Microsoft .Net Framework and C# Programming Language

Lesson 10

Interacting with the File System. Serialization

Contents

1. The model of streams in C#. The System.IO space

Virtually all operations associated with the input/output of any sequence of bytes (writing/reading files, using I/O devices, an encryption stream, etc.) in the .NET Framework are performed using streams.

A stream is an abstraction that is used for both reading and writing information. Although streams are associated with different physical devices, their behavior is largely similar and independent of the device, so I/O classes can be used with different types of devices.

The main namespace in the .NET Framework associated with information I/O is System.IO, which contains the required set of classes, structures, delegates and enumerations, most of which will be discussed in this lesson.

2. Stream class

The `Stream` class of the System.IO namespace is the base class for all streams, it provides a universal representation of the various types of input and output, isolating the programmer from separate pieces of operating system and base device information. However, depending on the underlying source or data store, the streams can only support some of the features provided by the `Stream` class.

The `CanRead` and `CanWrite` properties are used to determine the capabilities of the stream to read and write, respectively. The `Read()` and `Write()` methods allow you to read and write data in various formats. If the stream supports seeking (the value of the `CanSeek` property is `true`), you can use the `Seek()` method or the `Position` property to set the position in a specific stream, and the `SetLength()` method to change the length of the stream. Otherwise, to determine the length of the stream, you need to read it to the end.

Some stream implementations perform local buffering of underlying data to improve performance. In such streams, the `Flush()` method is used to clear internal buffers and ensure that all data has been written to the underlying data source or repository.

Calling the `Close()` method of the `Stream` class releases the operating system resources such as file handles, network connections, or the memory used for internal buffering.

3. Analysis of byte stream classes

Classes `FileStream`, `MemoryStream` and `BufferedStream` are the children of the `Stream` class, which are designed to work with different byte streams.

The `FileStream` class is used to perform various file operations (read, write, open, close, etc.), providing access at the byte level. It allows you to work with both binary and text files. However, in case of interaction with the latter, you need to use special methods of conversion from bytes to strings, because the `FileStream` class allows you to work with files only at the byte level. Therefore it makes sense in this case to use special classes for working with text files (`StreamReader`, `StreamWriter`).

Intensive use of data stored in files can lead to a decrease in the performance of your application due to the large number of operations associated with opening and closing files. In such a situation, it makes sense to use the `MemoryStream` class, which allows you to store temporary data in RAM, and thus the work with this data is very fast. In this kind of streams, all data is stored as an array of bytes, the size of which cannot be changed.

The `BufferedStream` class provides a buffer mechanism when performing read and write operations on different streams. This mechanism provides for the use of a reserved memory area, which is used for temporary storage of data — the buffer. The buffering mechanism provides increased

performance when reading and writing data and can be used to synchronize data transfer between devices that run at different speeds.

Byte stream classes allow you to get information in the form of an array of bytes, and if necessary, write this data to a file, otherwise programmers would have to independently convert the bytes into values of the required type. Therefore, in the `System.IO` space, there are wrapper classes that are designed to automatically convert the byte streams to either binary streams or streams that can read/write a sequential series of characters.

4. Analysis of character stream classes

The base classes for working with character streams are `TextReader` and `TextWriter` abstract classes.

The `TextReader` class is the base class for the `StringReader` class that reads data from a string, and the `StreamReader` class, which is used to read characters from a byte stream in a particular encoding. The encoding is specified using the `Encoding` class from the `System.Text` namespace, by default the UTF-8 encoding is used.

The `TextWriter` class is the base class for the classes that write characters: the `StringWriter` class allows writing characters to a string and the `StreamWriter` class writes characters to the byte stream in the required encoding (UTF-8 by default).

5. Analysis of binary stream classes

Wrapper classes, `BinaryReader` and `BinaryWriter`, are used to read and write data in binary format.

The `BinaryReader` class is intended for reading binary data from a stream, it is possible to specify the required encoding, the UTF-8 encoding is used by default. This class has a number of methods that allow you to read various standard data types from the current byte stream.

The `BinaryWriter` class is used to write standard data types in binary format to the stream, with the ability to specify the encoding (UTF-8 by default). There are overloaded methods for writing any standard data type to the stream in the class.

6. Using the FileStream class for file operations

The `FileStream` class provides `Read()` and `Write()` methods that ensure that the byte stream is read and written to the file, respectively. However, before using them, you must create an instance of the `FileStream` class, for which you can use one of the 15 overloaded constructors. The most universal is the constructor, which takes four parameters: the path to the file, the creation mode, the access mode, and the sharing mode.

The path to the file is specified either as a literal or as a variable of `string` type and must contain the name of the file with the extension, and both the absolute and relative path to the file (the Debug folder of the current project) can be specified.

The file creation mode is specified using the `FileMode` enumeration, by specifying one of the following values:

- `FileMode.Append` — opens the file if it exists, and moves the seeks to the end of the file, if the file does not exist — creates a new file; can be used only together with `FileAccess.Write`;
- `FileMode.Create` — creates a new file, if the file already exists, it will be overwritten;
- `FileMode.CreateNew` — creates a new file, if the file already exists, then an `IOException` will be thrown;
- `FileMode.Open` — opens an existing file, if the file does not exist, then the `FileNotFoundException` will be thrown;

- `FileMode.OpenOrCreate` — opens an existing file, if the file does not exist, a new file will be created;
- `FileMode.Truncate` — opens an existing file and truncates its size to zero.

The file access mode is set using the values of the `FileAccess` enumeration and determines the behavior of the stream:

- `FileAccess.Write` — the data can only be written to a file;
- `FileAccess.Read` — the data can only be read from the file;
- `FileAccess.ReadWrite` — the data can be written to a file and read from a file.

Sharing mode indicates how this file will be accessible to other objects and is set using the `FileShare` enumeration values:

- `FileShare.Delete` — allows to delete a file;
- `FileShare.Inheritable` — allows inheritance of a file handle by child processes;
- `FileShare.None` — rejects sharing of the current file, any file open request will fail until the file is closed;
- `FileShare.Read` — allows subsequent opening of the file for reading, if this flag is not set, any request to open the file for reading will fail until the file is closed;
- `FileShare.ReadWrite` — allows subsequent opening of the file for reading or writing, if this flag is not set, any request to open the file for writing or reading will fail until the file is closed;
- `FileShare.Write` — allows subsequent opening of a file for writing, if this flag is not set, any request to open a file for writing will fail until the file is closed.

