) из пространства имен `System.Runtime.Serialization`, в котором определены два метода `Serialize()` и `Deserialize()`, выполняющие соответственно сериализацию и десериализацию графа объектов в двоичный формат. Пример бинарного форматирования представлен ниже (Рисунок 18.1).

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
using static System.Console;

namespace SimpleProject
{
    [Serializable]
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
        int _identNumber;

        [NonSerialized]
        const string Planet = "Earth";
    }
}
```

```
public Person(int number)
{
    _identNumber = number;
}

public override string ToString()
{
    return $"Name: {Name}, Age: {Age},
Identification number:
{_identNumber}, Planet: {Planet}.";
```

}

```
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person(346875) { Name =
            "Jack", Age = 34 };
        BinaryFormatter binFormat =
            new BinaryFormatter();
        try
        {
            using (Stream fStream =
                File.Create("test.bin"))
            {
                binFormat.Serialize(fStream, person);
            }
            WriteLine("BinarySerialize OK!\n");
        }

        Person p = null;
        using (Stream fStream =
            File.OpenRead("test.bin"))
        {
            p = (Person)binFormat.
                Deserialize(fStream);
        }
    }
}
```

```
        WriteLine(p);
    }
    catch (Exception ex)
    {
        WriteLine(ex);
    }
}
```

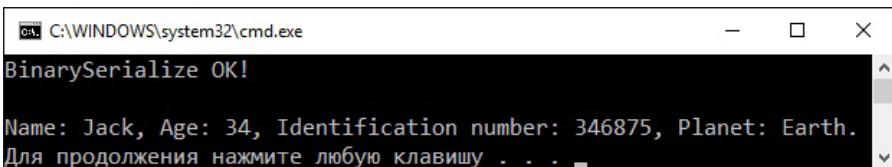


Рисунок 18.1. Бинарное форматирование объекта

Как вы заметили, в результате десериализации мы получили объект поля и свойства, которого в точности соответствуют значениям исходного объекта. На рисунке 18.2 показан внешний вид данных в полученном файле.

```
SimpleProject
test.bin ✘ X
00000000 00 01 00 00 00 FF FF FF FF 01 00 00 00 00 00 00 00 .....  
00000010 00 0C 02 00 00 00 44 53 69 6D 70 6C 65 50 72 6F .....DSimplePro  
00000020 6A 65 63 74 2C 20 56 65 72 73 69 6F 6E 3D 31 2E ject, Version=1.  
00000030 30 2E 30 2E 30 2C 20 43 75 6C 74 75 72 65 3D 6E 0.0.0., Culture=n  
00000040 65 75 74 72 61 6C 2C 20 50 75 62 6C 69 63 4B 65 eutral, PublicKey  
00000050 79 54 6F 6B 65 6E 3D 6E 75 6C 6C 05 01 00 00 00 yToken=null.....  
00000060 14 53 69 6D 70 6C 65 50 72 6F 6A 65 63 74 2E 50 .SimpleProject.P  
00000070 65 72 73 6F 6E 03 00 00 00 15 3C 4E 61 6D 65 3E erson.....<Name>  
00000080 6B 5F 5F 42 61 63 6B 69 6E 67 46 69 65 6C 64 14 kBackingField.  
00000090 3C 41 67 65 3E 6B 5F 5F 42 61 63 6B 69 6E 67 46 <Age>kBackingF  
000000a0 69 65 6C 64 0C 5F 69 64 65 6E 74 4E 75 6D 62 65 ield._identNumbe  
000000b0 72 01 00 00 08 08 02 00 00 00 06 03 00 00 00 04 r.....  
000000c0 4A 61 63 6B 22 00 00 00 FB 4A 05 00 0B Jack"....J...
```

Рисунок 18.2. Данные объекта в бинарном файле

Soap форматирование. Класс SoapFormatter

SOAP (*Simple Object Access Protocol* — простой протокол доступа к объектам) — протокол обмена сообщениями при работе с распределенными приложениями. В сущности SOAP определяет стандартный процесс, с помощью которого можно вызывать методы, способом не зависящим от операционной системы.

SOAP форматирование осуществляется объектами класса [SoapFormatter](#), который находится в пространстве имен `System.Runtime.Serialization.Formatters.Soap`. При этом форматировании граф объектов сохраняется в SOAP сообщении, что делает этот вариант форматирования прекрасным выбором при передаче объектов средствами удаленного взаимодействия при помощи различных сетевых протоколов. Класс [SoapFormatter](#) реализует интерфейс [IFormatter](#) и также как и при двоичном форматировании для сериализации и десериализации используются методы `Serialize()` и `Deserialize()` и сериализируются как открытые, так и закрытые поля типа. Пример SOAP форматирования представлен на рисунке 18.3.

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
using static System.Console;

namespace SimpleProject
{
    [Serializable]
    public class Person
    {
```

```
public string Name { get; set; }
public int Age { get; set; }
int _identNumber;
[NonSerialized]
const string Planet = "Earth";

public Person(int number)
{
    _identNumber = number;
}

public override string ToString()
{
    return $"Name: {Name}, Age: {Age},
Identification number: {_identNumber},
Planet: {Planet}.";
}

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person(346875)
        { Name = "Jack", Age = 34 };
        SoapFormatter soapFormat = new SoapFormatter();
        try
        {
            using (Stream fStream =
                File.Create("test.soap"))
            {
                soapFormat.Serialize(fStream, person);
            }
            WriteLine("SoapSerialize OK!\n");
            Person p = null;
            using (Stream fStream =
                File.OpenRead("test.soap"))
            {

