37. Streams

We are now about to start the first chapter in the lecture about Input and Output (IO). Traditionally, IO deals with transfer of data to/from secondary storage, most notably disks. IO also covers the transmission of data to/from networks.

In this and the following chapters we will study the classes that are related to input and output. This includes file and directory classes. At the abstract level, the `Stream` class is the most important class in the IO landscape. Therefore we choose to start the IO story with an exploration of streams, and an understanding of the `Stream` class in C#. This includes several `Stream` subclasses and several client classes of `Stream`. The clients we have in mind are the so-called reader and writer classes.

37.1. The Stream Concept

Lecture 10 - slide 2

A stream is an abstract concept. A stream is a *connection* between a program and a storage/network. Essentially, we can read data from the stream into a program, or we can write data from a program to the stream. This understanding of a stream is illustrated in Figure 37.1.

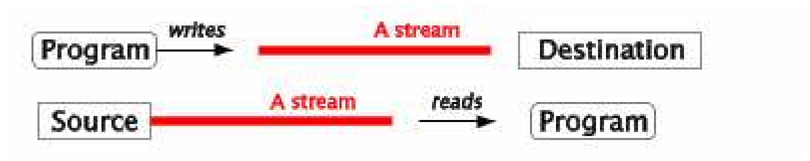

Figure 37.1 *Reading from and writing to a stream*

A *stream* is a flow of data from a program to a backing store, or from a backing store to a program

The program can either *write* to a stream, or *read* from a stream.

Stream and stream processing includes the following:

- Reading from or writing to files in secondary memory (disk)
- Reading from or writing to primary memory (RAM)
- Connection to the Internet
- Socket connection between two programs

The second item (reading and writing to/from primary memory) seems to be special compared to the others. Sometimes it may be attractive to have files in primary memory, and therefore it is natural that we should be able to use stream operation to access such files as well. In other situations, we wish to use internal data structures as sources or destinations of streams. It is, for instance, typical that we wish to read and write data from/to strings. We will see how this can be done in Section 37.14.

37.2. The abstract class Stream in C#

Lecture 10 - slide 3

The `Stream` class in C# is an abstract class (see Section 30.1). It belongs to the `System.IO` namespace, together with a lot other IO related types. The abstract `Stream` class provides a generic view on different kinds of sources and destinations, and it isolates client classes from the operating system details of these.

The `Stream` class supports both synchronous and asynchronous IO operations. Client classes that invoke a synchronous operation wait until the operation is completed before they can initiate other operations or actions. Use of a synchronous operation is not a problem if the operation is fast. Many IO operations on secondary storage are, however, very slow seen relative to the speed of the operations on primary storage. Therefore it may in some circumstances be attractive to initiate an IO operation, do something else, and consult the result of the IO operation at a later point in time. In order to provide for this, the `Stream` class supports the asynchronous IO operations `BeginRead` and `BeginWrite`. In the current version of the material we do not cover the asynchronous operations.

Let us now look at the most important operations on streams. The *`italic names`* refer to abstract methods. The abstract methods will be implemented in non-abstract subclasses of `Stream`.

- `int` *`Read`* `(byte[] buf, int pos, int len)`
- `int ReadByte()`
- `void` *`Write`* `(byte[] buf, int pos, int len)`
- `void WriteByte(byte b)`
- `bool` *`CanRead`*
- `bool` *`CanWrite`*
- `bool` *`CanSeek`*
- `long` *`Length`*
- `void` *`Seek`* `(long offset, SeekOrigin org)`
- `void` *`Flush`* `()`
- `void Close()`

In order to use `Read` you should allocate a byte array and pass (a reference to) this array as the first parameter of `Read`. The call `Read(buf, p, lgt)` reads *at most* `lgt` bytes, and stores them in `buf[p]` ... `buf[p+lgt-1]`. `Read` returns the actual number of characters read, which can be less than `lgt`.

`Write` works in a similar way. We assume that a number of bytes are stored in an existing byte array called `buf`. The call `Write(buf, p, lgt)` writes `lgt` bytes, `buf[p]` ... `buf[p+lgt-1]`, to the stream.

As you can see, only `ReadByte` and `WriteByte` are non-abstract methods. `ReadByte` returns the integer value of the byte being read, or -1 in case that the end of the stream has bee encountered. The two operations `ReadByte` and `WriteByte` rely on `Read` and `Write`. Internally, `ReadByte` calls `Read` on a one-byte array, it accesses this byte, and it returns this byte. `WriteByte` works in a similar way. Based on these informations, it is not surprising that it is recommended to redefine `ReadByte` and `WriteByte` in specialized `Stream` classes. The default implementations of `ReadByte` and `WriteByte` are simply too inefficient. The redefinitions should be able to profit from internal buffering.

The explanations of `Read` in relation to `ReadByte` (and `Write` in relation to `WriteByte`) may seem a little surprising. Why not have `ReadByte` as an abstract method, and `Read` as a non-abstract method, which once and for all is implemented in class `Stream` by multiple calls of `ReadByte`? Such a design seems to be ideal:

The task of implementing `ReadByte` in subclasses is easy, and no subclass should ever need to implement `Read`. The reason behind the actual design of the abstract `Stream` class is - of course - *efficiency*. The basic read and write primitives of streams should provide for efficient reading and writing. It is typically inefficient to read a single byte from a file. On many types of hardware (such as harddisks) we always read many bytes at a time. The design of the read and write operations of stream take advantage of this observation.

It is not possible to read, write, and seek in all streams. Therefore it is possible to query a stream for its actual capabilities. The boolean operations (properties) `CanRead`, `CanWrite`, `CanSeek` are used for such querying.

The static field **`Null`** represents a stream without a backing store.

`Null` is a public static field of type `Stream` in the abstract class `Stream`. If you, for some reason, wish to discard the data that you write, you can write it to `Stream.Null`. You can also read from `Stream.Null`; This will always give zero as result, however.

37.3. Subclasses of class Stream

Lecture 10 - slide 5

The abstract class `Stream` is the superclass of a number of non-abstract classes. Below we list the most important of these. Like the class `Stream`, many of the subclasses of `Stream` belong to the `System.IO` namespace.

- `System.IO.FileStream`
  - Provides a stream backed by a file from the operating system
- `System.IO.BufferedStream`
  - Encapsulates buffering around another stream
- `System.IO.MemoryStream`
  - Provides a stream backed by RAM memory
- `System.Net.Sockets.NetworkStream`
  - Encapsulates a socket connection as a stream
- `System.IO.Compression.GZipStream`
  - Provides stream access to compressed data
- `System.Security.Cryptography.CryptoStream`
  - `Write` encrypts and `Read` decrypts
- *And others...*

We show example uses of class `FileStream` in Section 37.4 and Section 37.6. Please notice, however, that file IO is typically handled through one of the reader and writer classes, which behind the scene delegates the work to a `Stream` class. We have a lot more to say about the reader and writer classes later in this material. Section 37.9 will supply you with an overview of the reader and writer classes in C#.

The class `BufferedStream` is intended to be used as a so-called decorator of another stream class. In Section 40.1 we discuss the ***Decorator*** design pattern. The concrete example of ***Decorator***, which we will discuss in Section 40.2, involves compressed streams. Notice that it is not relevant to use buffering on `FileStream`, because it natively makes use of buffering.

37.4. Example: Filestreams

Lecture 10 - slide 6

`FileStream` IO, as illustrated by the examples in this section, is used for *binary input and output*. It means that the `FileStream` operations transfer raw chuncks of bits between the program and the file. The bits are not interpreted. As a contrast, the reader and writer classes introduced in Section 37.9 interpret and transforms the raw binary data to values in C# types.

Let us show a couple of very simple programs that write to and read from filestreams. Figure 37.1 writes bytes corresponding to the three characters 'O', 'O', and 'P' to the file `myFile.bin`. Notice that we do not write characters, but numbers that belong to the simple type `byte`. The file opening is done via construction of the `FileStream` object in `Create` mode. `Create` is a value in the enumeration type `FileMode` in the namespace `System.IO`. File closing is done by the `Close` method.

```
using System.IO;

class ReadProg {
  static void Main() {
    Stream s = new FileStream("myFile.bin", FileMode.Create);
    s.WriteByte(79);  // O    01001111
    s.WriteByte(79);  // O    01001111
    s.WriteByte(80);  // P    01010000
    s.Close();
  }
}
```

Program 37.1 *A program that writes bytes corresponding to 'O' 'O' 'P' to a file stream.*

After having executed the program in Figure 37.1 the file `myFile.bin` exists. Program 37.2 reads it. We create a `FileStream` object in `Open` mode, and we read the individual bytes with use of the `ReadByte` method. In line 11 and 12 we illustrate what happens if we read beyond the end of the file. We see that `ReadByte` in that case returns -1. The number -1 is not a value in type `byte`, which supports the range 0..255. Therefore the type of the value returned by `ReadByte` is `int`.

```
1  using System;
2  using System.IO;
3
4  class WriteProg {
5    static void Main() {
6      Stream s = new FileStream("myFile.bin", FileMode.Open);
7      int i, j, k, m, n;
8      i = s.ReadByte();  // O   79  01001111
9      j = s.ReadByte();  // O   79  01001111
10     k = s.ReadByte();  // P   80  01010000
11     m = s.ReadByte();  // -1  EOF
12     n = s.ReadByte();  // -1  EOF
13
14     Console.WriteLine("{0} {1} {2} {3} {4}", i, j, k, m, n);
15     s.Close();
16   }
17 }
```