The `FileAccess` and `FileShare` enumerations support a bitwise connection of values, that is, using the bitwise `OR (|)` operator, several values of the corresponding enumeration can be combined.

An example of using the `FileStream` class is shown in the Figure 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))
            {

                // getting data to write to the file
                WriteLine("Enter the data to write
                          to the file:");
                string writeText = ReadLine();

                // converting the string to the byte array
                byte[] writeBytes = Encoding.Default.
                                    GetBytes(writeText);

                // writing data to the file
                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];
                // reading data from the file
                fs.Read(readBytes, 0, readBytes.Length);
                // converting bytes into the string
                return Encoding.Default.
                       GetString(readBytes);
            }
        }

        static void Main(string[] args)
        {
            string filePath = "test.bin";
            WriteFile(filePath);

            // outputting the result to the console
            WriteLine($"\nData read from the file:
                     {ReadFile(filePath)}");
        }
    }
}
```

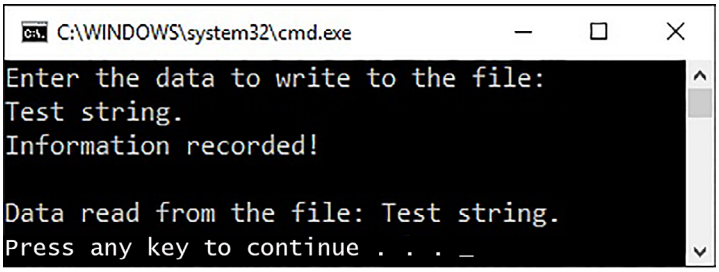

Figure 6.1. Example of using the FileStream class

To ensure the stable operation of your program when working with streams, always remember to call the `Close()` method to free up the resources of the operating system allocated to this stream after finishing work with it. However, as you remember from the seventh lesson, for the classes implementing the `IDisposable` interface, there is a possibility of applying a special syntax with the `using` keyword, which guarantees automatic cleaning of resources. You saw this approach in the previous example when using the `FileStream` class.

7. Using the StreamWriter class for file operations

As mentioned earlier, the `StreamWriter` class allows you to write to a stream of character data, that is, in fact, provides the recording of information in text files. You have already seen the `Write()` and `WriteLine()` methods, which allow you to write data to a text file, when working with the `Console` class. This is not surprising, because the Out property of the `Console` class returns the `TextWriter` type — the standard output stream (console). Here is an example of writing text to a file (Figure 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))
                {
```

```
                    // getting data to write to the file
                    WriteLine("Enter the data to write
                              to the file:");
                    string writeText = ReadLine();
                    // writing data to the file
                    sw.WriteLine(writeText);
                    foreach (var item in writeText)
                    {
                        sw.Write($"{item} ");
                    }
                    WriteLine("\nData recorded!");
                }
            }
        }
    }
}
```

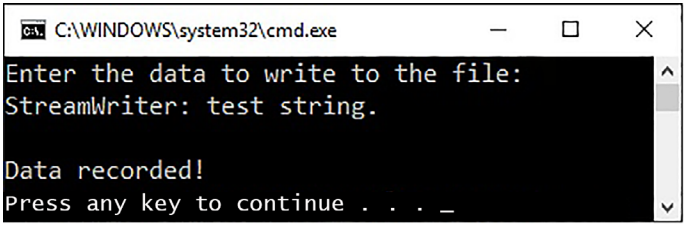

Figure 7.1. Writing data to a text file

Open the resulting file in Notepad to verify that the data was written correctly (Figure 7.2).

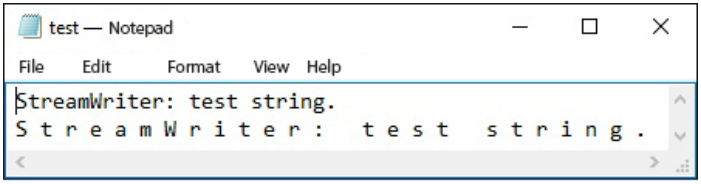

Figure 7.2. Content of the recorded text file

8. Using the StreamReader class for file operations

The `StreamReader` class is used to read information from text files using the appropriate methods:

- `Read()` — reads the next character, or an array of characters;
- `ReadBlock()` — reads an array of characters;
- `ReadLine()` — reads a string from the current position to the newline character;
- `ReadToEnd()` — reads all characters from the stream, starting from the current position to the end.

Again, you already know some of the methods due to the `Console` class, because the In property of the `Console` class returns the `TextReader` type, the standard input stream (keyboard).

Also, the `StreamReader` class has three properties:

- `BaseStream` — returns the base stream;
- `CurrentEncoding` — allows you to determine the current encoding of the stream;
- `EndOfStream` — indicates whether we have reached the end of the stream or not.

Here is an example of reading text from a file (Figure 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))
                {
                    // outputting the result to the console
                    WriteLine($"Data read from the file:\n");
                    WriteLine(sr.ReadToEnd()); // получаем
                    // getting all data from the file
                }
            }
        }
    }
}
```

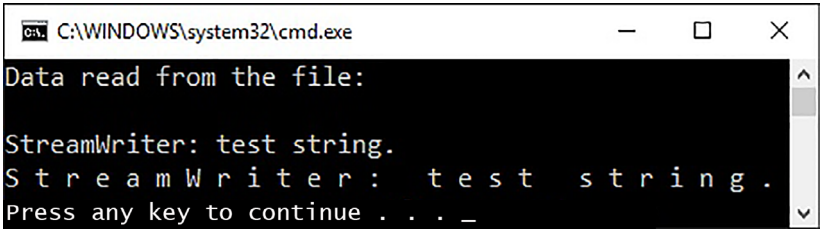

Figure 8.1. Reading data from a text file

9. Using the BinaryWriter class for file operations

The information is written to the files in binary form using the `BinaryWriter` class, for which there are a number of overloaded `Write()` methods that take different types of data. An example of writing binary information to a file is shown below (Figure 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;
```

```
                    // writing data to the file
                    bw.Write(writeText);
                    bw.Write(pi);
                    bw.Write(number);

                    WriteLine("\nData recorded!");
                }
            }
        }
    }
}
```

