Chapter 15. Text Files

In This Chapter

In this chapter we will review how to **work with text files** in C#. We will explain what a **stream** is, what its purpose is, and how to use it. We will explain what a text file is and how can you **read and write data to a text file** and how to deal with different **character encodings**. We will demonstrate and explain the good practices for exception handling when working with files. All of this will be demonstrated with many examples in this chapter.

Streams

Streams are an **essential part of any input-output library**. You can use streams when your program needs to "read" or "write" data to an external data source such as files, other PCs, servers etc. It is important to say that the term **input** is associated with reading data, whereas the term **output** is associated with writing data.

What Is a Stream?

A **stream** is an **ordered sequence of bytes**, which is send from one application or input device to another application or output device. These bytes are written and read one after the other and always arrive in the same order as they were sent. Streams are an **abstraction of a data communication channel that connects two devices or applications**.

Streams are the primary means of exchanging information in the computer world. Because of streams, different applications are able to access files on the computer and are able to establish network communication between remote computers. In the world of computers, many operations can be interpreted as **reading and writing to a stream**. For example, printing is a process of sending a sequence of bytes to a stream, associated with the corresponding port, to which is the printer connected. Recreating sounds from the computer's sound card can be done by sending some commands, followed by the sample sound, which is actually a sequence of bytes. The scanning of documents from a scanner can be done by sending commands to the scanner (an output stream) and then reading the scanned image (an input stream). This way, you can work with any peripheral device (camera, mouse, keyboard, USB stick, soundcard, printer, scanner etc.).

Every time when you read or write from or to a file, you have to **open a stream** to the corresponding file, **do the reading or writing**, and then

close the stream. There are two types of streams – **text streams** and **binary streams** but this separation has to do with the interpretation of the sent and received bytes. Sometimes, for convenience, a sequence of bytes can be treated as text (in a predefined encoding) and is referred to as a text stream.

Today's modern web sites cannot do without the so-called **streaming**, which represents stream access to bulky multimedia files coming from the Internet. Streaming audio and video allows files to be played before they are downloaded locally, making the site more interactive. Streams and media streaming are different concepts but both use **sequences of data**.

Basic Things You Need to Know about Streams

Many devices **use streams for reading and writing** data. Because of streams, communication between program and file, program and remote computer, is made easy.

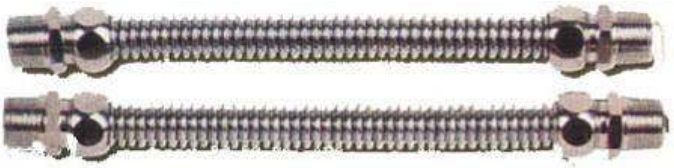
Streams are **ordered sequences of bytes**. The word "order" is intentionally left stressed, because it is of great importance to remember that streams are highly ordered and organized. In no way must you influence the order of the information flow, because it will render it unusable. If a byte is sent to a stream earlier than another byte, it will arrive earlier at the other end of the stream, which is guaranteed by the abstraction "stream".

Streams allow **sequential data access**. Again, it is important to understand the meaning of the word sequential. You can manipulate the data only in the order in which it arrives from the stream. This is closely related to the previous feature. You cannot take the first, than the eight, third, thirteenth byte and so on. Streams **do not allow random access** to their data, only **sequential**. You can think of streams as of a linked list that contains bytes, in which they have a strict order.

Different situations require **different types of streams**. Some streams are used with text files, others-with binary files and then there are those that work with strings. For network communication, you have to use a specific type of stream. The vast variety of streams can help us in different situations, but can also trouble us, because we need to be familiar with every type of stream, before we can use it in our application.

Streams are opened before we can begin working with them and are closed after they have served their purpose. **Closing the stream is very important** and must not be left out, because you risk losing data, damaging the file, to which the stream is opened, and so on – all of these are very troublesome scenarios, which must not happen in our programs.

We can say that streams are like **pipes that connect two points**:



From one side **we pour data in** and from the other **data leaks out**. The one who pours data is not concerned of how it is transferred, but can be sure that what he has poured will come out the same on the other side. Those who use streams do not care how the data reaches them. They know that if someone poured something on the other side, it will reach them. Therefore, we can consider streams as a **data transport channel**, such as pipes.

Basic Operations with Streams

You can do the following **operations with streams**: creation / opening, reading data, writing data, seeking / positioning, closing / disconnecting.

Creation

To **create or open a stream** means to connect the stream to a data source, a mechanism for data transfer or another stream. For example, when we have a file stream, then we pass the file name and the file mode in which it is to be opened (reading, writing or reading and writing simultaneously).

Reading

Reading means extracting data from the stream. Reading is always performed **sequentially** from the current position of the stream. Reading is a **blocking operation**, and if the other party has not sent data while we are trying to read or the sent data has not yet arrived, there may occur a delay – a few milliseconds to hours, days or greater. For example, when reading from a network stream data can be slowed down because of the network or the other party might not have send any data.

Writing

Writing means sending data to the stream in a specific way. The writing is performed from the current position of the stream. Writing may be a potentially **blocking operation**, before the data is sent on its way. For example, if you send bulk data via a network stream, the operation may be delayed while the data is traveling over the network.