Program 37.2 *A program that reads the written file.*

37.5. The using control structure

Lecture 10 - slide 7

The simple file reading and writing examples in Section 37.4 show that file opening (in terms of creating the `FileStream` object) and file closing (in terms of sending a `Close` message to the stream) appear in pairs. This inspires a new control structure which ensures that the file always is closed when we are done with it. The syntax of the **using** construct is explained below.

```
using (type variable = initializer)
  body
```

Syntax 37.1 *The syntax of the using statement C#*

The meaning (semantics) of the using construct is the following:

- In the scope of **`using`**, bind *`variable`* to the value of *`initializer`*
- The *`type`* must implement the interface **`IDisposable`**
- Execute *body* with the established name binding
- At the end of *body* do *variable*`.Dispose`
    - The `Dispose` methods in the subclasses of `Stream` call `Close`

We encountered the interface `IDisposable` when we studied the interfaces in the C# libraries, see Section 31.4. The interface `IDisposable` prescribes a single method, `Dispose`, which in general is supposed to release resources. The abstract class `Stream` implements `IDisposable`, and the `Dispose` method of class `Stream` calls the Stream `Close` method.

Program 37.3 is a reimplementation of Program 37.1 that illustrates the **using** construct. Notice that we do not explicitly call `Close` in Program 37.3.

```
using System.IO;

class ReadProg {
  static void Main() {
    using(Stream s = new FileStream("myFile.txt", FileMode.Create)){
      s.WriteByte(79);  // O   01001111
      s.WriteByte(79);  // O   01001111
      s.WriteByte(80);  // P   01010000
    }
  }
}
```

Program 37.3 *The simple write-program programmed with 'using'.*

The following fragment shows what is actually covered by a **using** construct. Most important, a **try-finally** construct is involved, see Section 36.9. The use of **try-finally** implies that `Dispose` will be called independent of the way we leave `body`. Even if we attempt to exit `body` with a jump or via an exception, `Dispose` will be called.

```
// The using statement ...

  using (type variable = initializer)
    body

// ... is equivalent to the following try-finally statement

  {type variable = initializer;
    try {
      body
    }
    finally {
      if (variable != null)
         ((IDisposable)variable).Dispose();
    }
  }
```

Program 37.4 *The control structure 'using' defined by 'try-finally'.*

37.6. More FileStream Examples

Lecture 10 - slide 8

We will show yet another simple example of `FileStreams`, namely a static method that copies one file to another.

```
using System;
using System.IO;

public class CopyApp {

  public static void Main(string[] args) {
    FileCopy(args[0], args[1]);
  }

  public static void FileCopy(string fromFile, string toFile){
    try{
      using(FileStream fromStream =
                       new FileStream(fromFile, FileMode.Open)){
        using(FileStream toStream  =
                         new FileStream(toFile, FileMode.Create)){
          int c;

          do{
            c = fromStream.ReadByte();
            if(c != -1) toStream.WriteByte((byte)c);
          } while (c != -1);
        }
      }
    }
    catch(FileNotFoundException e){
      Console.WriteLine("File {0} not found: ", e.FileName);
      throw;
    }
    catch(Exception){
      Console.WriteLine("Other file copy exception");
      throw;
    }
 }

}
```

Program 37.5 *A FileCopy method in a source file copy-file.cs - uses two FileStreams.*

Exercise 10.1. *A variant of the file copy program*

The purpose of this exercise is to train the use of the `Read` method in class `Stream`, and subclasses of class `Stream`.

Write a variant of the file copy program. Your program should copy the entire file into a byte array. Instead of the method `ReadByte` you should use the `Read` method, which reads a number of bytes into a byte array. (Please take careful look at the documentation of `Read` in class `FileStream` before you proceed). After this, write out the byte array to standard output such that you can make sure that the file is correctly read.

Are you able to read the entire file with a single call to `Read`? Or do you prefer to read chunks of a certain (maximum) size?

37.7. The class Encoding

Lecture 10 - slide 10

Before we study the reader and writer classes we will clarify one important topic, namely *encodings*.

The problem is that a byte (as represented by a value of type `byte`) and a character (as represented as value of type `char`) are two different things. In the old days they were basically the same, or it was at least straightforward to convert one to the other. In old days there were at most 256 different characters available at a given point in time (corresponding to a straightforward encoding of a single character in a single byte). Today, the datatype `char` should be able to represent a wide variety of different characters that belong to different alphabets in different cultures. We still need to represent a character by means of a number of bytes, because a byte is a fundamental unit in most software and in most digital hardware.

As a naive approach, we could go for the following solution:

> We want to be able to represent a maximum of, say, 200000 different characters. For this purpose we need $\log_2(200000)$ bits, which is 18 bits. If we operate in units if 8 bits (= one byte) we see that we need at least 3 bytes per characters. Most likely, we will go for 4 bytes per character, because it fits much better with the word length of most computers. Thus, the byte size of a text will now be four times the size of an ASCII text. This is not acceptable because it would bloat the representation of text files on secondary disk storage.

As of 2007, the Unicode standard defines more than 100000 different characters. Unicode organizes characters in a number of *planes* of up to 2^{16} (= 65536) characters. The Basic Multilingual Plane - BMP - contains the most common characters.

Encodings are invented to solve the problem that we have outlined above. An encoding is a mapping between values of type character (a *code point* number between 0 and 200000 in our case) to a sequence of bytes. The naive approach outlined above represents a simple encoding, in which we need 4 bytes even for the original ASCII characters. It is attractive, however, if characters in the original, 7-bit ASCII alphabet can be encoded in a single byte. The price of that may very well be that some rarely used characters will need considerable more bytes for their encoding.

Let us remind ourselves that in C#, the type `char` is represented as 16 bit entities (Unicode characters) and that a `string` is a sequence of values of type `char`. We have already touched on this in Section 6.1. At the time Unicode was designed, it was hypothesized that 16 bits was enough to to represent all characters in the world. As mentione above, this turned out not to be true. Therefore the type `char` in C# is not big enough to hold all Unicode characters. The remedy is to use multiple `char` values for representation of a single Unicode character. We see that history repeats itself...

An encoding is a mapping between characters/strings and byte arrays

An object of class **`System.Text.Encoding`** represents knowledge about a particular character encoding

Let us now review the operations in class `Encoding:`, which is located in the namespace `System.Text` :

- **`byte[] GetBytes(string)`** *Instance method*
- **`byte[] GetBytes(char[])`** *Instance method*
  - Encodes a string/char array to a byte array relative to the current encoding
- **`char[] GetChars(byte[])`** *Instance method*
  - Decodes a byte array to a char array relative to the current encoding
- **`byte[] Convert(Encoding, Encoding, byte[])`** *Static method*
  - Converts a byte array from one encoding (first parameter) to another encoding (second parameter)

The method `GetBytes` implements the encoding in the direction of characters to byte sequences. In concrete terms, the method `GetBytes` transforms a `String` or an array of `char`s to a `byte` array.

The inverse method, `GetChars` converts an array of bytes to the corresponding array of characters. On a given string `str` and for a given encoding `e` `e.GetChars(e.GetBytes(str))` corresponds to `str`.

For given encodings `e1` and `e2`, and for some given byte array `ba` supposed to be encoded in `e1`, `Convert(e1,e2,ba)` is equivalent to `e2.GetBytes(e1.GetChars(ba))`.

37.8. Sample use of class Encoding

Lecture 10 - slide 11

Now that we understand the idea behind encodings, let us play a little with them. In Program 37.6 we make a number different encodings, and we convert a given string to some of these encodings. We explain the details after the program.

```
1  using System;
2  using System.Text;
3
4  /* Adapted from an example provided by Microsoft */
5  class ConvertExampleClass{
6    public static void Main(){
7      string unicodeStr =          // "A æ u å æ ø i æ å"
8          "A \u00E6 u \u00E5 \u00E6 \u00F8 i \u00E6 \u00E5";
9
10     // Different encodings.
11     Encoding ascii = Encoding.ASCII,
12              unicode = Encoding.Unicode,
13              utf8 = Encoding.UTF8,
14              isoLatin1 = Encoding.GetEncoding("iso-8859-1");
15
16     // Encodes the characters in a string to a byte array:
17     byte[] unicodeBytes = unicode.GetBytes(unicodeStr),
18            asciiBytes =   ascii.GetBytes(unicodeStr),
19            utf8Bytes =   utf8.GetBytes(unicodeStr),
20            isoLatin1Bytes =   utf8.GetBytes(unicodeStr);
21
22     // Convert from byte array in unicode to byte array in utf8:
23     byte[] utf8BytesFromUnicode =
24       Encoding.Convert(unicode, utf8, unicodeBytes);
25
26     // Convert from byte array in utf8 to byte array in ascii:
27     byte[] asciiBytesFromUtf8 =
28       Encoding.Convert(utf8, ascii, utf8Bytes);
29
30     // Decodes the bytes in byte arrays to a char array:
31     char[] utf8Chars = utf8.GetChars(utf8BytesFromUnicode);
32     char[] asciiChars = ascii.GetChars(asciiBytesFromUtf8);
33
34     // Convert char[] to string:
35     string utf8String = new string(utf8Chars),
36            asciiString = new String(asciiChars);
37
38     // Display the strings created before and after the conversion.
39     Console.WriteLine("Original string: {0}", unicodeStr);
40     Console.WriteLine("String via UTF-8: {0}", utf8String);
41
42     Console.WriteLine("Original string: {0}", unicodeStr);
43     Console.WriteLine("ASCII converted string: {0}", asciiString);
44   }
45 }
```

Program 37.6   *Sample encodings, conversions, and decodings of a string of Danish characters.*

In line 7 we declare a sample string, `unicodeStr`, which we initialize to a string with plenty of national Danish characters. We notate the string with escape notation `\u`*dddd* where *d* is a hexadecimal digit. We could, as well, have used the string constant in the comment at the end of line 7.