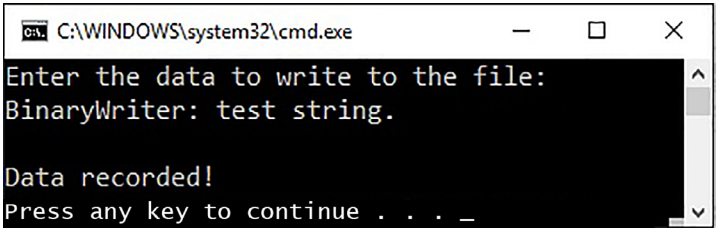

Figure 9.1. Writing binary data to a file

If you open the resulting file in Notepad, we will see that we can read the text information in English, and the rest of the data is unreadable, since it is stored in a binary form (Figure 9.2).

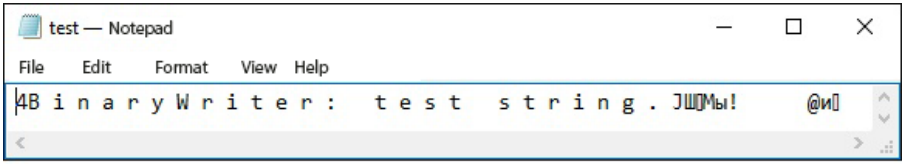

Figure 9.2. Contents of the recorded binary file

Text with Latin letters is not encoded, no matter what type of encoding you indicate when writing. This is due to the fact

that the basis for all text encodings is the ASCII (*American Standard Code for Information Interchange*) encoding, which describes 128 characters — punctuation marks, Arabic numerals and Latin letters. If you want to get a completely unreadable binary file, then enter the text in some other language, for example Russian (we suggest you check it yourself).

10. Using the BinaryReader class for file operations

The `BinaryReader` class is intended for reading information that is stored in files in binary format. For this purpose, a number of methods are used that allow you to read various types of data from a file; they all start with the word Read followed by the name of a specific data type. It should be taken into account that you should read the data in the same order as it was written, we will demonstrate this with an example (Figure 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))
                {
                    // getting data from the file
                    WriteLine($"Data read from the file:\n");
```

```
                    WriteLine(br.ReadString());
                    WriteLine(br.ReadDouble());
                    WriteLine(br.ReadInt32());
                }
            }
        }
    }
}
```

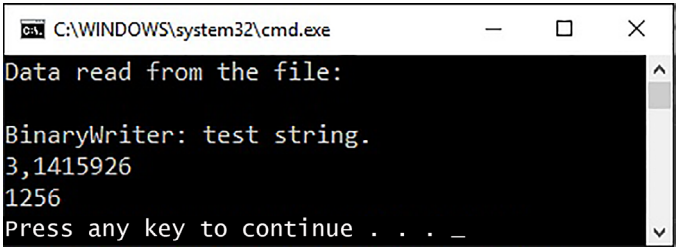

Figure 10.1. Reading data from a binary file

11. Using the Directory, DirectoryInfo, File and FileInfo classes for file operations

In the `System.IO` namespace, there are classes that allow you to perform various operations with directories — `Directory` and `DirectoryInfo`, as well as with file — `File` and `FileInfo`. These classes have similar functionality, but differ in the way they are used. In order to call the methods of the `DirectoryInfo` and `FileInfo` classes, you must create instances of these classes, and the `Directory` and `File` classes are static and provide static methods, respectively. Therefore, if you need an object for multiple use when working with folders or files, it makes sense to use the `DirectoryInfo` or `FileInfo` classes, respectively, and to perform one specific action (creation, opening, copying, etc.), it is most effectively to use the methods of the `Directory` or `File` classes. First, we'll give an example of getting information about the directory using the `DirectoryInfo` class (Figure 11.1).

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

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

```
            // current directory
            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(); // all files in
                                               // the directory
            foreach (FileInfo f in files)
            {
                WriteLine(f.Name);
            }
            WriteLine();
        }
    }
}
```

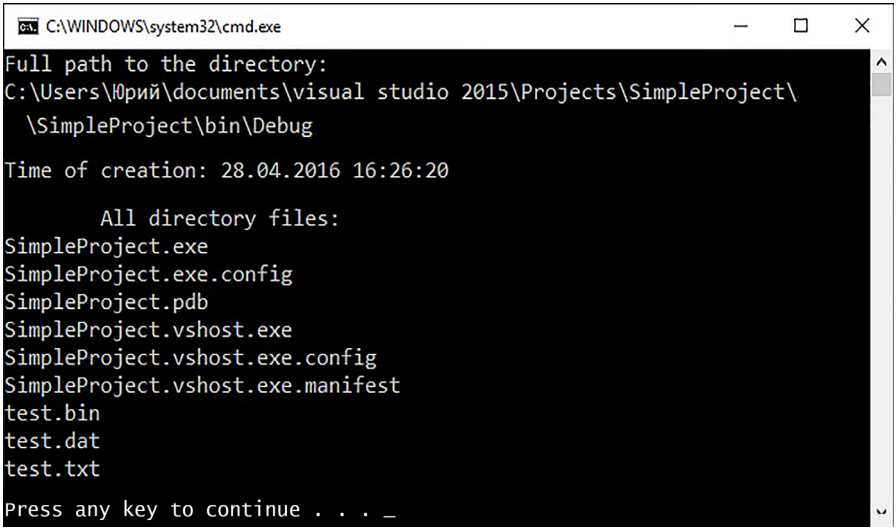

Figure 11.1. Getting information about the current directory

The following example demonstrates the ability to create a directory and subdirectory using the methods of the `DirectoryInfo` class and using the methods of the `FileInfo` class to create a file and write information to it (Figure 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, FileShare.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) // if the directory does not exist
            {
                dir.Create(); // create a directory
            }
            WriteLine($"Last access time to the directory:
                      {dir.LastAccessTime}");

            // create a subdirectory
            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();

        }
    }
}
```