```

Урок №10

```
        p = (Person)soapFormat.  
            Deserialize(fStream);  
    }  
    WriteLine(p);  
}  
catch (Exception ex)  
{  
    WriteLine(ex);  
}  
}  
}  
}  
}
```

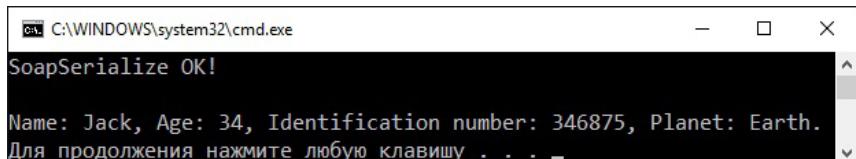


Рисунок 18.3. SOAP форматирование объекта



Рисунок 18.4. Данные объекта в SOAP файле

В результате SOAP сериализации мы получили файл, содержимое которого выглядит как XML, что не удивительно ведь протокол SOAP использует формат XML (Рисунок 18.4).

Сериализация Xml. Класс XmlSerializer

Для сериализации и десериализации объектов в документы формата XML используются методы `Serialize()` и `Deserialize()` класса `XmlSerializer`, который находится в пространстве имен `System.Xml.Serialization`.

В отличии от двух предыдущих форматов сериализации, при XML сериализации наличие атрибута `[Serializable]` у класса необязательно. Также при XML форматировании класса необходимо учитывать несколько особенностей:

- этот класс должен иметь модификатор доступа `public`;
- у этого класса должен быть конструктор не принимающий параметров;
- при XML сериализации учитываются только открытые свойства класса, закрытые и защищенные поля просто игнорируются.

Еще одна особенность XML сериализации заключается в том, что при создании объекта класса `XmlSerializer`, необходимо указывать тип сериализируемого объекта. Приведем пример выполнения XML сериализации класса (Рисунок 18.5).

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

namespace SimpleProject
{
    public class Person
    {
```

```
public string Name { get; set; }
public int Age { get; set; }

int _identNumber;

[NonSerialized]
const string Planet = "Earth";

public Person() { }
public Person(int number)
{
    _identNumber = number;
}

public override string ToString()
{
    return $"Name: {Name}, Age: {Age},
Identification number: { _identNumber},
Planet: {Planet}." ;
}

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person(346875) { Name =
            "Jack", Age = 34 };
        XmlSerializer xmlFormat =
            new XmlSerializer(typeof(Person));
        try
        {
            using (Stream fStream =
                File.Create("test.xml"))
            {
                xmlFormat.Serialize(fStream, person);
            }
        }
    }
}
```

```
        WriteLine("XmlSerialize OK!\n");  
  