In line 11-14 we make a number of instances of class `Encoding`. Some common `Encoding` objects can be accessed conveniently via static properties of class `Encoding`. The UTF-8 encoding can in that way be accessed with `Encoding.UTF8`. The static method `GetEncoding` accesses an encoding via the name of the encoding. (In order to get access to all supported encodings, the static method `GetEncodings` (plural) is useful). The ISO Latin 1 encoding is accessed via use with use of `GetEncoding` in line 14.

In line 17-20 we convert the string `unicodeStr` to `byte` arrays in different encodings. For this purpose we use the instance method `GetBytes`.

Next, in line 22-28, we show how to use the static method `Convert` to convert a `byte` array in one encoding to a `byte` array in another encoding.

In line 30-32 it is shown how to convert byte arrays in a particular encoding to a char array. It is done by the instance method `GetChars`. We most probably wish to obtain a string instead of a `char` array. For that purpose we just use an appropriate `String` constructor, as shown in line 34-36.

In line 38-43 we display the values of `utf8String` and `asciiString`, and for comparison we also print the original `unicodeStr`. The printed result is shown in Listing 37.7. It is not surprising that the national Danish characters cannot be represented in the ASCII character set. The Danish characters are (ambiguously) translated to '?'.

```
Original string: A æ u å æ ø i æ å
String via UTF-8: A æ u å æ ø i æ å
Original string: A æ u å æ ø i æ å
ASCII converted string: A ? u ? ? ? i ? ?
```

Listing 37.7 *Output from the Encoding program.*

Exercise 10.2. *Finding the encoding of a given text file*

Make a UTF-8 text file with some words in Danish. Be sure to use plenty of special Danish characters. You may consider to write a simple C# program to create the file. You may also create the text file in another way.

In this exercise you should avoid writing a byte order mark (BOM) in your UTF-8 text file. (A BOM in the UTF-8 text file may short circuit the decoding we are asking for later in the exercise). One way to avoid the BOM is to denote the UTF-8 encoding with `new UTF8Encoding()`, or equivalently `new UTF8Encoding(false)`. You may want to consult the constructors in class `UFT8Encoding` for more information.

Now write a C# program which systematically - in a loop - reads the text file six times with the following objects of type `Encoding:` ISO-8859-1, UTF-7, UTF-8, UTF-16 (Unicode), UTF32, and 7 bits ASCII.

More concretely, I suggest you make a list of six encoding objects. For each encoding, open a `TextReader` and read the entire file (with `ReadToEnd`, for instance) with the current encoding. Echo the characters, which you read, to standard output.

You should be able to recognize the correct, matching encoding (UTF-8) when you see it.

37.9. Readers and Writers in C#

Lecture 10 - slide 9

In the rest of this chapter we will explore a family of so-called reader and writer classes. In most practical cases one or more of these classes are used for IO purposes instead of a `Stream` subclass, see Section 37.2.

Table 37.1 provides an overview of the reader and writer classes. In the horizontal dimension we have input (readers) and output (writers). In the vertical dimension we distinguish between binary (bits structured as bytes) and text (char/string) IO.

	Input	Output

Text	*TextReader*
	  StreamReader
	  StringReader
		TextWriter
		  StreamWriter
		  StringWriter

Binary	BinaryReader	BinaryWriter

Table 37.1 *An overview of Reader and Writer classes*

The class `Stream` and its subclasses are oriented towards input and output of bytes. In contrast, the reader and writer classes are able to deal with input and output of characters (values of type char) and values of other simple types. Thus, the reader and writer classes operate at a higher level of abstraction than the stream classes.

In Section 37.3 we listed some important subclasses of class `Stream`. We will now discuss how the reader and writer classes in Table 37.1 are related to the stream classes. None of the classes in Table 37.1 inherit from class `Stream`. Rather, they *delegate* part of their work to a `Stream` class. Thus, the reader and writer classes aggregate (***have a***) `Stream` class together with other pieces of data. The class `StreamReader`, `StreamWriter`, `BinaryReader`, and `BinaryWriter` all have constructors that take a `Stream` class as parameter. In that way, it is possible to build such readers and writes on a `Stream` class.

`TextReader` and `TextWriter` in Table 37.1 are abstract classes. Their subclasses `StringReader` and `StringWriter` are build on strings rather than on streams. We have more to say about `StringReader` and `StringWriter` in Section 37.14.

In the following sections we will rather systematically describe the reader and writer classes in Table 37.1, and we will show examples of their use.

37.10. The class TextWriter

Lecture 10 - slide 12

In this section we discuss the abstract class `TextWriter`, and not least its non-abstract subclass `StreamWriter`. We cover the sibling classes `StringWriter` and `StringReader` in Section 37.14.

Most important, class `TextWriter` supports writing of text - characters and strings - via a chosen encoding. Encodings were discussed in Section 37.7. With use of class `TextWriter` it is also possible to write textual representations of simple types, such as `int` and `double`.

We illustrate the use of class `StreamWriter` in Program 37.8. Recall from Table 37.1 that `StreamWriter` is a non-abstract subclass of class `TextWriter`.

In Program 37.8 we write `str` and `strEquiv` (in line 9-10) to three different files. Both strings are identical, they contain a lot of Danish letters, but they are notated differently. It is the same string that we used in Program 37.6 for illustration of encodings. For each of the files we use a particular encoding (see Section 37.7). Notice that we in line 12, 16 and 20 use a `StreamWriter` constructor that takes a `Stream` and an encoding as parameters. There a six other constructors to chose from (see below). In line 24-26 we write the two strings to each of the three files. Try out the program, and read the three text files with your favorite text editor. Depending of the capabilities of your editor, you may or may not be able to read them all.

```
using System;
using System.IO;
using System.Text;

public class TextWriterProg{

  public static void Main(){
    string str =      "A æ u å æ ø i æ å",
           strEquiv = "A \u00E6 u \u00E5 \u00E6 \u00F8 i \u00E6 \u00E5";

    TextWriter
      tw1 = new StreamWriter(                        // Iso-Latin-1
             new FileStream("f-iso.txt", FileMode.Create),
             Encoding.GetEncoding("iso-8859-1")),

      tw2 = new StreamWriter(                        // UTF-8
             new FileStream("f-utf8.txt", FileMode.Create),
             new UTF8Encoding()),

      tw3 = new StreamWriter(                        // UTF-16
             new FileStream("f-utf16.txt", FileMode.Create),
             new UnicodeEncoding());

    tw1.WriteLine(str);     tw1.WriteLine(strEquiv);
    tw2.WriteLine(str);     tw2.WriteLine(strEquiv);
    tw3.WriteLine(str);     tw3.WriteLine(strEquiv);

    tw1.Close();
    tw2.Close();
    tw3.Close();
  }

}
```

Program 37.8 *Writing a text string using three different encodings with StreamWriters.*

You may wonder if knowledge about the applied encoding is somehow represented in the text file. The first few bytes in a text file created from a `TextWriter` may contain some information about the encoding. `StreamWriter` calls `Encoding.GetPreamble()` in order to get a byte array that represents knowledge about the encoding. This byte array is written in the beginning of the text file. This preamble is primarily used to determine the byte order of UTF-16 and UTF-32 encodings. (Two different byte orders are widely used on computers from different CPU manufacturers: Big-endian (most significant byte first) and little-endian (least significant byte first)). The preambles of the ASCII and the ISO Latin 1 encodings are empty.

The next program, shown in Program 37.9, first creates a `StreamWriter` on a given file path (a text string) "`simple-types.txt`". The default encoding is used. (The default encoding is system/culture dependent. It can be accessed with the static property `Encoding.Default`). By use of the heavily overloaded `Write` method it writes an integer, a double, a decimal, and a boolean to the file.