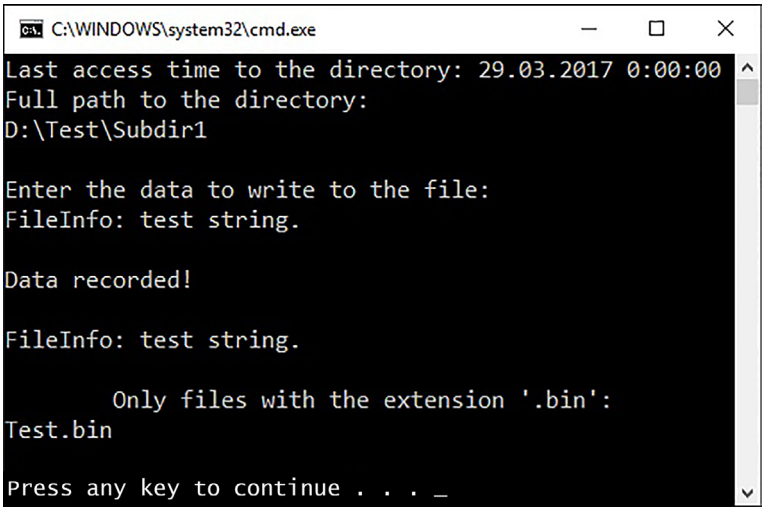

Figure 11.2. Example of using the DirectoryInfo and FileInfo classes

The following example is similar to the previous one, but this time we will create a text file, and we will use the `File` class for this (Figure 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();
        }
    }
}
```

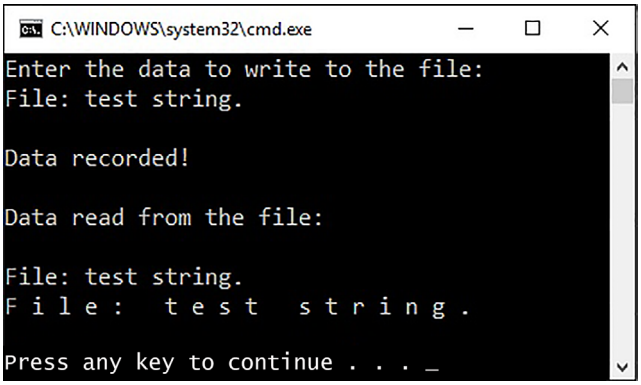

Figure 11.3. Example of using the File class

The last example in this section demonstrates using the `Directory` class to work with directories (Figure 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($"\t{item}");
                }
                // deleting directories,
                // all subdirectories and files
                Directory.Delete(path, true);
            }
            if (!Directory.Exists(path))
            {
                WriteLine($"\nSpecified directory
                          does not exist.\n");
            }
        }
    }
}
```

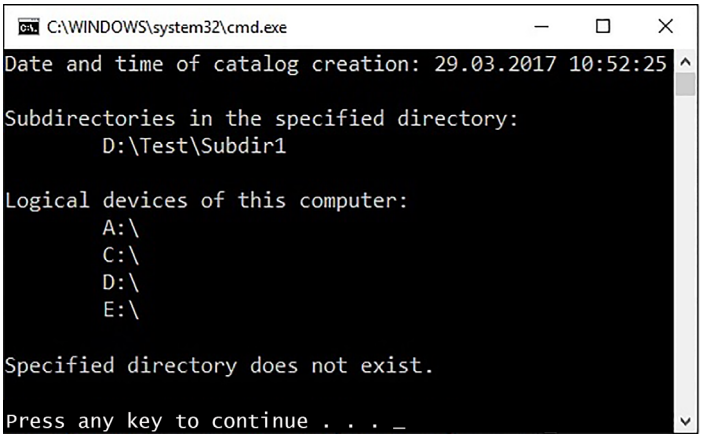

Figure 11.4. Example of using the Directory class

12. Regular expressions

In many programming languages, you can use regular expressions, the meaning of which is to find the strings that satisfy the given pattern, to search for specific information in the text.

The .NET Framework platform is no exception, it uses a number of classes from the `System.Text.RegularExpressions` namespace to support regular expressions, you can get detailed information about them on MSDN.

The main class that is always used when working with regular expressions is the `Regex` class. Usually, when creating instances of this class, a string, which is a regular expression used to search in the text, is transferred to the constructor.

When writing regular expressions, various special characters are used, the presence of each of them has a certain value, and all of them together define the syntax of regular expressions. In Table 12.1, special characters are presented in part, we recommend that you contact MSDN for more information.

Table 12.1. Characters of regular expressions (partially)

Character	Meaning
.	Any single character, except the newline character (\ n)
\w	Matches any alphanumeric character
\W	Matches any non-alphanumeric character
\s	Matches any space character

Character	Meaning
`\S`	Matches any non-space character
`\d`	Matches any decimal digit
`\D`	Matches any character that is not a decimal digit
`[characters]`	Matches any character from the set given in brackets
`[^characters]`	Matches any character, except for the characters specified in brackets
`*`	The previous pattern is repeated 0 or more times
`+`	The previous pattern is repeated 1 or more times
`?`	The previous pattern is repeated 0 or 1 times
`{n, m}`	The previous pattern is repeated at least n and not more than m times
`{n,}`	The previous pattern is repeated n or more times
`{n}`	The previous pattern is repeated exactly n times
`^`	The pattern should be at the beginning of the text
`$`	The pattern should be at the end of the text

As you might have noticed, when writing special regular expression characters, there is a need to use the backslash character (\), which is the escape character. Therefore, to correctly write a string, you either need to escape the character itself, or put the '@' character before the entire string, creating a literal string, the second approach is usually used.

Also, it is possible to pass several values of the `RegexOptions` enumeration connected using the bitwise `OR(|)` operator in the constructor:

- `RegexOptions.Compiled` — indicates that the regular expression will be compiled into the assembly, which leads to faster work, but increases the start time;

- `RegexOptions.CultureInvariant` — indicates that regional language differences are ignored;
- `RegexOptions.ECMAScript` — includes ECMAScript-compatible behavior for a regular expression. This value can only be used in conjunction with the IgnoreCase, Multiline, and Compiled values. Using this value together with any other parameters results in an exception;
- `RegexOptions.ExplicitCapture` — indicates that the only valid entries are explicitly named or numbered groups in the form (?<name>…);
- `RegexOptions.IgnoreCase` — indicates that the regular expression will not be case sensitive;
- `RegexOptions.IgnorePatternWhitespace` — removes unavoidable spaces from the pattern and includes comments marked with "#";
- `RegexOptions.Multiline` — multi-line mode. Changes the value of the "^" and "$" characters so that they coincide, respectively, at the beginning and end of any line, and not only at the beginning and end of the whole line;
- `RegexOptions.None` — indicates that there are no additional parameters;
- `RegexOptions.RightToLeft` — indicates the direction of the search — from right to left;
- `RegexOptions.Singleline` — indicates the use of single-line mode.