    Person p = null;  
    using (Stream fStream =  
          File.OpenRead("test.xml"))  
    {  
        p = (Person)xmlFormat.  
            Deserialize(fStream);  
    }  
    WriteLine(p);  
}  
catch (Exception ex)  
{  
    WriteLine(ex);  
}  
}  
}  
}
```

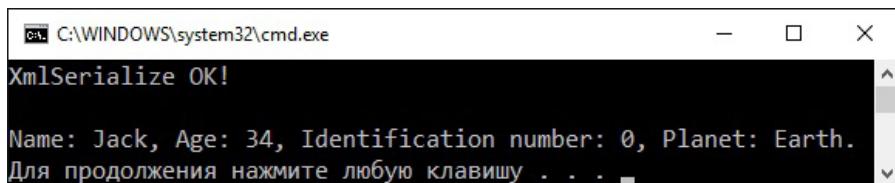


Рисунок 18.5. XML форматирование объекта



Рисунок 18.6. Данные объекта в XML файле

Как вы могли заметить, при XML форматировании поля с модификатором доступа `private` не учитываются в процессе сериализации. Содержимое полученного файла представлено на рисунке 18.6.

Примеры использования сериализации

В качестве примера использования сериализации мы продемонстрируем XML форматирование коллекции объектов. На самом деле процесс форматирования коллекций не представляет особой сложности, так как большинство существующих коллекций поддерживают механизм сериализации (Рисунок 18.7).

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Xml.Serialization;
using static System.Console;

namespace SimpleProject
{
    public class Person
    {
        public string Name { get; set; }
        public int Age { get; set; }
        int _identNumber;
        [NonSerialized]
        const string Planet = "Earth";

        public Person() { }
        public Person(int number)
        {
            _identNumber = number;
        }
    }
}
```

```
public override string ToString()
{
    return $"Name: {Name}, Age: {Age},
Identification number: { _identNumber},
Planet: {Planet}.";
}

class Program
{
    static void Main(string[] args)
    {
        List<Person> persons = new List<Person>()
        {
            new Person(346875) { Name = "Jack", Age = 34 },
            new Person(975648) { Name = "Bob", Age = 37 },
            new Person(870312) { Name = "John", Age = 23 }
        };

        XmlSerializer xmlFormat =
            new XmlSerializer(typeof(List<Person>));
        try
        {
            using (Stream fStream = File.Create("test.xml"))
            {
                xmlFormat.Serialize(fStream, persons);
            }
            WriteLine("XmlSerialize OK!\n");

            List<Person> list = null;
            using (Stream fStream =
                File.OpenRead("test.xml"))
            {
                list = (List<Person>)xmlFormat.
                    Deserialize(fStream);
            }
        }
    }
}
```

```
        foreach (Person item in list)
    {
        WriteLine(item);
    }
}

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

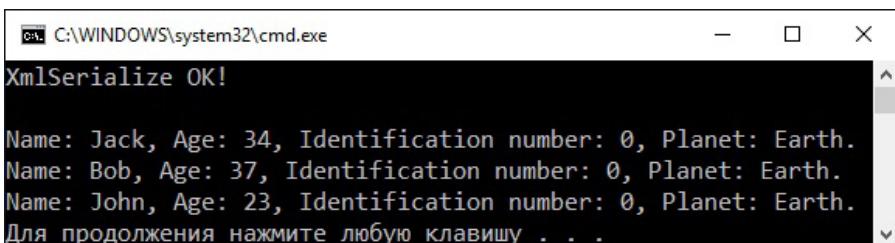


Рисунок 18.7. XML форматирование коллекции объектов

Полученный результат вы можете посмотреть самостоятельно, открыв соответствующий файл.

В следующем примере мы продемонстрируем возможность внесения изменений в объект при его сериализации и десериализации с использованием атрибутов: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]` и `[OnDeserialized]`. Данные атрибуты могут быть применены только к специальным методам, которые не возвращают значение и принимают параметр типа структуры `StreamingContext`, при помощи которой

описывается источник данной сериализации. В зависимости от установленного атрибута, методы будут вызываться автоматически на определенных стадиях процесса форматирования объекта. Если название атрибута оканчивается на «*ing*», то этот метод вызывается во время процесса сериализации или десериализации, если оканчивается на «*ed*» — тогда по завершению соответствующего процесса.

Для большей наглядности в качестве примера приведем SOAP форматирование, хотя бинарное форматирование будет приводить к аналогичному результату (Рисунок 18.8).

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
using static System.Console;

namespace SimpleProject
{
    [Serializable]
    public class Person
    {
        public string Name { get; set; }

        public DateTime DateBirth { get; set; }

        // вызывается во время процесса сериализации
        [OnSerializing]
        private void OnSerializing(StreamingContext context)
        {
            Name = Name.ToUpper();
            DateBirth = DateBirth.ToUniversalTime();
        }
    }
}
```

```
// вызывается по завершении процесса десериализации
[OnDeserialized]
private void OnDeserialized(StreamingContext
                           context)
{
    Name = Name.ToLower();
    DateBirth = DateBirth.ToLocalTime();
}

public override string ToString()
{
    return $"Name: {Name},
           Date of Birth: {DateBirth}.";
}

class Program
{
    static void Main(string[] args)
    {
        Person person = new Person { Name = "Jack",
                                     DateBirth = new DateTime(1995, 11, 5) };
        SoapFormatter soapFormat = new SoapFormatter();
        try
        {
            using (Stream fStream =
                  File.Create("test.soap"))
            {
                soapFormat.Serialize(fStream, person);
            }
            WriteLine("SoapSerialize OK!\n");
            Person p = null;
            using (Stream fStream =
                  File.OpenRead("test.soap"))
            {
                p = (Person)soapFormat.
                    Deserialize(fStream);
            }
        }
    }
}
```

```
        WriteLine(p);
    }
    catch (Exception ex)
    {
        WriteLine(ex);
    }
}
```