Positioning

Positioning or seeking in the stream means to move the current position of the stream. Moving is done according to the current position, where we can position according to the current position, beginning of the stream, or the end of the stream. Moving can be done only in **streams that support positioning**. For example, file streams typically maintain positioning while network streams do not.

Closing

To close or disconnect a stream means to complete the work with the stream and **releases the occupied resources**. Closing must take place as soon as possible after the stream has served its purpose, because a resource opened by a user, usually cannot be used by other users (including other programs on the same computer that run parallel to our program).

Streams in .NET – Basic Classes

In .NET Framework **classes for working with streams** are located in the namespace **`System.IO`**. Let's focus on their hierarchy, organization and functionality.

We can distinguish two main types of streams – those who work with **binary data** and those who work with **text data**. Later we will discuss the main characteristics of these two types.

At the top of the stream hierarchy stands an **abstract input-output stream class**. It cannot be instantiated, but defines the basic functionality that all the other streams have.

There are **buffered streams** that do not add any extra functionality, but use a buffer for reading and writing data, which significantly enhances performance. Buffered streams will not be analyzed in this chapter, as we will focus on working with text files. You can check with the rich documentation available on the Internet or a textbook for advanced programming.

Some streams **add additional functionality** to reading and writing data. For example, there are streams that compress / decompress data sent to them and streams that encrypt / decrypt data. These streams are connected to another stream (such as file or network stream) and add additional processing to its functionality.

The main classes in the **`System.IO`** namespace are **`Stream`** (abstract base class for all streams in .NET Framework), **`BufferedStream`**, **`FileStream`**, **`MemoryStream`**, **`GZipStream`** and **`NetworkStream`**. We will discuss in more details some of them, separating them in their basic feature – the type of data with which they work.

All streams in C# are similar in one basic thing – **it is mandatory to close them** after we have finished working with them. Otherwise we risk damaging the data in the stream or file that we have opened. This brings us to the first and basic rule that we should always remember when working with streams:

Always close the streams and files you work with! Leaving an open stream or file leads to loss of resources and can block the work of other users or processes in your system.

Binary and Text Streams

As we mentioned earlier, we can divide the streams into two large groups according to the type of data that we deal with – **binary streams** and **text streams**.

Binary Streams

Binary streams, as their name suggests, work with **binary (raw) data**. You probably guess that that makes them universal and they can be used to read information from all sorts of files (images, music and multimedia files, text files etc.). We will take a brief look over them, because we will currently focus on working with text files.

The main classes that we use to read and write from and to binary streams are: **`FileStream`**, **`BinaryReader`** and **`BinaryWriter`**.

The class **`FileStream`** provides us with various methods for **reading and writing from a binary file** (read / write one byte and a sequence of bytes), skipping a number of bytes, checking the number of bytes available and, of course, a method for closing the stream. We can get an object of that class by calling him his constructor with parameter-a file name.

The class **`BinaryWriter`** enables you to **write primitive types and binary values** in a specific encoding to a stream. It has one main method – **`Write(…)`**, which allows recording of any primitive data types – integers, characters, Booleans, arrays, strings and more.

`BinaryReader` allows you to **read primitive data types and binary values** recorded using a **`BinaryWriter`**. Its main methods allow us to read a character, an array of characters, integers, floating point, etc. Like the previous two classes, we can get on object of that class by calling its constructor.

Text Streams

Text streams are very similar to binary, but only **work with text data** or rather a sequence of characters (**`char`**) and strings (**`string`**). Text streams are ideal for working with text files. On the other hand, this makes them unusable when working with any binaries.

The main classes for working with text streams in .NET are **`TextReader`** and **`TextWriter`**. They are abstract classes, and they cannot be instantiated. These classes define the basic functionality for reading and writing for the classes that inherit them. Their more important methods are:

- **`ReadLine()`** – reads one line of text and returns a string.
- **`ReadToEnd()`** – reads the entire stream to its end and returns a string.
- **`Write()`** – writes a string to the stream.
- **`WriteLine()`** – writes one line of text into the stream.

As you know, the characters in .NET are Unicode characters, but **streams can also work with Unicode and other encodings** like the standard encoding for Cyrillic languages Windows-1251.

The classes, to which we will turn our attention to in this chapter, are **StreamReader** and **StreamWriter**. They directly inherit the **TextReader** and **TextWriter** classes and implement functionality for **reading and writing textual information to and from a file**.

To create an object of type **StreamReader** or **StreamWriter**, we need a file or a string, containing the file path. Working with these classes, we can use all of the methods that we are already familiar with, to work with the console. Reading and writing to the console is much like reading and writing respectively with **StreamReader** and **StreamWriter**.

Relationship between Text and Binary Streams

When writing text, hidden from us, the class **StreamWriter transforms the text into bytes** before recording it at the current position in the file. For this purpose, it uses the character encoding, which is set during its creation. The **StreamReader** class works similarly. It uses **StringBuilder** internally and when reading binary data from a file, it **converts the received bytes to text** before sending the text back as a result from reading.