Now, here are some examples of using regular expressions (Figure 12.1).

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

namespace SimpleProject
{
    class Program
    {
        static void Main(string[] args)
        {
            string emailPattern = @"^([a-z0-9_-]+\.)*
                                    [a-z0-9_-]+@[a-z0-9_-]+
                                    (\.[a-z0-9_-]+)*
                                    \.[a-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#"));
        }
    }
}
```

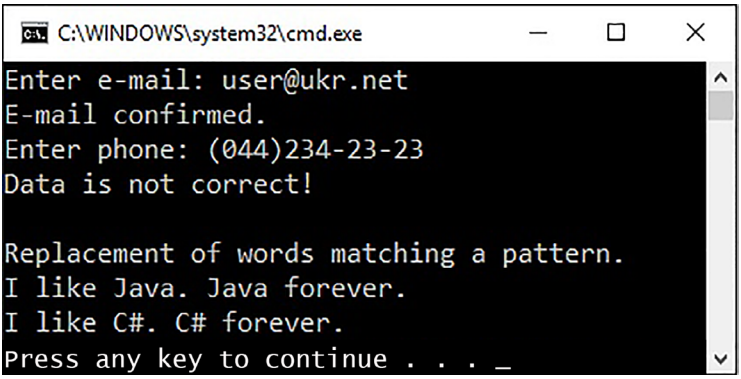

Figure 12.1. Example of using regular expressions

13. The notion of attributes

As you know, when compiling the code, you wrote, an executable module (assembly) is formed, consisting of: the CLR header, metadata, and CIL code. So the attributes in the .NET Framework are special classes, using which you can add additional information (metadata) to the assembly. By default, in any application, there are attributes that describe the current assembly, they are located in the AssemblyInfo.cs file in the Properties folder. When using attributes in the code, metadata remains unused until another piece of code is explicitly used to display the information.

Attributes are specified in square brackets in front of the application element, additional information about which should be added to the assembly. By convention, all attributes end in the word Attribute, for example `ObsoleteAttribute`. However, when writing an attribute, it is allowed not to specify the full name of the attribute, we give some examples:

- `[Obsolete]` — indicates that this item is obsolete, when used in the `Error List` window, an appropriate warning will be displayed;
- `[WebMethod]` — applies only to methods and allows you to call these methods on remote Web clients;
- `[ServiceContract]` — applies to interfaces or classes that define the service contract in the WCF application.

Attributes are applied to any element of the application (assembly, class, method, delegate, etc.) and each of them has its own purpose. In the .NET Framework, there are a lot of standard attributes, but the programmer can create an attribute for his/her own needs, the so-called custom attribute, and they all are children of the `System.Attribute` base class.

When creating a custom attribute, you must use the `[AttributeUsage]` attribute whose constructor can accept up to three parameters.

The first parameter always specifies the value of the `AttributeTargets` enumeration, which specifies the elements of the application to which the attribute is allowed to be applied, multiple values can be connected using the bitwise OR (|)operator.

The second and third parameters are of `bool` type and are optional, and can be specified in any sequence. If you explicitly set the `AllowMultiple` parameter to `true`, then the attribute can be applied multiple times to one element, the default value is `false`. The value of the next `Inherited` parameter, by default, allows inheritance of attribute by child classes (`true` value), explicit assignment of the `false` value prevents inheritance.

As an example, consider creating and using a custom attribute, which will store information about the developer and the creation time of the application element. This attribute can only be applied to classes and methods which is demonstrated on the example of the `Employee` class. In the `Main()` method, we simply display information about all the attributes of this class (Figure 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())
            {
                foreach (var attr in info.
                         GetCustomAttributes(true))
                {
                    WriteLine(attr);
                }
            }
        }
    }
}
```

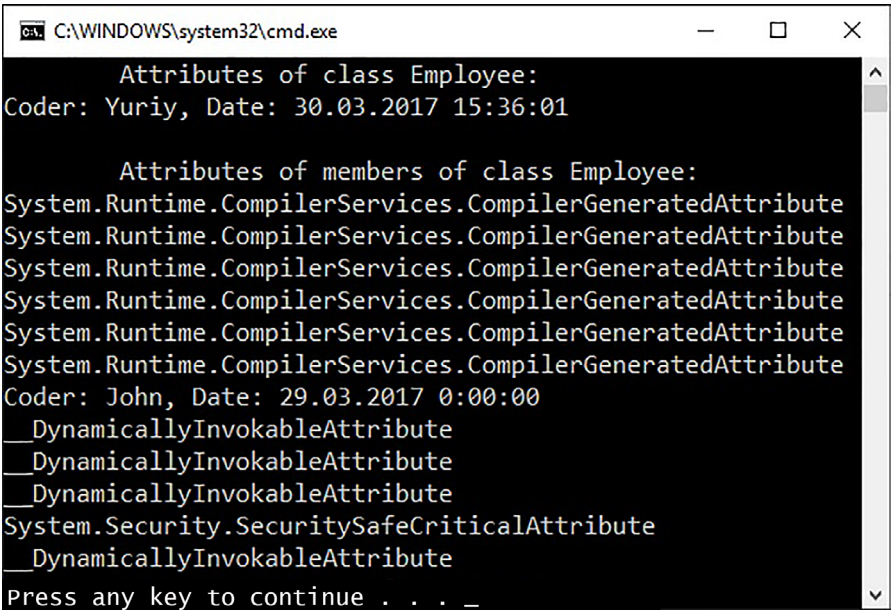

Figure 13.1. Using a custom attribute

14. What is serialization?

Serialization is the process of preserving the state of an object in any thread, with the possibility of its subsequent recovery, while the stored byte sequence contains all the necessary information for restoring this object. Using this technology greatly facilitates the process of saving and retrieving large amounts of application data in a variety of formats, compared to similar operations using standard `System.IO` classes.