Next, from line 15-18, it writes a `Point` and a `Die` to a text file named "`non-simple-types.txt`". As expected, the `ToString` method is used on the `Point` and the `Die` objects. The contents of the two text files are shown in Listing 37.10 (only on web) and Listing 37.11 (only on web).

```
1 using System;
2 using System.IO;
3 
4 public class TextSimpleTypes{
5 
6   public static void Main(){

    using(TextWriter tw = new StreamWriter("simple-types.txt")){
      tw.Write(5);  tw.WriteLine();
      tw.Write(5.5);  tw.WriteLine();
      tw.Write(5555M); tw.WriteLine();
      tw.Write(5==6); tw.WriteLine();
    }

    using(TextWriter twnst = new StreamWriter("non-simple-types.txt")){
      twnst.Write(new Point(1,2)); twnst.WriteLine();
      twnst.Write(new Die(6)); twnst.WriteLine();
    }

  }
}
```

Program 37.9 *Writing values of simple types and objects of our own classes.*

The following items summarize the operations in class `StreamWriter`:

- 7 overloaded constructors
  - Parameters involved: File name, stream, encoding, buffer size
  - **`StreamWriter(String)`**
  - **`StreamWriter(Stream)`**
  - **`StreamWriter(Stream, Encoding)`**
  - *others*
- 17/18 overloaded **`Write`** / **`WriteLine`** operations
  - Chars, strings, simple types. Formatted output
- **`Encoding`**
  - A property that gets the encoding used for this `TextWriter`
- **`NewLine`**
  - A property that gets/sets the applied newline string of this `TextWriter`
- *others*

Exercise 10.3. *Die tossing - writing to text file*

Write a program that tosses a `Die` 1000 times, and writes the outcome of the tosses to a textfile. Use a `TextWriter` to accomplish the task.

Write another program that reads the text file. Report the number of ones, twos, threes, fours, fives, and sixes.

37.11. The class TextReader

Lecture 10 - slide 15

The class `TextReader` is an abstract class of which `StreamReader` is a non-abstract subclass. `StreamReader` is able to read characters from a byte stream relative to a given encoding. In most respects, the class `TextReader` is symmetric to class `TextWriter`. However, there are no `Read` counterparts to all the overloaded `Write` methods in `TextWriter`. We will come back to this observation below.

Program 37.12 is a program that reads the text that was produced by Program 37.8. In Program 37.12 we create three `TextReader` object. They are all based on file stream objects and encodings similar to the ones used in Program 37.8. From each `TextReader` we read the two strings that we wrote in Program 37.8. It is hardly surprising that we get six instances of the strange string "A æ u å æ ø i æ å". In line 19-21 they are all written to standard output via use of `Console.WriteLine`.

The last half part of Program 37.12 (from line 27) reads the three files as binary information (as raw bytes). The purpose of this reading is to exercise the *actual contents* of the three files. This is done by opening each of the files via `FileStream` objects, see Section 37.4. Recall that `FileStream` allows for binary reading (in terms of bytes) of a file. The function `StreamReport` (line 39-49) reads each byte of a given `FileStream`, and it prints these bytes on the console. The output in Listing 37.13 reveals - as expected - substantial differences between the actual, binary contents of the three files. Notice that the ISO Latin 1 file is the shortest, the UTF-8 file is in between, and the UTF-16 file is the longest.

```
1  using System;
2  using System.IO;
3  using System.Text;
4  
5  public class TextReaderProg{
6  
7    public static void Main(){
8  
9      TextReader tr1 = new StreamReader(
10                         new FileStream("f-iso.txt", FileMode.Open),
11                         Encoding.GetEncoding("iso-8859-1")),
12                tr2 = new StreamReader(
13                         new FileStream("f-utf8.txt", FileMode.Open),
14                         new UTF8Encoding()),
15                tr3 = new StreamReader(             // UTF-16
16                         new FileStream("f-utf16.txt", FileMode.Open),
17                         new UnicodeEncoding());
18  
19     Console.WriteLine(tr1.ReadLine());  Console.WriteLine(tr1.ReadLine());
20     Console.WriteLine(tr2.ReadLine());  Console.WriteLine(tr2.ReadLine());
21     Console.WriteLine(tr3.ReadLine());  Console.WriteLine(tr3.ReadLine());
22  
23     tr1.Close();
24     tr2.Close();
25     tr3.Close();
26  
27     // Raw reading of the files to control the contents at byte level
28     FileStream  fs1 = new FileStream("f-iso.txt", FileMode.Open),
29                 fs2 = new FileStream("f-utf8.txt", FileMode.Open),
30                 fs3 = new FileStream("f-utf16.txt", FileMode.Open);
31  
32     StreamReport(fs1, "Iso Latin 1");
33     StreamReport(fs2, "UTF-8");
34     StreamReport(fs3, "UTF-16");
35  
36     fs1.Close();
37     fs2.Close();
38     fs3.Close();
39   }
40  
41   public static void StreamReport(FileStream fs, string encoding){
42     Console.WriteLine();
43     Console.WriteLine(encoding);
44     int ch, i = 0;
45     do{
46       ch = fs.ReadByte();
47       if (ch != -1) Console.Write("{0,4}", ch);
```

```
      i++;
      if (i%10 == 0) Console.WriteLine();
    } while (ch != -1);
    Console.WriteLine();
  }

}
```

Program 37.12 *Reading back the text strings encoded in three different ways, with StreamReader.*

```
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å
A æ u å æ ø i æ å

Iso Latin 1
  65  32 230  32 117  32 229  32 230  32
 248  32 105  32 230  32 229  13  10  65
  32 230  32 117  32 229  32 230  32 248
  32 105  32 230  32 229  13  10

UTF-8
  65  32 195 166  32 117  32 195 165  32
 195 166  32 195 184  32 105  32 195 166
  32 195 165  13  10  65  32 195 166  32
 117  32 195 165  32 195 166  32 195 184
  32 105  32 195 166  32 195 165  13  10


UTF-16
 255 254  65   0  32   0 230   0  32   0
 117   0  32   0 229   0  32   0 230   0
  32   0 248   0  32   0 105   0  32   0
 230   0  32   0 229   0  13   0  10   0
  65   0  32   0 230   0  32   0 117   0
  32   0 229   0  32   0 230   0  32   0
 248   0  32   0 105   0  32   0 230   0
  32   0 229   0  13   0  10   0
```

Listing 37.13 *Output from the program that reads back the strings encoded in three different ways.*

Below, in Program 37.14, we show a program that reads the values from the file "`simple-types.txt`", as written by Program 37.9. Notice that we read a line at a time using the `ReadLine` method of `StreamReader`. `ReadLine` returns a string, which we parse by the static `Parse` methods in the structs `Int32`, `Double`, `Decimal`, and `Boolean` respectively. There are no dedicated methods in class `StreamReader` for reading the textual representations of integers, doubles, decimals, booleans, etc. The output of Program 37.14 is shown in Listing 37.15 (only on web).

```
using System;
using System.IO;

public class TextSimpleTypes{

  public static void Main(){

    using(TextReader twst = new StreamReader("simple-types.txt")){
      int i = Int32.Parse(twst.ReadLine());
      double d = Double.Parse(twst.ReadLine());
      decimal m = Decimal.Parse(twst.ReadLine());
      bool b = Boolean.Parse(twst.ReadLine());

      Console.WriteLine("{0} \n{1} \n{2} \n{3}", i, d, m, b);
    }

  }
}
```

Program 37.14 *A program that reads line of text and parses them to values of simple types.*

As we did for class `TextWriter` in Section 37.10 we summarize the operations in class `TextReader` below:

- 10 StreamReader constructors
  - Similar to the StreamWriter constructors
  - **`StreamReader(String)`**
  - **`StreamReader(Stream)`**
  - **`StreamReader(Stream, bool)`**
  - **`StreamReader(Stream, Encoding)`**
  - *others*
- **`int Read()`** Reads a single character. Returns -1 if at end of file
- **`int Read(char[], int, int)`** Returns the number of characters read
- **`int Peek()`**
- **`String ReadLine()`**
- **`String ReadToEnd()`**
- **`CurrentEncoding`**
  - A property that gets the encoding of this StreamReader

The method `Read` reads a single character; It returns -1 if the file is positioned at the end of the file. The `Read` method that accepts three parameters is similar to the `Stream` method of the same name, see Section 37.2. As such, it reads a number of characters into an already allocated char array (which is passed as the first parameter of `Read`). `Peek` reads the next available character without advancing the file position. You can use the method to look a little ahead of the actual reading. As we have seen, `ReadLine` reads characters until an end of line character is encountered. Similarly, `ReadToEnd` reads the rest of stream - from the current position until the end of the file - and returns it as a string. `ReadToEnd` is often convenient if you wish to get access to a text file as a (potentially large) text string.

37.12. The class BinaryWriter

Lecture 10 - slide 18

In this section we will study a writer class which produces binary data. As such, a binary writer is similar to a `FileStream` used in write access mode, see Section 37.4. The justification of `BinaryWriter` is, however, that it supports a heavily overloaded `Write` method just like the class `TextWriter` did. The `Write` methods can be applied on most simple data types. The `Write` methods of `BinaryWriter` produce binary data, not characters.

Encodings, see Section 37.7, played important roles for `TextReader` and `TextWriter`. Encodings only play a minimal role in `BinaryWriter`; Encodings are only used when we write characters to the binary file.