Remember that the **operating systems have no concept of "text file"**. The file is always a sequence of bytes, but whether it is text or binary depends on the interpretation of these bytes. If we want to look at a file or a stream as text, we must read and write to it with text streams (**StreamReader** or **StreamWriter**), but if we wish to treat it as binary, we must read and write with a binary stream (**FileStream**).

Bear in mind that text streams work with text lines, that is, they **interpret binary data as a sequence of text lines**, separated from each other with a new line separators.

The **character for the new line** is not the same for different platforms and operating systems. For UNIX and Linux it is **LF (0x0A)**, for Windows and DOS it is **CR + LF (0x0D + 0x0A)**, and for Mac OS (up to version 9) it is **CR (0x0A)**. Reading one line of text from a given file or a stream means reading a sequence of bytes until reading one of the characters **CR** or **LF** and converting these bytes to text according to the encoding, used by the stream. Similarly, writing one line of text to a text file or stream means writing the binary representation of the text (according to the current encoding), followed by the character (or characters) for a new line for the current operating system (such as **CR + LF**).

Reading from a Text File

Text files provide the ideal solution for reading and writing data. If we want to enter some data automatically (instead by hand), we could read it from a text

files. So now, we will take a look at how to read and write text files with the classes from .NET Framework and the C# language.

StreamReader Class for Reading a Text File

C# provides several ways to read files but not all are easy and intuitive to use. This is why we will use the **StreamReader** class. The **System.IO.StreamReader** class provides the easiest way to read a text file, as it resembles reading from the console, which by now you have probably mastered to perfection.

Having read everything until now, you are probably a bit confused. We already explained that reading and writing to and from text files is only and exclusively possible with streams, but **StreamReader** did not appear anywhere in the above-mentioned streams and you are not sure whether it is actually a stream. Indeed, **StreamReader** is not a stream, but it can work with streams. It provides the easiest and comprehensive way to read from a text file.

Opening a Text File for Reading

You can simply create a **StreamReader** from a filename (or full file path), which greatly eases us and reduces the probability of an error. On its creation, we can specify the character encoding. Here is an example of how an object of the class **StreamReader** can be created:

```
// Create a StreamReader connected to a file
StreamReader reader = new StreamReader("test.txt");

// Read the file here …

// Close the reader resource after you've finished using it
reader.Close();
```

The first thing to do, when reading from a text file, is to create a variable of type **StreamReader**, which we can associate with a specific file from the file system on our computer. To do this we need only pass the file path as a parameter to the constructor. Note that if the file is located in the folder where the compiled project (subdirectory **bin\Debug**) is, we can only provide its filename. Otherwise, we have to provide the full file path or relative path.

The code in the above example that creates an object of type **StreamReader** can cause an error. For now, simply pass a path to an existing file, and later on we will turn to the handling of errors when working with files.

Full and Relative Paths

When working with files we can **use full paths** (e.g. **C:\Temp\example.txt**) or **relative paths**, to the directory from which the application was started (e.g. **..\..\example.txt**).

If you use full paths, where you pass the full path to a file, do not forget to apply escaping of slashes, which is used to separate the folders. In C# you can do this in two ways – with a double slash or with a quoted string beginning with **@** before the string literal. For example, to enroll the path to the file "**C:\Temp\work\test.txt**" in a string we have two options:

```
string fileName = "C:\\Temp\\work\\test.txt";
string theSamefileName = @"C:\Temp\work\test.txt";
```

Although the use of relative paths is more difficult because you have to take into account the directory structure of your project which may change during the life of the project, it is **highly recommended avoiding full paths**.

Avoid full file paths and work with relative paths! This makes your application portable and easy for installation and maintenance.

Using the full path to a file (e.g. **C:\Temp\test.txt**) is bad practice because it makes your application **dependent on the environment** and also non-transferable. If you transfer it to another computer, you will need to correct paths to the files, which it seeks, to work correctly. If you use a relative path to the current directory (e.g. **..\..\example.txt**), your program will be easily portable.

Remember that when you start the C# program, the current directory is the one, in which the executable (.exe) file is located. Most often this is the subdirectory bin\Debug or bin\Release directory to the root of the project. Therefore, to open the file example.txt from the root directory of your Visual Studio project, you should use a relative path ..\..\example.txt.

Universal Relative to Physical Path Resolver

If you want to write a portable application, you might benefit of Nakov's universal path resolver: http://www.nakov.com/blog/2009/07/14/universal-relative-to-physical-path-resolver-for-console-wpf-and-aspnet-apps/. It can automatically **resolve a relative path to full (physical) file path** in Web, desktop, console or other .NET application. For example, if your application consists of an assembly **App.exe** and a file **logo.gif** and these files are located in the same directory, at runtime you will be able to get the physical location of **logo.gif** through the following code:

```
string logoPath =
  UniversalFilePathResolver.ResolvePath(@"~\logo.gif");
```

Reading a Text File Line by Line – Example

Now, we have learned how to create **StreamReader**. We can go further by trying to do something more complicated: to **read the entire text file line by line** and print the read text on to the console. Our advice is to create the text file in the **Debug** folder of the project (**.\bin\Debug**), so that it will be in the same directory in which your compiled application will be and you will not have to set the full path to it when opening the file. Let's see what our file looks like:

Sample.txt

```
This is our first line.
This is our second line.
This is our third line.
This is our fourth line.
```

We have a text file from which to read. Now we must create an object of type **StreamReader** to read the file and loop though it line by line:

FileReader.cs

```
class FileReader
{
  static void Main()
  {
    // Create an instance of StreamReader to read from a file
    StreamReader reader = new StreamReader("Sample.txt");

    int lineNumber = 0;

    // Read first line from the text file
    string line = reader.ReadLine();

    // Read the other lines from the text file
    while (line != null)
    {
      lineNumber++;
      Console.WriteLine("Line {0}: {1}", lineNumber, line);
      line = reader.ReadLine();
    }

    // Close the resource after you've finished using it
    reader.Close();
  }
}
```

There is nothing difficult to read text files. The first part of our program is already well known – create a variable of type **StreamReader**, to whose constructor we pass the file's name, which will be read. The parameter of the constructor is the path to the file, but since our file is found in the **Debug** directory of the project, we set only its name as a path. If our file were located in the project directory, then we would have submitted the string – **"..\..\Sample.txt"** as a path.

After that, we create a variable – counter, whose purpose is to count and display on which row of the file we are currently located.

Then, we create a variable that will store each read line. With its creation, we directly read the first line of text file. If the text file is empty, the method **ReadLine()** of the **StreamReader** object will return **null**.

For the main part – reading the file line by line, we will use a **while** loop. The condition for the loop is: as long as there is something in the variable **line**, we should continue reading. In the body of the loop, our task is to increase the value of the counter variable by one and then print the current line in the format we like. Finally, again we use **ReadLine()** to read the next line in the file and write it in the variable **line**. For printing, we use a method that is well known to us from the tasks, which required something to be printed on to the console – **WriteLine()**.

Once we have read everything we need from the file, we should not forget to close the object **StreamReader**, as to avoid loss of resources. For this, we use the method **Close()**.

Always close the StreamReader instances after you finish working with them. Otherwise you risk losing system resources. Use the method Close() or the statement using.

The result of the program should look like this:

```
Line 1: This is our first line.
Line 2: This is our second line.
Line 3: This is our third line.
Line 4: This is our fourth line.
```

Automatic Closing of the Stream after Working with It

As noted in the previous example, having finished working with the object of type **StreamReader**, we called **Close()** and closed the stream behind the **StreamReader** object. Very often, however, novice programmers forget to call the **Close()** method, thus blocking the file they use. Also in case of runtime exception when reading from a file, the file might be left open. This causes resource leakage and can lead to very unpleasant effects like program hanging, program misbehavior and strange errors.

The correct way to handle the file closing is though the **using** keyword:

```
using (<stream object>) { … }
```

The C# construct **using(…)** ensures that after leaving its body, the method **Close() will automatically be called**. This will happen even if an exception occurs when reading the file.

Now let's rework the previous example to benefit from the **using** construct:

FileReader.cs

```
class FileReader
{
   static void Main()
   {
      // Create an instance of StreamReader to read from a file
      StreamReader reader = new StreamReader("Sample.txt");

      using (reader)
      {
         int lineNumber = 0;

         // Read first line from the text file
         string line = reader.ReadLine();

         // Read the other lines from the text file
         while (line != null)
         {
            lineNumber++;
            Console.WriteLine("Line {0}: {1}", lineNumber, line);
            line = reader.ReadLine();
         }
      }
   }
}
```

Now the code guarantees that once opened successfully, **the text file will be closed correctly** regardless of whether reading from it will succeed or fail.

If you are wondering how it is best to take care of closing your program's streams and files, follow the following rule:

Always use the using construct in C# in order to properly close files and streams!

File Encodings. Reading in Cyrillic

Let's now consider the problems that occur when reading a file using an incorrect encoding, such as reading a file in Cyrillic.

Character Encodings

You know that in memory **everything is stored in binary form**. This means that it is necessary for text files to be represented digitally, so that they can be stored in memory, as well as on the hard disk. This process is called **encoding files** or more correctly encoding the characters stored in text files.

The **encoding process** consists of replacing the text characters (letters, digits, punctuation, etc.) with specific sequences of binary values. You can imagine this as a large table in which each character corresponds to a certain value (sequence of bytes).

We already know **the concept of character encodings** and few character encoding schemes like **`UTF-8`** and **`Windows-1251`** from the section "Encoding Schemes" of chapter "Numeral Systems and Data Representation" and also from the section about "File Encodings in Visual Studio" of chapter "Defining Classes". Now we will extend this concept a bit and will use character encodings to work correctly with text files.

Character encodings specify the rules for converting from text to sequence of bytes and vice versa. An encoding scheme is a table of characters along with their numbers, but may also contain special rules. For example, the character "accent" (U + 0300) is special and sticks to the last character that precedes it. It is encoded as one or more bytes (depending on the character encoding scheme), and it does not correspond to any character, but to a part of the character. We will take a look at two encodings that are used most often when working with Cyrillic: **`UTF-8`** and **`Windows-1251`**.