15. Relationships between objects

Although storing objects with the help of serialization tools is very simple, it should be understood that the processes occurring in the background are rather complicated. For example, when an object is stored in a stream, all the relevant data (base classes, nested objects, etc.) are also automatically saved. Thus, when you try to serialize a derived class, the whole chain of inheritance will be involved in this process.

During the serialization, the set of interconnected objects is represented as a graph of objects. Serialization services allow you to save the resulting object graph in a variety of formats: binary format, SOAP format, XML format or JSON format.

16. Graphs of object relations

In order to ensure that the object data is correctly saved when it is serialized, the CLR takes into account all objects associated with it. This is a set of related objects and is called an object graph, which provides an easy way to account for the relationships between objects. Each object in the **object graph** is assigned an arbitrary but unique numerical value that has no meaning outside the graph, as soon as this happens, the object graph records all the sets of each object dependencies, all of which happens automatically in the background.

For example, suppose that we have created a set of classes that model the types of cars. We have a base class called Auto that has a `Radio` type field. Another class, `SportAuto`, extends the `Auto` base type. Figure 16.1 shows a possible object graph modeling these relationships.

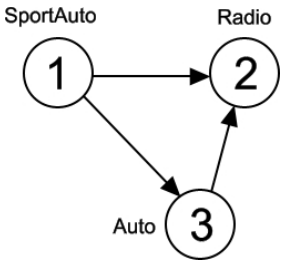

Figure 16.1. Example of an object graph

The relationships shown in the diagram are represented by the CLR as a formula that looks something like this:

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

From this formula, you can see that object 3 `(Auto)` has a relationship with respect to object 2 `(Radio)`, object 2 `(Radio)` is an object which requires nothing and finally, object 1 `(SportAuto)` has a relationship with respect to both object 3 and object 2.

When a `SportAuto` instance is serialized or deserialized, the object graph will guarantee that the Radio and Auto types will also participate in these processes.

The object graph can be stored in any derivative of the `System.IO.Stream` type, the main thing is that the data sequence correctly represents the states of the objects of the corresponding graph.

17. Attributes for serialization [Serializable] and [NonSerialized]

In order for your class or structure to participate in the serialization process, you must specify the `[Serializable]` attribute before declaring them. If there are fields in your class that do not need serialization (random values, fixed or short-term data), they should be marked with the `[NonSerialized]` attribute. These fields will not be saved during serialization, which will reduce the size of the stored data (the object graph).

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

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

The `Person` class is marked with the `[Serializable]` attribute, so all its elements will be serialized, except for the `Planet` constant field (all people live on the planet Earth), because it is marked with the `[NonSerialized]` attribute.

The `[Serializable]` attribute is not inherited, so if your class is inherited from a class with the `[Serializable]` attribute, then your class must also have the `[Serializable]`

attribute, otherwise it will not be saved in the object graph when serializing. When you try to serialize an object that does not support serialization, a `SerializationException` will be thrown in the runtime.

18. Formats of serialization

The .NET Framework provides the ability to serialize in four different formats:

- binary format;
- XML format;
- SOAP format;
- JSON format (will be considered in subsequent courses).

There are certain differences between the above serialization formats.

When using binary formatting, not only the object data fields from the object graph, but also the absolute name of each type, as well as the full name of the assembly unit defining type will be saved. This kind of formatting is very convenient when you need to use the value of objects in .NET applications, both the public and private fields of the type are serialized with this formatting.

SOAP and XML formats are not serialization formats that save the type exactly — they do not record the absolute names of types and assembly blocks. These serialization formats are designed to preserve the state of objects so that they can be used in any operating system, on any application platform (.NET, Java, QT, etc.) or in any programming language. Therefore, there is no need to maintain absolute precision for the type, as there is no guarantee that all possible recipients will be able to understand the .NET-specific data types.

System.Runtime.Serialization.Formatters Space

The `System.Runtime.Serialization.Formatters` space is a nested namespace and provides interfaces and classes that use different formatter classes. In particular, it contains classes that provide information in the event of errors in SOAP serialization:

- `SoapFault` — provides information about errors in the SOAP message;
- `ServerFault` — contains information about errors on the server side.

This namespace is basic for the namespaces required for binary and SOAP serializations (they are discussed in the following sections).

The base for this namespace is the `System.Runtime.Serialization` space, which in turn contains the classes, structures, interfaces, and enumerations used for serialization and deserialization. For example, the `OnSerializing`, `OnSerialized`, `OnDeserializing`, `OnDeserialized` classes with which you can manage the serialization and deserialization processes. In the same namespace, there is the `SerializationException` class — an exception that is generated as a result of serialization and deserialization errors, and the `ISerializable` interface, the implementation of which allows the class to manage its serialization and deserialization processes. For more information about the other members of this namespace, please, address to MSDN.

Binary formatting. The BinaryFormatter class

Binary formatting is performed using objects of the `BinaryFormatter` class, which is the only class of the `System`.