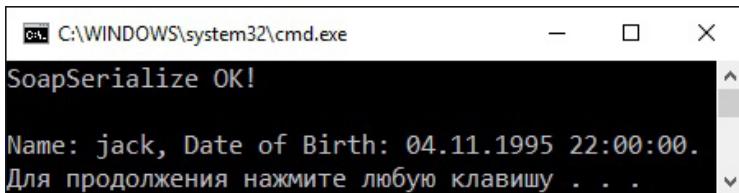


Рисунок 18.8. Внесение изменений при SOAP форматировании объекта

На рисунке 18.9 представлено содержимое полученного SOAP файла.

```
SimpleProject
test.soap + X
1 <SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
    ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
2 </SOAP-ENV:Body>
3 <a1:Person id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SimpleProject/
  SimpleProject%2C%20Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
4   <x003C_Name_x003E_k__BackingField id="ref-3">JACK</x003C_Name_x003E_k__BackingField>
5   <x003C_DateBirth_x003E_k__BackingField>1995-11-04T22:00:00.000000+02:00</
    x003C_DateBirth_x003E_k__BackingField>
6 </a1:Person>
7 </SOAP-ENV:Body>
8 </SOAP-ENV:Envelope>
```

Рисунок 18.9. Измененные данные объекта в SOAP файле

Создание пользовательского формата сериализации. Интерфейс `ISerializable`

Хотя для сериализации объектов в большинстве случаев достаточно существующих механизмов форматирования, в платформе .NET Framework предоставляется возможность создания собственного механизма сериализации и десериализации объекта, например, вам необходимо выполнить определенные операции до и после сериализации данных для изменения их формата. Для этого необходимо чтобы требуемый класс был помечен атрибутом `[Serializable]`, также в нем должен быть реализован интерфейс `ISerializable` и определен специальный конструктор.

В интерфейсе `ISerializable` определен метод `GetObjectData()`, который вызывается в процессе сериализации любым модулем форматирования среды выполнения. Метод `GetObjectData()` принимает два параметра: объект `SerializationInfo`, в котором хранятся пары тип/значение каждой части класса, участвующей в сериализации и структуру `StreamingContext` (знакомую нам по предыдущему разделу).

Специальный конструктор класса вызывается при десериализации объекта и должен принимать два параметра: объект `SerializationInfo` и структуру `StreamingContext`.