`UTF-8` is a **universal encoding scheme**, which supports all languages and alphabets in the world. In UTF-8 the most commonly used characters (Latin alphabet, numerals and special characters) are encoded in one byte, rarely used Unicode characters (such as Cyrillic, Greek and Arabic) are encoded in two bytes and all other characters (Chinese, Japanese and many others) are encoded in 3 or 4 bytes. UTF-8 encoding can convert any Unicode text in binary form and back and support all of the 100,000 characters of Unicode standard. UTF-8 encoding is universal and suitable for any language alphabet.

Another commonly used encoding is **`Windows-1251`**, which is usually used for **encoding of Cyrillic texts** (such as messages sent by e-mail). It contains 256 characters, including the Latin alphabet, Cyrillic alphabet and some commonly used signs. It uses one byte for each character, but at the expense of some characters that cannot be stored in it (as the Chinese alphabet characters), and are lost in an attempt of doing so.

Other examples of encoding schemes (encodings or charsets) are **`ISO 8859-1`**, **`Windows-1252`**, **`UTF-16`**, **`KOI8-R`**, etc. They are used in specific regions of

the world and define their own sets of characters and rules for the transition from text to binary data and vice versa.

For working with encodings (charsets) in .NET Framework, the class **System.Text.Encoding** is used, which is created the following way:

```
Encoding win1251 = Encoding.GetEncoding("Windows-1251");
```

Reading a Cyrillic Content

You probably already guessed that if we want to read from a file that contains **characters from the Cyrillic alphabet**, we must use the correct encoding that "understands" correctly these special characters. Typically, in a Windows environment, text files, containing Cyrillic text, are stored in **Windows-1251** encoding. To use it, we should set it as the encoding of the stream, which our **StreamReader** will process:

```
Encoding win1251 = Encoding.GetEncoding("Windows-1251");
StreamReader reader = new StreamReader("test.txt", win1251);
```

If you do not explicitly set the encoding scheme (encoding) for the file read, in .NET Framework, the default encoding **UTF-8** will be used.

You might wonder what happens if you use wrong encoding when reading or writing a file. There are several scenarios possible:

- If you use read / write only Latin letters, everything will work normally.
- If you write Cyrillic letters, to a files open with encoding, which does not support the Cyrillic alphabet (e.g. **ASCII**), Cyrillic letters will be permanently replaced by the character "**?**" (question mark).

In any case, these are unpleasant problems, which cannot be immediately noticed.

To avoid problems with incorrect encoding of files, always check the encoding explicitly. Otherwise, you may work incorrectly or break at a later stage.

The Unicode Standard. Reading in Unicode

Unicode is an industry standard that allows computers and other electronic devices always to present and manipulate text, which was written in most of the world's literacies. It consists of over 100,000 characters, as well as various encoding schemes (encodings). The unification of different characters, which Unicode offers, leads to its greater distribution. As you know, characters in C# (types **char** and **string**) are also presented in Unicode.

To read characters, stored in Unicode, we must use one of the supported encoding schemes for this standard. The most popular and widely used is **UTF-8**. We can set it as a code scheme with an already familiar way:

```
StreamReader reader = new StreamReader("test.txt",
  Encoding.GetEncoding("UTF-8"));
```

If you are wondering, whether to read a text file, encoded in Cyrillic, **Windows-1251** or **UTF-8**, then this question has no clear answer. Both standards are widely used for the recording of non-Latin text. Both encoding schemes are allowed and can be used. You should only always follow the rule that a **certain files should always be read and written using the same encoding**.

Writing to a Text File

Text files are very convenient for storing various types of information. For example, we can record the results of a program. We can use text files to make something like a journal (log) for the program – a convenient way to monitor it at runtime.

Again, as with reading a text file, we will use a similar to the **Console** class when writing, called **StreamWriter**.

The StreamWriter Class

The class **StreamWriter** is part of the **System.IO** namespace and is used exclusively for working with text data. It resembles the class **StreamReader**, but instead of methods for reading, it offers similar methods for writing to a text file. Unlike other streams, before writing data to the desired destination, **StreamWriter** turns it into bytes. **StreamWriter** enables us to set a preferred character encoding at the time it is created. We can create an instance of the class the following way:

```
StreamWriter writer = new StreamWriter("test.txt");
```

In the constructor of the class can pass as a parameter a file path, as well as an existing stream, to which we will write, or an encoding scheme. The **StreamWriter** class has several predefined constructors, depending on whether we will write to a file or a stream. In the examples, we will use the constructor with the parameter – file path. Example of the usage of the **StreamWriter** class constructor with more than one parameter is:

```
StreamWriter writer = new StreamWriter("test.txt",
  false, Encoding.GetEncoding("Windows-1251"));
```

In this example, we pass a file path as the first parameter. As a second parameter, we pass a Boolean variable that indicates whether to overwrite the file or to append the data at the end of the file. As a third parameter, we pass an encoding scheme (charset).

The example lines of code could trigger an exception, but the handling of input / output exceptions will be discussed later in this chapter.