Runtime.Serialization.Formatters.Binary namespace. This class implements the IFormatter interface from the System.Runtime.Serialization namespace, in which two methods are defined, Serialize() and Deserialize(), which perform the serialization and deserialization of the graph of objects into a binary format, respectively. An example of binary formatting is shown below (Figure 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);
            }
        }
    }
}
```

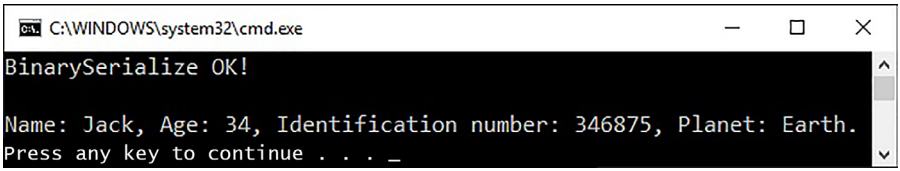

Figure 18.1. Binary object formatting

As you noticed, as a result of deserialization, we got an object, the fields and properties of which exactly correspond to the values of the original object. Figure 18.2 shows the appearance of the data in the resulting file.

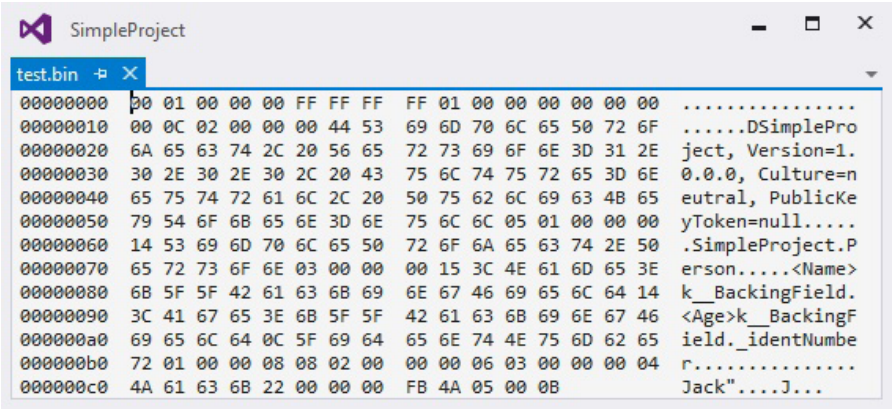

Figure 18.2. Object data in the binary file

Soap formatting. The SoapFormatter Class

SOAP (*Simple Object Access Protocol*) is a messaging protocol for working with distributed applications. In essence, SOAP defines a standard process by which you can call methods in a way that does not depend on the operating system.

SOAP formatting is performed by objects of the `SoapFormatter` class, which is in the `System.Runtime.`

`Serialization.Formatters.Soap` namespace. With this formatting, the object graph is stored in a SOAP message, which makes this formatting an excellent choice when transferring objects by remote interaction using various network protocols. The `SoapFormatter` class implements the `IFormatter` interface, and, like in the binary formatting, the `Serialize()` and `Deserialize()` methods are used for serialization and deserialization, and both the public and private fields of the type are serialized. An example of SOAP formatting is shown in Figure 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"))
                {
                      p = (Person)soapFormat.
                          Deserialize(fStream);
                }
                WriteLine(p);
            }

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

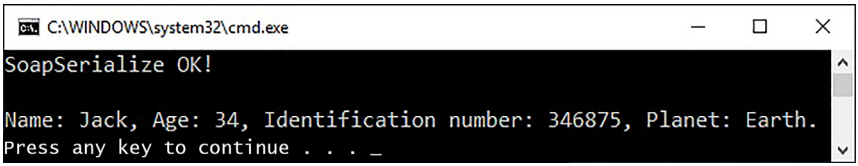

Figure 18.3. SOAP object formatting

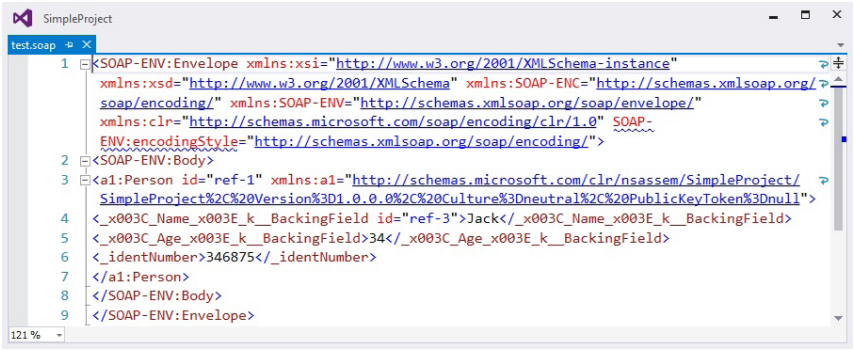

Figure 18.4. Object data in the SOAP file

As a result of SOAP serialization, we got a file whose contents look like XML, which is not surprising because SOAP uses the XML format (Figure 18.4).

Xml serialization. The XmlSerializer class

Serialization and deserialization of objects into XML documents use the `Serialize()` and `Deserialize()` methods of the `XmlSerializer` class, which is in the `System.Xml.Serialization` namespace.

Unlike the two previous serialization formats, in XML serialization, the presence of the `[Serializable]` attribute is optional in the class. Also, when XML formatting a class, you need to consider several features:

- this class must have a `public` access modifier;

- this class must have a constructor that does not accept parameters;
- XML serialization takes into account only the open properties of the class, closed and protected fields are simply ignored.

Another feature of XML serialization is that when you create an object of the `XmlSerializer` class, you must specify the type of the serialized object. Here is an example of the XML serialization of a class (Figure 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);
            }
        }
    }
}
```

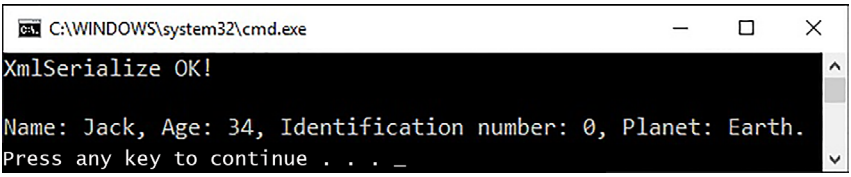

Figure 18.5. XML formatting of the object

Figure 18.6. Object data in the XML file

As you can see, with XML formatting, the fields with the `private` access modifier are not considered in the serialization process. The contents of the resulting file are shown in Figure 18.6.

Examples of using serialization

As an example of using serialization, we demonstrate the XML formatting of a collection of objects. In fact, the process of formatting collections is not particularly difficult, since most existing collections support the serialization mechanism (Figure 18.7).