Следующий пример демонстрирует сериализацию класса, у которого прописан собственный механизм форматирования. Действия, производимые над свойствами этого класса, аналогичны действиям из предыдущего примера, только в этом случае операции при сериализации и десериализации мы поменяли местами (Рисунок 18.10).

```
using System;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Soap;
using static System.Console;

namespace SimpleProject
{
    [Serializable]
    public class Person : ISerializable
    {
        public string Name { get; set; }
        public DateTime DateBirth { get; set; }
        public Person() { }

        private Person(SerializationInfo info,
                      StreamingContext context)
        {
            Name = info.GetString("PersonName").ToUpper();
            DateBirth = info.GetDateTime("DateBirth").
                ToUniversalTime();
        }

        void ISerializable.GetObjectData(SerializationInfo
            info, StreamingContext context)
        {
            info.AddValue("PersonName", Name.ToLower());
            info.AddValue("DateBirth",
                DateBirth.ToLocalTime());
        }

        public override string ToString()
        {
            return $"Name: {Name},
Date of Birth: {DateBirth}";
        }
    }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person { Name = "Jack",
                                     DateBirth = new DateTime(1995, 11, 5) };

        SoapFormatter soapFormat = new SoapFormatter();

        try
        {
            using (Stream fStream =
                  File.Create("test.soap"))
            {
                soapFormat.Serialize(fStream, person);
            }

            WriteLine("SoapSerialize OK!\n");

            Person p = null;
            using (Stream fStream =
                  File.OpenRead("test.soap"))
            {
                p = (Person)soapFormat.
                    Deserialize(fStream);
            }
            WriteLine(p);
        }

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

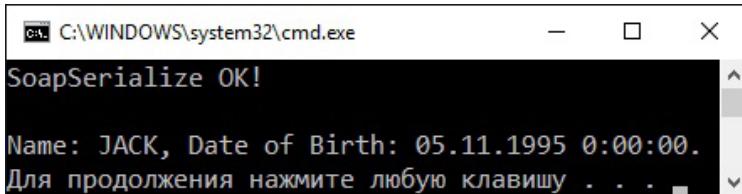


Рисунок 18.10. Применение пользовательской сериализации

В этом примере интерфейс `ISerializable` реализован явным образом, чтобы скрыть от вызывающего кода метод `GetObjectData()`, но сохранить при этом требуемую функциональность. Обращаем также ваше внимание на модификатор доступа `private` у специального конструктора, это сделано для того чтобы нельзя было создать экземпляр данного класса, вызвав этот конструктор из внешнего кода. И в заключении, продемонстрируем содержимое полученного файла (Рисунок 18.11).

```

<?xml version="1.0" encoding="utf-8"?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0" SOAP-
    ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Body>
        <a1:Person id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SimpleProject/
            SimpleProject%2C%20Version%3D1.0.0.%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
            <PersonName id="ref-3">jack</PersonName>
            <DateBirth xsi:type="xsd:dateTime">1995-11-05T02:00:00.0000000+02:00</DateBirth>
        </a1:Person>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

Рисунок 18.11. Данные объекта при пользовательской сериализации

19. Домашнее задание

Разработать класс «Счет для оплаты». В классе предусмотреть следующие поля:

- оплата за день;
- количество дней;
- штраф за один день задержки оплаты;
- количество дней задержки оплаты;
- сумма к оплате без штрафа (вычисляемое поле);
- штраф (вычисляемое поле);
- общая сумма к оплате (вычисляемое поле).

В классе объявить статическое свойство типа `bool`, значение которого влияет на процесс форматирования объектов этого класса. Если значение этого свойства равно `true`, тогда сериализуются и десериализируются все поля, если `false` — вычисляемые поля не сериализуются.

Разработать приложение, в котором необходимо продемонстрировать использование этого класса, результаты должны записываться и считываться из файла.



Урок №10

Взаимодействие с файловой системой. Сериализация

© Юрий Задерей.

© Компьютерная Академия «Шаг»
www.itstep.org.

Все права на охраняемые авторским правом фото-, аудио- и видеопрограммные изделия, фрагменты которых использованы в материале, принадлежат их законным владельцам. Фрагменты произведений используются в иллюстративных целях в объеме, 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». Объем 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.