Printing the Numbers [1…20] in a Text File – Example

Once we know how to create a **StreamWriter** class, we will use it as intended. Our goal is to enroll in a text file the numbers from 1 to 20, each number on a separate line. We can do this the following way:

```
class FileWriter
{
   static void Main()
   {
      // Create a StreamWriter instance
      StreamWriter writer = new StreamWriter("numbers.txt");

      // Ensure the writer will be closed when no longer used
      using(writer)
      {
         // Loop through the numbers from 1 to 20 and write them
         for (int i = 1; i <= 20; i++)
         {
            writer.WriteLine(i);
         }
      }
   }
}
```

We start by creating an instance of **StreamWriter** in the already well-known way from the examples.

To list the numbers from 1 to 20 we will use a **for**-loop. Inside the loop, we use the method **WriteLine(…)**, which again we know from our previous work with the console, to record the current number on a new line in the file. You need not worry if a file with the chosen name does not exist. If such the case, it will automatically be created in the folder of the project and if it already exists, it will be overwritten (old content will be deleted). The outcome is:

numbers.txt

```
1
2
3
…
20
```

To make sure that after the end of the file it will be closed, we should use the **using** construct.

Be sure to close the stream after you finish using it! The best way to dispose any unused resources is with the using construct in C#.

When you want to print text in Cyrillic and are unsure what encoding to use, prefer the UTF-8 encoding. It is universal and not only supports Cyrillic, but all widespread international alphabets: Greek, Arabic, Chinese, Japanese, etc.

Input / Output Exception Handling

If you have followed the examples so far, you have probably noticed that many of the operations, related to files, can cause exceptional situations. The basic principles and approaches for their capture and processing are already familiar to you from the chapter "Handling Exceptions". Now we will concentrate on the specific **errors when working with files** and best practices for their handling.

Intercepting Exceptions when Working with Files

Perhaps the most common exception when working with files is the **FileNotFoundException** (its name infers that the desired file was not found). It can occur when creating **StreamReader**.

When setting a specified encoding by the creation of a **StreamReader** or a **StreamWriter** object, an **ArgumentException** can be thrown. This means, that the encoding we have chosen is not supported.

Another common mistake is **IOException**. This is the base class for all IO errors when working with streams.

The standard approach for handling exceptions when working with files is the following: declare variables of class **StreamReader** or **StreamWriter** in **try-catch** block. Initialize them with the necessary values in the block and handle the potential exceptions properly. To close the stream, we use the structure **using**. To illustrate what we just said, will give an example.

Catching an Exception when Opening a File – Example

Here's how we can catch exceptions that occur when working with files:

```
class HandlingExceptions
{
  static void Main()
  {
    string fileName = @"somedir\somefile.txt";
    try
```

```
    {
      StreamReader reader = new StreamReader(fileName);
      Console.WriteLine(
        "File {0} successfully opened.", fileName);
      Console.WriteLine("File contents:");
      using (reader)
      {
        Console.WriteLine(reader.ReadToEnd());
      }
    }
    catch (FileNotFoundException)
    {
      Console.Error.WriteLine(
        "Can not find file {0}.", fileName);
    }
    catch (DirectoryNotFoundException)
    {
      Console.Error.WriteLine(
        "Invalid directory in the file path.");
    }
    catch (IOException)
    {
      Console.Error.WriteLine(
        "Can not open the file {0}", fileName);
    }
  }
}
```

The example demonstrates reading a file and printing its contents to the console. If we accidentally have confused the name of the file or have deleted it, an exception of type **FileNotFoundException** will be thrown. In the **catch** block we intercept this sort of exception and if such occurs, we will process it properly and print a message to the console, saying that this file cannot be found. The same will happen if there were no directory named **"somedir"**. Finally, for better security, we have also added a **catch** block for **IOExceptions**. There all other IO exceptions, that might occur when working with files, will be intercepted.

Text Files – More Examples

We hope the theoretical explanations and examples so far have helped you get into the subtleties when working with text files. Now we will take a look at some **more complex examples**, so as to review the gained knowledge and to illustrate how to use them in solving practical problems.

Occurrences of a Substring in a File – Example

Here is how to implement a simple program that counts how many times a substring occurs in a text file. In the example, let's look for the substring "**C#**" in a text file as follows:

sample.txt

```
This is our "Intro to Programming in C#" book.
In it you will learn the basics of C# programming.
You will find out how nice C# is.
```

We can implement the counting as follows: will read the file line by line and each time we meet the desired word inside the last read line, we will increase the value of a variable (counter). We will process the possible exceptional situations to enable users to receive adequate information in case of errors. Here is a sample implementation:

CountingWordOccurrences.cs

```
static void Main()
{
   string fileName = @"..\..\sample.txt";
   string word = "C#";
   try
   {
      StreamReader reader = new StreamReader(fileName);
      using (reader)
      {
         int occurrences = 0;
         string line = reader.ReadLine();
         while (line != null)
         {
            int index = line.IndexOf(word);
            while (index != -1)
            {
               occurrences++;
               index = line.IndexOf(word, (index + 1));
            }
            line = reader.ReadLine();
         }
         Console.WriteLine(
            "The word {0} occurs {1} times.", word, occurrences);
      }
   }
   catch (FileNotFoundException)
```

```
    {
        Console.Error.WriteLine(
            "Can not find file {0}.", fileName);
    }
    catch (IOException)
    {
        Console.Error.WriteLine(
            "Cannot read the file {0}.", fileName);
    }
}
```

For simplicity of the example, the word we seek is hardcoded. You can implement the program to search a word entered by the user.