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

namespace SimpleProject
{
```

```
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);
            }
        }
    }
}
```

```
C:\WINDOWS\system32\cmd.exe
XmlSerialize OK!
Name: Jack, Age: 34, Identification number: 0, Planet: Earth.
Name: Bob, Age: 37, Identification number: 0, Planet: Earth.
Name: John, Age: 23, Identification number: 0, Planet: Earth.
Press any key to continue . . . _
```

Figure 18.7. XML formatting of the collection of objects

You can see the result by yourself, opening the corresponding file.

In the following example, we demonstrate the ability to make changes to an object when it is serialized and deserialized using attributes: `[OnSerializing]`, `[OnSerialized]`, `[OnDeserializing]`, and `[OnDeserialized]`. These attributes can only be applied to special methods that do not return a value and take a parameter of the `StreamingContext` structure type, which describes the source of this serialization. Depending on the attribute set, the methods will be called automatically at certain stages of the object's formatting process. If the name of the attribute ends in "ing", then this method is called during the serialization or deserialization process, if it ends with "ed" — then upon completion of the corresponding process.

For greater clarity, we give SOAP formatting as an example, although binary formatting will lead to a similar result (Figure 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; }
```

```
        // called during the serialization process
        [OnSerializing]
        private void OnSerializing(StreamingContext context)
        {
            Name = Name.ToUpper();
            DateBirth = DateBirth.ToUniversalTime();
        }

        // called upon completion
        // of the deserialization process
        [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);
            }
        }
    }
}
```

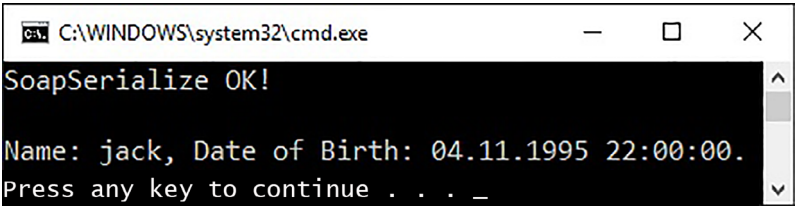

Figure 18.8. Making changes during the SOAP formatting of an object

Figure 18.9 shows the contents of the received SOAP file.

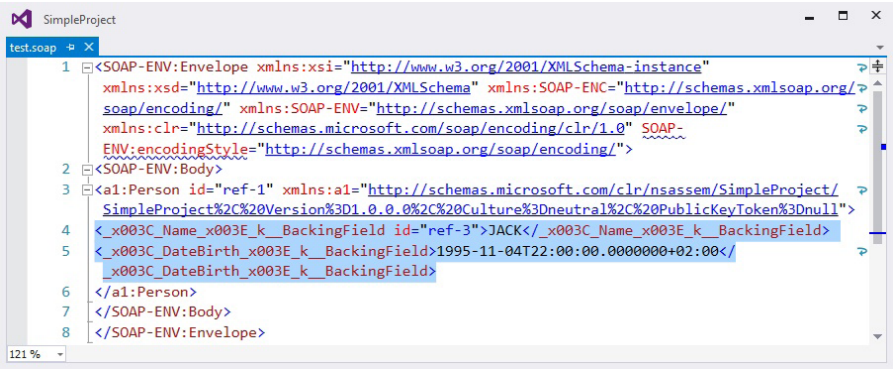

Figure 18.9. Changed object data in the SOAP file

Creating a custom serialization format. The ISerializable interface

Although in most cases, existing formatting mechanisms are sufficient for the serialization of objects, the .NET Framework provides the ability to create your own mechanism for serializing and deserializing an object; for example, you need to perform certain operations before and after serializing data to change their format. To do this, it is necessary that the required class be marked with the `[Serializable]` attribute, also the `ISerializable` interface must be implemented and a special constructor must be defined.

In the `ISerializable` interface, the `GetObjectData()` method is defined, which is called during the serialization process by any runtime formatting module. The `GetObjectData()` method takes two parameters: the `SerializationInfo` object, which stores the type/value pairs of each part of the class that participates in the serialization, and the `StreamingContext` structure (familiar to us from the previous section).

A special class constructor is called when the object is deserialized and it must take two parameters: the `SerializationInfo` object and the `StreamingContext` structure.

The following example demonstrates the serialization of a class that has its own formatting mechanism. The actions performed on the properties of this class are similar to those of the previous example, only in this case we changed the operations for serialization and deserialization (Figure 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);
            }
        }
    }
}
```

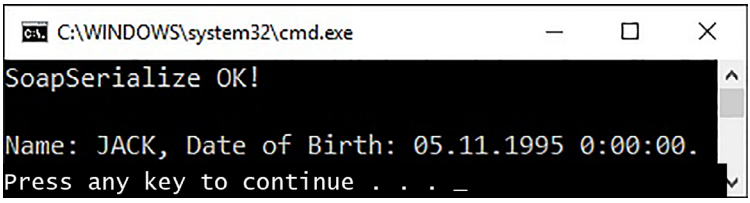

Figure 18.10. Using custom serialization

In this example, the `ISerializable` interface is implemented explicitly to hide the `GetObjectData()` method from the calling code, but to keep the required functionality. We also draw your attention to the `private` access modifier of the special constructor; this is done so that one cannot create an instance of this class by calling this constructor from an external code. Finally, show the contents of the resulting file (Figure 18.11).

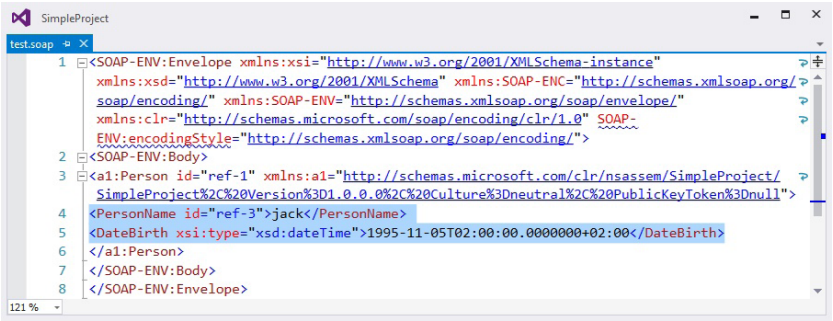

Figure 18.11. Object data with custom serialization

19. Homework assignment

Develop the "Invoice for payment" class. In the class, provide the following fields:

- payment for the day;
- number of days
- penalty for one day of payment delay;
- number of days of payment delay;
- the amount to be paid without penalty (calculated field);
- penalty (calculated field);
- total payment amount (calculated field).

In the class, declare a static property of `bool` type, the value of which affects the process of formatting objects of this class. If the value of this property is `true`, then all fields are serialized and deserialized, if `false` — the calculated fields are not serialized.

Develop an application in which you need to demonstrate the use of this class, the results should be written to and read from the file.

Lesson 10
Interacting with the File System. Serialization