Below, in Program 37.16 we show a program similar to Program 37.9. We write four values of different simple types to a file with use of a `BinaryWriter`. In comments of the program we show the expected number of bytes to be written. With use of a `FileInfo` object (see Section 38.1) we check our expectations in line 18-19. The output of the program is 29, as expected.

```
1  using System;
2  using System.IO;
3  
4  public class BinaryWriteSimpleTypes{
5  
6    public static void Main(){
7      string fn = "simple-types.bin";
8  
9      using(BinaryWriter bw = 
10             new BinaryWriter(
11               new FileStream(fn, FileMode.Create))){
12       bw.Write(5);      // 4  bytes
13       bw.Write(5.5);    // 8  bytes
14       bw.Write(5555M);  // 16 bytes
15       bw.Write(5==6);   // 1  bytes
16     }
17 
18     FileInfo fi = new FileInfo(fn);
19     Console.WriteLine("Length of {0}: {1}", fn, fi.Length);
20 
21   }
22 }
```

Program 37.16 *Use of a BinaryWriter to write some values of simple types.*

The following operations are supplied by `BinaryWriter`:

- Two public constructors
  - **`BinaryWriter(Stream)`**
  - **`BinaryWriter(Stream, Encoding)`**
- 18 overloaded **`Write`** operations
  - One for each simple type
  - **`Write(char)`**, **`Write(char[])`**, and **`Write(char[], int, int)`** - use Encoding
  - **`Write(string)`** - use Encoding
  - **`Write(byte[])`** and **`Write(byte[], int, int)`**
- **`Seek(int offset, SeekOrigin origin)`**
- *others*

The second constructor allows for registration of an encoding, which is used if we write characters as binary data. The `Write` methods, which accepts an array as first parameter together with two integers as second and third parameters, write a section of the involved arrays.

Exercise 10.4. *Die tossing - writing to a binary file*

This exercise is a variant of the die tossing and file writing exercise based on text files.

Modify the program to use a `BinaryWriter` and a `BinaryReader`.

Take notice of the different sizes of the text file from the previous exercise and the binary file from this exercise. Explain your observations.

37.13. The class BinaryReader

Lecture 10 - slide 20

The class `BinaryReader` is the natural counterpart to `BinaryWriter`. Both of them deal with input from and output to binary data (in contrast to text in some given encoding).

The following program reads the binary file produced by Program 37.16. It produces the expected output, see Program 37.16 (only on web).

```
using System;
using System.IO;

public class BinaryReadSimpleTypes{

  public static void Main(){
    string fn = "simple-types.bin";

    using(BinaryReader br =
            new BinaryReader(
              new FileStream(fn, FileMode.Open))){

      int i = br.ReadInt32();
      double d = br.ReadDouble();
      decimal dm = br.ReadDecimal();
      bool b = br.ReadBoolean();

      Console.WriteLine("Integer i: {0}", i);
      Console.WriteLine("Double d: {0}", d);
      Console.WriteLine("Decimal dm: {0}", dm);
      Console.WriteLine("Boolean b: {0}", b);
    }

  }
}
```

Program 37.17 *Use of a BinaryReader to write the values written by means of the BinaryWriter.*

The following gives an overview of the operations in the class `BinaryReader`:

- Two public constructors
  - **`BinaryReader(Stream)`**
  - **`BinaryReader(Stream, Encoding)`**
- 15 individually name **`Read`***type* operations
  - **`ReadBoolean`**, **`ReadChar`**, **`ReadByte`**, **`ReadDouble`**, **`ReadDecimal`**, **`ReadInt16`**, ...
- Three overloaded **`Read`** operations
  - **`Read()`** and **`Read (char[] buffer, int index, int count)`** read characters - using Encoding
  - **`Read (bytes[] buffer, int index, int count)`** reads bytes

The most noteworthy observation is that there exist a large number of specifically named operations (such as `ReadInt32` and `ReadDouble`) through which it is possible to read the binary representations of values in simple types.

37.14. The classes StringReader and StringWriter

Lecture 10 - slide 22

`StringReader` is a non-abstract subclass of `TextReader`. Similarly, `StringWriter` is a non-abstract subclass of `TextWriter`. Table 37.1 gives you an overview of these classes.

The idea of `StringReader` is to use traditional stream/file input operations for string access, and to use traditional stream/file output operations for string mutation. Thus, relative to Figure 37.1 the source and destinations of reading and writing will be strings.

A `StringReader` can be constructed on a string. A `StringWriter`, however, cannot be constructed on a string, because strings are non-mutable in C#, see Section 6.4. Therefore a `StringWriter` object is constructed on an instance of `StringBuilder`.

In Program 37.19 we illustrate, in concrete terms, how to make a `StringWriter` on the `StringBuilder` referred by the variable `sb` (see line 9). In line 11-17 we iterate five times through the for loop, with increasing integer values in the variable `i`. In total, the textual representations of 20 simple values are written to the `StringBuilder` object. The content of the `StringBuilder` object is printed in line 19. The output of Program 37.19 is shown in Program 37.20 (only on web).

```
1  using System;
2  using System.IO;
3  using System.Text;
4  
5  public class TextSimpleTypes{
6  
7    public static void Main(){
8  
9      StringBuilder sb = new StringBuilder();   // A mutable string
10 
11     using(TextWriter tw = new StringWriter(sb)){
12       for (int i = 0; i < 5; i++){
13         tw.Write(5 * i);  tw.WriteLine();
14         tw.Write(5.5 * i);  tw.WriteLine();
15         tw.Write(5555M * i); tw.WriteLine();
16         tw.Write(5 * i == 6); tw.WriteLine();}
17     }
18
19     Console.WriteLine(sb);
20
21   }
22 }
```

Program 37.19 *A StringWriter program similar to the StreamReader program shown earlier.*

Symmetrically, we illustrate how to read from a string. In Program 37.21 we make a string `str` with broken lines in line 8-11. With use of a `StringReader` built on `str` we read an integer, a double, a decimal, and a boolean value. The output is shown in Program 37.22 (only on web).

```
1  using System;
2  using System.IO;
3
4  public class TextSimpleTypes{
5
6    public static void Main(){
7
8      string str = "5" + "\n" +
9                   "5,5" + "\n" +
10                  "5555,0" + "\n" +
11                  "false";
12
13     using(TextReader tr = new StringReader(str)){
14       int i = Int32.Parse(tr.ReadLine());
15       double d = Double.Parse(tr.ReadLine());
16       decimal m = Decimal.Parse(tr.ReadLine());
17       bool b = Boolean.Parse(tr.ReadLine());
18
19       Console.WriteLine("{0} \n{1} \n{2} \n{3}", i, d, m, b);
20     }
21
22   }
23 }
```

Program 37.21 *A StringReader program.*

The use of `StringWriter` and `StringReader` objects for accessing the characters in strings is an attractive alternative to use of the native `String` and `StringBuilder` operations. It is, in particular, attractive and convenient that we can switch from a file source/destination to a string source/destination. In that way existing file manipulation programs may be used directly as string manipulation programs. The only necessary modification of the program is a replacement of a `StreamReader` with `StringReader`, or a replacement of `StreamWriter` with a `StringWriter`.

Be sure to use the abstract classes `TextReader` and `TextWriter` as much as possible. You should only use `StreamReader`/`StringReader` and `StreamWriter`/`StringWriter` for instantiation purposes in the context of a constructor (such as line 11 of Program 37.19 and line 13 of Program 37.21).

37.15. The Console class

Lecture 10 - slide 23

We have used static methods in the `Console` class in almost all our programs. It is now time to examine the `Console` class a little closer. In contrast to most other IO related classes, the `Console` class resides in the `System` namespace, and not in `System.IO`. The `Console` class encapsulates three streams: *standard input*, *standard output*, and *standard error*. The static property `In`, of type `TextReader`, represents standard input. The static properties `Out` and `Error` represent standard output and standard error respectively, and they are both of type `TextWriter`. Recall in this context that `TextReader` and `TextWriter` are both abstract classes, see Section 37.9.

```
1  using System;
2  using System.IO;
3  
4  class App{
5  
6    public static void Main(string[] args){
7  
8       TextWriter standardOutput = Console.Out;
9       StreamWriter myOut = null,
10                   myError = null;
11 
12      if (args.Length == 2) {
13         Console.Out.WriteLine("Redirecting std output and error to files");
14         myOut = new StreamWriter(args[0]);
15         Console.SetOut(myOut);
16         myError = new StreamWriter(args[1]);
17         Console.SetError(myError);
18      } else {
19         Console.Out.WriteLine("Keeping standard output and error unchanged");
20      }
21 
22      // Output from this section of the program may be redirected
23      Console.Out.WriteLine("Text to std output - by Console.Out.WriteLine");
24      Console.WriteLine("Text to standard output -  by Console.WriteLine(...)");
25      Console.Error.WriteLine("Error msg - by Console.Error.WriteLine(...)");
26 
27      if (args.Length == 2) {
28        myOut.Close(); myError.Close();
29      }
30 
31      Console.SetOut(standardOutput);
32      Console.Out.WriteLine("Now we are back again");
33      Console.Out.WriteLine("Good Bye");
34   }
35 }
```

Program 37.23 *A program that redirects standard output and standard error to a file.*

In the program shown above it is demonstrated how to control standard output and standard error. If we pass two program arguments (`args` in line 6) to Program 37.23 we redirect standard output and standard error to specific files (instances of `StreamWriter`) in line 13-17. That is the main point, which we wish to illustrate in Program 37.23.