You can see that the example is not very different from the previous ones. We initialize the variables outside of the **try-catch** block. Again, we use a **while**-loop to read the lines of the text file one by one. Inside its body, there is another while-loop, which counts how many times the searched word occurs in the given line, and then increases the number of occurrences. This is done using the method **IndexOf(…)** of the class **String** (remember what it does in case you have forgotten). We do not forget to ensure the closing of the **StreamReader** object using the **using** structure. All that remains is to print the results on to the console.

For our example, the result is the following:

```
The word C# occurs 3 times.
```

Editing a Subtitles File – Example

Now we will look at a more complex example, in which we at the same time read from a file and record to another. This program **fixes a subtitles file for a movie**.

Our goal will be to read a file with subtitles, that are incorrect and do not appear at the right time, and to shift the times in an appropriate manner, so that they can appear correctly. One such file generally contains the time of the on-screen duration and the text, that should appear in the defined time interval. Here is how **typical subtitles files** look like:

GORA.sub

```
{1029}{1122}{Y:i}Captain, systems are|at the ready.
{1123}{1270}{Y:i}The preassure is stable.|Prepare for landing.
{1343}{1468}{Y:i}Please,fasten your seatbelts|and take your
places.
{1509}{1610}{Y:i}Coordinates 5.6|- Five, Five, Six, dot com.
```

```
{1632}{1718}{Y:i}Where did the coordinates|go to?
{1756}{1820}Commander Logar,|everyone is speaking in English.
{1821}{1938}Can't we switch|to Turkish from the beginning?
{1942}{1992}Yes, we can!
{3104}{3228}{Y:b}G.O.R.A.|a movie about the cosmos
…
```

StarWars.sub

```
{1029}{1122}{Y:i}I'll never join you.
{1123}{1270}{Y:i}If you only knew | the power of the dark side.
{1343}{1468}{Y:i}Obi One never told you what happened to your
father!
{1509}{1610}{Y:i}He told me enough! | He told me you killed him.
{1632}{1718}{Y:i}No... I am your father!
{1756}{1820}(dramatic music playing)...
{1821}{1938}No, no that's not true... | That's impossible!
{1942}{1992}Search your feelings,| you know it's true.
{3104}{3228}{Y:b}Nooo…
…
```

To **fix the subtitles**, we usually just need to make an adjustment in the time for displaying the subtitles. Such an adjustment may be offsetting the start / end time for each subtitle (by addition or subtraction of a constant) or changing the speed (multiplying by a factor, say 1.05).

Here is sample code that can implement such a program:

FixingSubtitles.cs

```
using System;
using System.IO;

class FixingSubtitles
{
   const double COEFFICIENT = 1.05;
   const int ADDITION = 5000;
   const string INPUT_FILE = @"..\..\source.sub";
   const string OUTPUT_FILE = @"..\..\fixed.sub";

   static void Main()
   {
      try
      {
```

```
      // Create reader
      StreamReader streamReader = new StreamReader(INPUT_FILE);

      // Create writer
      StreamWriter streamWriter =
        new StreamWriter(OUTPUT_FILE, false);

      using (streamReader)
      {
        using (streamWriter)
        {
          string line;
          while ((line = streamReader.ReadLine()) != null)
          {
            streamWriter.WriteLine(FixLine(line));
          }
        }
      }
    }
    catch (IOException exc)
    {
      Console.WriteLine("Error: {0}.", exc.Message);
    }
  }

  static string FixLine(string line)
  {
    // Find closing brace
    int bracketFromIndex = line.IndexOf('}');

    // Extract 'from' time
    string fromTime = line.Substring(1, bracketFromIndex - 1);

    // Calculate new 'from' time
    int newFromTime = (int) (Convert.ToInt32(fromTime) *
      COEFFICIENT + ADDITION);

    // Find the following closing brace
    int bracketToIndex = line.IndexOf('}',
      bracketFromIndex + 1);

    // Extract 'to' time
    string toTime = line.Substring(bracketFromIndex + 2,
      bracketToIndex - bracketFromIndex - 2);
```

```
        // Calculate new 'to' time
        int newToTime = (int) (Convert.ToInt32(toTime) *
          COEFFICIENT + ADDITION);

        // Create a new line using the new 'from' and 'to' times
        string fixedLine = "{" + newFromTime + "}" + "{" +
          newToTime + "}" + line.Substring(bracketToIndex + 1);

        return fixedLine;
    }
}
```

Again, we use the already familiar method for **reading a file line by line**. The difference this time is, that in the body of the loop, we write every line of the file with already corrected subtitles, after we have fixed them with the method **`FixLine(string)`** (this method is not the subject of our discussion, since it can be implemented in many different ways depending on what you want to adjust). Because we use the **`using`** block for both files, we can guarantee that they will be closed even if an exception occurs during processing (this may happen, for example if one of the lines in the file is not in the expected format).