Below we supply an overview of the methods and properties of the `Console` class. The `Console` class is static. As such, all methods and properties in class `Console` are static. There will never be objects of type `Console` around. The `Console` class offers the following operations:

- Access to and control of **`in`**, **`out`**, and **`error`**
- **`Write`**, **`WriteLine`**, **`Read`**, and **`ReadLine`** methods
    - Shortcuts to **`out.Write`**, **`out.WriteLine`**, **`in.Read`**, and **`in.ReadLine`**
- Many properties and methods that control the underlying buffer and window
    - Size, colors, and positions
- Immediate, non-blocking input from the Console
    - The property **`KeyAvailable`** returns if a key is pressed (non-blocking)
    - **`ReadKey()`** returns info about the pressed key (blocking)
- Other operations
    - **`Clear()`**, **`Beep()`**, and **`Beep(int, int)`** methods.

38. Directories and Files

The previous chapter was about streams, and as such also about files. In this chapter we will deal with the properties of files beyond reading and writing. File copying, renaming, creation time, existence, and deletion represent a few of these. In addition to files we will also in this chapter discuss directories.

38.1. The File and FileInfo classes

Lecture 10 - slide 26

Two overlapping file-related classes are available to the C# programmer: `FileInfo` and `File`. Both classes belong to the namespace `System.IO`. Objects of class `FileInfo` represents a single file, created on the basis of the name or path of the file (which is a string). The class `File` contains static methods for file manipulation. Class `File` is static, see Section 11.12, and as such there can be no instances of class `File`. If you intend to write object-oriented programs with file manipulation needs it is recommended that you represent files as instances of class `FileInfo`.

Let us right away write a program which illustrates how to use instances of class `FileInfo` for representation of files. All aspects related to class `FileInfo` is shown in **purple** in Program 38.1.

```
using System;
using System.IO;

public class FileInfoDemo{

  public static void Main(){
    // Setting up file names
    string fileName = "file-info.cs",
           fileNameCopy = "file-info-copy.cs";

    // Testing file existence
    FileInfo fi = new FileInfo(fileName); // this source file
    Console.WriteLine("{0} does {1} exist",
                      fileName, fi.Exists ? "" : "not");

    // Show file info properties:
    Console.WriteLine("DirectoryName: {0}", fi.DirectoryName);
    Console.WriteLine("FullName: {0}", fi.FullName);
    Console.WriteLine("Extension: {0}", fi.Extension);
    Console.WriteLine("Name: {0}", fi.Name);
    Console.WriteLine("Length: {0}", fi.Length);
    Console.WriteLine("CreationTime: {0}", fi.CreationTime);

    // Copy one file to another
    fi.CopyTo(fileNameCopy);
    FileInfo fiCopy  = new FileInfo(fileNameCopy);

    // Does the copy exist?
    Console.WriteLine("{0} does {1} exist",
                      fileNameCopy, fiCopy.Exists ? "" : "not");

    // Delete the copy again
    fiCopy.Delete();

    // Does the copy exist?
    Console.WriteLine("{0} does {1} exist",
                      fileNameCopy, fiCopy.Exists ? "" : "not"); // !!??
38
39   // Create new FileInfo object for the copy
40   FileInfo fiCopy1  = new FileInfo(fileNameCopy);
41   // Check if the copy exists?
42   Console.WriteLine("{0} does {1} exist", fileNameCopy,
43              fiCopy1.Exists ? "" : "not");
44
45   // Achieve a TextReader (StreamReader) from the file info object
46   // and echo the lines in the file to standard output
47   using(StreamReader sr = fi.OpenText()){
48     for (int i = 1; i <= 10; i++)
49       Console.WriteLine("  " + sr.ReadLine());
50   }
51  }
52 }
```

Program 38.1 *A demonstration of the FileInfo class.*

In line 12 we create a `FileInfo` object on the source file of the C# program text shown in Program 38.1. In line 13-14 we report on the existence of this file in the file system. (We expect existence, of course). In line 16-22 we access various properties (in the sense of C# properties, see Chapter 18) of the `FileInfo` object. In line 25 we copy the file, and in line 30 we check the existence of the copy. In line 33 we delete the copy, and in line 37 we check the existence of copy again. Against our intuition, we find out that the copy of the file still exists after its deletion. (See next paragraph for an explanation). If, however, we establish a fresh `FileInfo` object on the path to the deleted file, we get the expected result. In line 45-50 we use the `OpenText` method of the `FileInfo` object to establish a `TextReader` on the file. Via a number of `ReadLine` activations in line 49 we demonstrate that we can read the contents of the file.

The file existence problem described above occurs because the instance of class `FileInfo` and the state of the underlying file system become inconsistent. The instance method `Refresh` of class `FileInfo` can be used to update the `FileInfo` object from the information in the operating system. If you need trustworthy information about your files, you should always call the `Refresh` operation before you access any `FileInfo` attribute. If we call `fiCopy.Refresh()` in line 34, the problem observed in line 37 vanishes.

The output of Program 38.1 is shown in Listing 38.2 (only on web).

The following gives an overview of some selected operations in class `FileInfo`:

- A single constructor
  - **`FileInfo(string)`**
- Properties (getters) that access information about the current file
  - Examples: **`Length`**, **`Extension`**, **`Directory`**, **`Exists`**, **`LastAccessTime`**
- Stream, reader, and writer *factory methods*:
  - Examples: **`Create`**, **`AppendText`**, **`CreateText`**, **`Open`**, **`OpenRead`**, **`OpenWrite`**, **`OpenText`**
- Classical file manipulations
  - **`CopyTo`**, **`Delete`**, **`MoveTo`**, **`Replace`**
- *Others*
  - **`Refresh`**, ...

The parameter of the `FileInfo` constructor is an absolute or relative path to a file. The file path must be *well-formed* according to a set of rules described in the class documentation. As examples, the file paths "`c:\temp c:\user`" and "` dir1\dir2\file.dat`" are both malformed.

We have also written af version of Program 38.1 in which we use the static class `File` instead of `FileInfo`, see Program 38.3. We do not include this program, nor the listing of its output, in the paper edition of the material. We notice that the file existence frustrations in Program 38.1 (of the deleted file) do not appear when we use the static operations of the static class `File`.

There is a substantial overlap between the instance methods of class **`FileInfo`** and the static methods in class **`File`**

38.2. The Directory and DirectoryInfo classes

Lecture 10 - slide 28

The classes `DirectoryInfo` and `Directory` are natural directory counterparts of the classes `FileInfo` and `File`, as described in Section 38.1. In this section we will show an example use of class `DirectoryInfo`, and we will provide an overview of the members in the class. Like for files, an instance of class `DirectoryInfo` is intended to represent a given directory of the underlying file system. We recommend that you use the class `DirectoryInfo`, rather than the static class `Directory`, when you write object-oriented programs.

It is worth noticing that the classes `FileInfo` and `DirectoryInfo` have a common abstract, superclass class `FileSystemInfo`.

Here follows a short program that use an instance of class `DirectoryInfo` for representation of a given directory from the underlying operating system.

```
using System;
using System.IO;

public class DirectoryInfoDemo{

  public static void Main(){
    string fileName = "directory-info.cs";    // The current source file

    // Get the DirectoryInfo of the current directory
    // from the FileInfo of the current source file
    FileInfo fi = new FileInfo(fileName);     // This source file
    DirectoryInfo di = fi.Directory;

    Console.WriteLine("File {0} is in directory \n   {1}", fi, di);

    // Get the files and directories in the parent directory.
    FileInfo[] files = di.Parent.GetFiles();
    DirectoryInfo[] dirs = di.Parent.GetDirectories();

    // Show the name of files and directories on the console
    Console.WriteLine("\nListing directory {0}:", di.Parent.Name);
    foreach(DirectoryInfo d in dirs)
      Console.WriteLine(d.Name);
    foreach(FileInfo f in files)
      Console.WriteLine(f.Name);

  }
}
```

Program 38.5 *A demonstration of the DirectoryInfo class.*

Like in Program 38.3 the starting point in Program 38.5 is a `FileInfo` object that represents the source file shown in Program 38.5. Based on the `FileInfo` object, we create a `DirectoryInfo` object in line 12. This `DirectoryInfo` object represents the directory in which the actual source file resides. Let us call it the *current directory* . In line 17 we illustrate the `Parent` property and the `GetFiles` method; We create an array, `files`, of `FileInfo` object of the parent directory of the current directory. Thus, this array holds all files of the parent of current directory. Similarly, `dirs` declared in line 18 is assigned to hold all directories of the parent of current directory. We print these files and directories in line 20-25.

The output of Program 38.5 (only on web) is shown in Listing 38.6 (only on web). A similar program, programmed with use of the static operations in class `Directory`, is shown in Program 38.7 (only on web).

The following shows an overview of the instance properties and instance methods in class **`DirectoryInfo`**:

- A single constructor
  - **`DirectoryInfo(string)`**
- Properties (getters) that access information about the current directory
  - Examples: **`CreationTime`**, **`LastAccessTime`**, **`Exists`**, **`Name`**, **`FullName`**
- Directory Navigation operations
  - Up: **`Parent`**, **`Root`**
  - Down: **`GetDirectories`**, **`GetFiles`**, **`GetFileSystemInfo`** (all overloaded)
- Classical directory manipulations
  - **`Create`**, **`MoveTo`**, **`Delete`**
- *Others*
  - **`Refresh`**, ...

The constructor takes a directory path string as parameter. It is possible to create a `DirectoryInfo` object on a string that represents a non-existing directory path. Like file paths, the given directory path must be well-formed (according to rules stated in the class documentation).

The downwards directory navigation operations `GetDirectories`, `GetFiles`, and `GetFileSystemInfo` are able to filter their results (with use of strings with wildcards, such as `"temp*"`, which match all files/directories whose names start with `"temp"`). It is also possible to specify if the operations should access direct files/directories, or if they should access direct as well as indirect file/directories.

As for **`File`** and **`FileInfo`**, there is substantial overlap between the classes **`Directory`** and **`DirectoryInfo`**

39. Serialization

In this material we care about object-oriented programming. All our data are encapsulated in objects. When we deal with IO it is therefore natural to look for solutions that help us with *output and input of objects*.

For each class `C` it is possible to decide a *storage format*. The storage format of class `C` tells which pieces of data in `C` instances to save on secondary storage. The details of the storage format need to be decided. This involves (1) which fields to store, (2) the sequence of fields in the stored representation, and (3) use of a binary or a textual representation. However, as long as we have pairs of `WriteObject` and `ReadObject` operations for which `ReadObject(WriteObject(C-object))` is equivalent to `C-object` the details of the storage format are of secondary interest.

Instances of class `C` may have references to instances of other classes, say `D` and `E`. In general, an instance of class `C` may be part of an *object graph* in which we find `C`-object, `D`-object, `E`-objects as well as objects of other types. We soon realize that the real problem is not how to store instances of `C` in isolation. Rather, the problem is how to store an object network in which `C`-objects take part (or in which a `C`-object is a root).

People who have devised a storage format for a class `C`, who have written then `WriteObject` and `ReadObject` operations for class `C`, and who have dealt with the IO problem of object graphs quickly realize that the invented solutions generalizes to arbitrary classes. Thus, instead of solving the object IO problem again and again for specific classes, it is attractive to solve the problem at a general level, and make the solution available for arbitrary classes. This is exactly what serialization is about. The serialization problem has been solved by the implementers of C#. It is therefore easy for the C# programmer to save and retrieve objects via serialization.

39.1. Serialization

Lecture 10 - slide 31

Serialization provides for input and output of a network of objects. Serialization is about object output, and deserialization is about object input.

- Serialization
  - Writes an object *o* to a file
  - Also writes the objects referred from *o*
- Deserialization
  - Reads a serialized file in order to reestablish the serialized object *o*
  - Also reestablishes the network of objects originally referred from *o*

Serialization of objects is, in principle, simple to deal with from C#. There are, however, a couple of circumstances that complicate the matters:

- The need to control or customize the serialization and the deserialization of objects of specific types.
- The support of more than one C# technique to obtain the same serialization or deserialization effect.

The need to control (customize) the details of serialization and deserialization is unavoidable, at least when the ideas should be applied on real-life examples.

The support of several different techniques for doing serialization is due to the development of C#. In C# 2.0 serialization relies almost exclusively on the use of serialization and deserialization attributes. In C# 1.0 it was also necessary to implement certain interfaces to control and customize the serialization. In this version of the material, we only describe serialization controlled by attributes.

- Serialization and deserialization is supported via classes that implement the **`Iformatter`** interface:
    - `BinaryFormatter` and `SoapFormatter`
- Methods in `Iformatter`:
    - `Serialize` and `Deserialize`

In the following section we will discuss an example that uses `BinaryFormatter`.

39.2. Examples of Serialization in C#

Lecture 10 - slide 32

Below we show the class `Person` and class `Date`, similar to the ones we used for illustration of privacy leaks in Section 16.5. Class `Person` in Program 39.1 encapsulates a name and two date objects: birth date and death date. For a person still alive, the death date refer to `null`. *Redundantly*, the `age` instance variable holds the age of the person. The `Update` method can be used to update the `age` variable.

The `Date` class shown in Program 39.2 is a very simple implementation of a date class. (In the paper version of the material we only show an outline of the `Date` class. The complete version is available in the web version). The `Person` class relies on the `Date`. We use class `Date` for illustration of serialization; In real life you should always use the struct `DateTime`. The `Date` class encapsulates year, month, and day. In addition it holds a `nameOfDay` instance variable (with values such as `Sunday` or `Monday`), which is *redundant*. With appropriate calendar knowledge, the `nameOfDay` can be calculated from `year`, `month`, and `day`. The `Person` class needs age calculation, which is provided by the `YearDiff` method of class `Date`. Internally in class `Date`, `YearDiff` relies on the methods `IsBefore` and `Equals`. (`Equals` is defined according the standard recommendations, see Section 28.16. We have not, in this class, included a redefinition of `GetHashCode` and therefore we get a warning from the compiler when class `Date` is compiled.)

The *redundancy* is class `Person` and class `Date` is introduced on purpose, because it helps us illustrate the serialization control in Program 39.2. In most circumstances we would avoid such redundancy, at least in simple classes.

The preparation of class `Person` and class `Date` for serialization is very simple. We mark both classes with the attribute **`[Serializable]`**, see line 3 in both classes. As of now you can consider **`[Serializable]`** as some magic, special purpose notation. In reality **`[Serializable]`** represents application of an attribute. When we are done with serialization we have seen several uses of attributes, and therefore we will be motivated to understand the general ideas of attributes in C#. We discuss the general ideas behind attributes in Section 39.6.

Please notice that in the paper version of this material most program examples have been abbreviated. The full details of all examples appear in the web version of the material.

```
using System;

[Serializable]
public class Person{

  private string name;
  private int age;    // Redundant
  private Date dateOfBirth, dateOfDeath;

  public Person (string name, Date dateOfBirth){
    this.name = name;
    this.dateOfBirth = dateOfBirth;
    this.dateOfDeath = null;
    age = Date.Today.YearDiff(dateOfBirth);
  }

  public Date DateOfBirth {
    get {return new Date(dateOfBirth);}
  }

  public int Age{
    get {return Alive ? age : dateOfDeath.YearDiff(dateOfBirth);}
  }

  public bool Alive{
    get {return dateOfDeath == null;}
  }

  public void Died(Date d){
    dateOfDeath = d;
  }

  public void Update(){
    age = Date.Today.YearDiff(dateOfBirth);
  }

  public override string ToString(){
    return "Person: " + name +
            "  *" + dateOfBirth +
            (Alive ? "" : "  +" + dateOfDeath) +
            "  Age: " + age;
  }

}
```

Program 39.1 *The Person class - Serializable.*

```
using System;

[Serializable]
public class Date{
  private ushort year;
  private byte month, day;
  private DayOfWeek nameOfDay;    // Redundant

  public Date(int year, int month, int day){
    this.year =  (ushort)year;
    this.month = (byte)month;
    this.day =   (byte)day;
    this.nameOfDay = (new DateTime(year, month, day)).DayOfWeek;
  }

  public Date(Date d){
    this.year = d.year; this.month = d.month;
    this.day = d.day; this.nameOfDay = d.nameOfDay;
  }

  public int Year{get{return year;}}
  public int Month{get{return month;}}
  public int Day{get{return day;}}

  // return this minus other, as of usual birthday calculations.
  public int YearDiff(Date other){
    // ...
  }

  public override bool Equals(Object obj){
    // ...
  }

  // Is this date less than other date
  public bool IsBefore(Date other){
    // ...
  }

  public static Date Today{
    // ...
  }

  public override string ToString(){
    return string.Format("{0} {1}.{2}.{3}", nameOfDay, day, month, year);
  }
}
```

Program 39.2 *An outline of the Date class - Serializable.*

In Program 39.3 it is illustrated how to serialize and deserialize a graph of objects. The graph, which we serialize, consists of one `Person` and the two `Date` objects referred by the `Person` object. The serialization, which takes place in line 13-17, is done by sending the `Serialize` message to the `BinaryFormatter` object. The serialization relies on a binary stream, as represented by an instance of class `FileStream`, see Section 37.4.