Exercises

1. Write a program that reads a text file and **prints its odd lines** on the console.
2. Write a program that **joins two text files** and records the results in a third file.
3. Write a program that reads the contents of a text file and **inserts the line numbers** at the beginning of each line, then rewrites the file contents.
4. Write a program that **compares two text files** with the same number of rows line by line and prints the number of equal and the number of different lines.
5. Write a program that reads a square matrix of integers from a file and **finds the sub-matrix with size 2 × 2 that has the maximal sum** and writes this sum to a separate text file. The first line of input file contains the size of the recorded matrix (N). The next N lines contain N integers separated by spaces.

   Sample input file:

```
4
2 3 3 4
```

```
0 2 3 4
3 7 1 2
4 3 3 2
```

Sample output: 17.

6. Write a program that **reads a list of names** from a text file, arranges them in **alphabetical order**, and writes them to another file. The lines are written one per row.
7. Write a program that **replaces every occurrence of the substring** "**start**" with "**finish**" in a text file. Can you rewrite the program to replace whole words only? Does the program work for large files (e.g. 800 MB)?
8. Write the previous program so that it changes only the **whole words** (not parts of the word).
9. Write a program that **deletes all the odd lines** of a text file.
10. Write a program that extracts from an XML file the **text only** (without the tags). Sample input file:

```
<?xml version="1.0"><student><name>Peter</name>
<age>21</age><interests count="3"><interest>
Games</interest><interest>C#</interest><interest>
Java</interest></interests></student>
```

Sample output:

```
Peter
21
Games
C#
Java
```

11. Write a program that **deletes all words** that begin with the word "**test**". The words will contain only the following chars: 0…9, a…z, A…Z.
12. A text file **words.txt** is given, containing a list of words, one per line. Write a program that **deletes in the file text.txt all the words that occur in the other file**. Catch and handle all possible exceptions.
13. Write a program that **reads a list of words** from a file called **words.txt**, **counts how many times each of these words is found in another file text.txt**, and records the results in a third file – **result.txt**, but before that, sorts them by the number of occurrences in descending order. Handle all possible exceptions.

Solutions and Guidelines

1. Use the **examples** discussed in **this chapter**. Use the **`using`** structure to ensure proper closing of the input and the resulting stream.
2. You will have to first **read the input file line by line** and save it in the resulting file in **overwrite** mode. Then you must open the second input file and save it line by line in the result file in append mode. To create a **`StreamWriter`** in overwrite / use mode use the appropriate constructor (find it in MSDN).

   An alternative way is to read both files in a **`string`** with **`ReadToEnd()`**, store them in memory and save them in the resulting file. This approach does not work for large files (the likes of several gigabytes).
3. Follow the **examples in this chapter**. Think of how you would solve the task if the file were large (several gigabytes).
4. Follow the **examples in this chapter**. You will have to open both files simultaneously and read them line by line in a loop. If you reach the end of the (read null) file, that does not match the other's, that means that both files have different number of rows and an exception should be thrown.
5. Read the first line of the file and **create a matrix** with the specified size. After that read the following lines, line by line and separate the numbers. Then save them in the corresponding (row, column) in the matrix. Finally, find the sub-matrix using **two nested loops**.
6. Write each read name in a list (**`List<string>`**), then sort it properly (look for information on the method **`Sort()`**) and then print it in the result file.
7. Read the file **line by line** and use the methods of the class **`String`**. If you load the entire file into memory, instead of reading it line by line, problems when loading large files might occur.
8. For every occurrence of "start", check if that is the whole word or just a part of it.
9. Look at the examples in this chapter.
10. Read the input file **character by character**. When you encounter a "**`<`**", then this is a **starting tag**, but when you encounter a "**`>`**", that means a **closing tag**. All characters you encounter outside of the tags build up the text that must be extracted. You can store it in **`StringBuilder`** and print its contents when you encounter "**`<`**" or reach the end of the file.
11. Read the file **line by line** and **replace** words that start with "**`test`**" with an empty string. Use **`Regex.Replace(…)`** with an appropriate regular expression. Alternatively, you can search in the line the substring "**`test`**" and every time you find it, get all of its neighboring letters to the left and right. This way you find the word in which the string "**`test`**" is part of and you can delete it if it begins with "**`test`**".

12. The task is **similar to the previous one**. You can read the text **line by line** and replace each of the given words with an empty string. Test whether your program properly handles exceptions by simulating different scenarios (e.g. no file, lack of rights for reading and writing, etc.).

13. Create a **hash table with keys – the words from `words.txt`** and **value number of occurrences** of each word (**`Dictionary<string, int>`**). Firstly, save to the hash table that all words are found 0 times. Then read the file line by line and split each line into words. Check whether each obtained word can be found in the hash table, and if so, add 1 to the number of occurrences. Finally, save all the words and their number of occurrences in an array of type **`KeyValuePair<string, int>`**. Sort the array with a suitable comparison function like so:

```
Array.Sort<KeyValuePair<string, int>>(
  arr, (a, b) => a.Value.CompareTo(b.Value));
```