The deserialization, as done in line 24-28, will in most real-life settings be done in another program. In our example we reset the program state in line 19-22 before the deserialization. The actual deserialization is done by sending the `Deserialize` message to the `BinaryFormatter` object. As in the serialization, the file stream with the binary data, is passed as a parameter.

```
1  using System;
2  using System.IO;
3  using System.Runtime.Serialization;
4  using System.Runtime.Serialization.Formatters.Binary;
5
6  class Client{
7
8    public static void Main(){
9      Person p = new Person("Peter", new Date(1936, 5, 11));
10     p.Died(new Date(2007,5,10));
11     Console.WriteLine("{0}", p);
12
13     using (FileStream strm =
14                new FileStream("person.dat", FileMode.Create)){
15       IFormatter fmt = new BinaryFormatter();
16       fmt.Serialize(strm, p);
17     }
18
19     // ----------------------------------------------------------
20     p = null;
21     Console.WriteLine("Reseting person");
22     // ----------------------------------------------------------
23 
24     using (FileStream strm =
25                new FileStream("person.dat", FileMode.Open)){
26       IFormatter fmt = new BinaryFormatter();
27       p = fmt.Deserialize(strm) as Person;
28     }
29 
30     Console.WriteLine("{0}", p);
31   }
32 
33 }
```

Program 39.3 *The Person client class - applies serialization and deserialization.*

The program output shown in Listing 39.4 tells that the `Person` object and the two `Date` objects have survived the serialization and deserialization processes. In between the two output lines in line 11 and line 30 of Program 39.3 the three objects have been transferred to and reestablished from the binary file.

```
Person: Peter  *Monday 11.5.1936  +Thursday 10.5.2007  Age: 71
Reseting person
Person: Peter  *Monday 11.5.1936  +Thursday 10.5.2007  Age: 71
```

Listing 39.4 *Output of the Person client class.*

Exercise 10.5. *Serializing with an XML formatter*

In the programs shown on the accompanying slide we have used a binary formatter for serialization of `Person` and `Date` object.

Modify the client program to use a so-called Soap formatter in the namespace `System.Runtime.Serialization.Formatters.Soap`. SOAP is an XML language intended for exchange of XML documents. SOAP is related to the discipline of web services in the area of Internet technology.

After the serialization you should take a look at the file `person.dat`, which is written and read by the client program.

39.3. Custom Serialization

Lecture 10 - slide 33

In the `Person` and `Date` classes, shown in Section 39.2, the redundant instance variables do not need to be serialized. In class `Person`, `age` does need to be serialized because it can be calculated from `dateOfBirth` and `dateOfDeath`. In class `Date`, `nameOfDay` does need to serialized because it can calculated from calendar knowledge. In relation to serialization and persistence, we say that these two instance variables are *transient*. It is sufficient to serialize the essential information, and to reestablish the values of the transient instance variables after deserialization. In Program 39.5 and Program 39.6 we show the serialization and the deserialization respectively.

The serialization is controlled by marking some fields (instance variables) as **[NonSerialized]**, see line 9 of Program 39.5 and line 9 of Program 39.6.

The deserialization is controlled by a method marked with the attribute **[OnDeserialized()]**, see line 21 of Program 39.5. This method is called when deserialization takes place. The method starting at line 21 of Program 39.5 assigns the redundant `age` variable of a `Person` object.

```
1  using System;
2  using System.Runtime.Serialization;
3  
4  [Serializable]
5  public class Person{
6  
7    private string name;
8  
9    [NonSerialized()]
10   private int age;
11 
12   private Date dateOfBirth, dateOfDeath;
13 
14   public Person (string name, Date dateOfBirth){
15     this.name = name;
16     this.dateOfBirth = dateOfBirth;
17     this.dateOfDeath = null;
18     age = Date.Today.YearDiff(dateOfBirth);
19   }
20 
21   [OnDeserialized()]
22   internal void FixPersonAfterDeserializing(
23                            StreamingContext context){
24     age = Date.Today.YearDiff(dateOfBirth);
25   }
26 
27   // ...
28 
29 }
```

Program 39.5 *The Person class - Serialization control with attributes.*

The `Date` class shown below in Program 39.6 follows the same pattern as the `Person` class of Program 39.5.

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class Date{
  private ushort year;
  private byte month, day;

  [NonSerialized()]
  private DayOfWeek nameOfDay;

  public Date(int year, int month, int day){
    this.year =  (ushort)year;
    this.month = (byte)month;
    this.day =   (byte)day;
    this.nameOfDay = (new DateTime(year, month, day)).DayOfWeek;
  }

  public Date(Date d){
    this.year = d.year; this.month = d.month;
    this.day = d.day; this.nameOfDay = d.nameOfDay;
  }

  [OnDeserialized()]
  internal void FixDateAfterDeserializing(
                                 StreamingContext context){
    nameOfDay = (new DateTime(year, month, day)).DayOfWeek;
  }

  // ...
}
```

Program 39.6 *The Date class - Serialization control with attributes .*

39.4. Considerations about Serialization

Lecture 10 - slide 34

We want to raise a few additional issues about serialization:

- Security
    - Encapsulated and private data is made available in files
- Versioning
    - The private state of class C is changed
    - It may not be possible to read serialized objects of type C
- Performance
    - Some claim that serialization is relatively slow

39.5. Serialization and Alternatives

Lecture 10 - slide 35

As mentioned in the introduction of this chapter - Chapter 39 - serialization deals with input and output of objects and object graphs. It should be remembered, however, that there are alternatives to serialization. As summarized below, it is possible to program object IO at a low level (using binary of textual IO primitives from Chapter 37). At the other end of the spectrum it is possible us database technology.

- Serialization
    - An easy way to save and restore objects in between program sessions
    - Useful in many projects where persistency is necessary, but not a key topic
    - Requires only little programming
- Custom programmed file IO
    - Full control of object IO
    - May require a lot of programming
- Objects in Relational Databases
    - *Impedance mismatch*: "Circular objects in retangular boxes"
    - Useful when the program handles large amounts of data
    - Useful if the data is accessed simultaneous from several programs
    - Not a topic in this course

39.6. Attributes

Lecture 10 - slide 36

In our treatment of serialization we made extensive use of attributes, see for instance Section 39.3. In this section we will discuss attributes at a more general level, and independent of serialization.

Attributes offer a mechanism that allows the programmer to extend the programming language in simple ways. Attributes allow the programmer to associate extra information (meta data) to selected and pre-defined constructs in C#. The constructs to which it is possible to attach attributes are assemblies, classes, structs, constructors, delegates, enumeration types, fields (variables), events, methods, parameters, properties, and returns.

We all know that members of a class in C# have associated visibility modifiers, see Section 11.16. In case visibility modifiers were not part of C#, we could have used attributes as a way to extend the language with different kinds of member visibilities. Certain attributes can be accessed by the compiler, and hereby these attributes can affect the checking done by the compiler and the code generated by the compiler. Attributes can also be accessed at run-time. There are ways for the running program to access the attributes of given constructs, such that the attribute and attribute values can affect the program execution.

Program 39.7 illustrates the use of the predefined `Obsolete` attribute. Being "obsolete" means "no longer in use". In line 3, the attribute is associated with class `C`. In line 9, another usage of the attribute is associated with method `M` in class `D`.

```
1  using System;
2  
3  [Obsolete("Use class D instead")]
4  class C{
5    // ...
6  }
7  
8  class D{
9    [Obsolete("Do not call this method.",true)]
10   public void M(){
11   }
12 }
13 
14 class E{
15   public static void Main(){
16     C c = new C();
17     D d = new D();
18     d.M();
19   }
20 }
```

Program 39.7 *An obsolete class C, and a class D with an obsolete method M.*

The compiler is aware of the `Obsolete` attribute. When we compile Program 39.7 we can see the effect of the attribute, see Listing 39.8.

```
>csc prog.cs
Microsoft (R) Visual C# 2005 Compiler version 8.00.50727.42
for Microsoft (R) Windows (R) 2005 Framework version 2.0.50727
Copyright (C) Microsoft Corporation 2001-2005. All rights reserved.

prog.cs(16,5): warning CS0618: 'C' is obsolete: 'Use class D instead'
prog.cs(16,15): warning CS0618: 'C' is obsolete: 'Use class D instead'
prog.cs(18,5): error CS0619: 'D.M()' is obsolete: 'Do not call this method.'
```

Listing 39.8 *Compiling class C, D, and E.*

C# comes with a lot of predefined attributes. `Obsolete` is one of them, and we encountered quite a few in Section 39.3 in the context of serialization. Unit testing frameworks for C# also heavily rely on attributes.

It is also possible to define our own attributes. An attribute is defined as a class. Attributes defined in this way are subclasses of the class `System.Attribute`. As a naming convention, the names of all attribute classes should have "`Attribute`" as a suffix. Thus, an attribute `X` is defined by a class `XAttribute`, which inherits from the class `System.Attribute`. The attribute usage notation **`[X(a,b,c)]`** in front of some C# construct *C* causes an instance of class `XAttribute`, made with the appropriate three-parameter constructor, to be associated with *C*. In the attribute usage notation **`[X(a,b,c,d=e)]`** `d` refers to a property of class `XAttribute`. The property `d` must be read-write (both gettable and settable), see Section 18.5. Thus, as it appears, an attribute accepts both *positional parameters* and *keyword parameters*.

Below, in Program 39.9 we have reproduced the class behind the `Obsolete` attribute. You should notice the three different constructors and the read/write property `IsError`. The attribute `AttributeUsage` attribute in 5-6 illustrates how attributes help define attributes. `AttributeUsage` define the constructs to which it possible to associate the `MyObsolete` attribute. The expression `AttributeTargets.Method | AttributeTargets.Property` denotes two values in the *combined enumeration type* `AttributeTargets` which carries a so-called flag attribute. Combined enumerations are discussed in Focus box 6.3.

```
1  // In part, reproduced from the book "C# to the Point"
2  
3  using System;
4  
5  [AttributeUsage(AttributeTargets.Method |
6                  AttributeTargets.Property)]
7  public sealed class MyObsoleteAttribute: Attribute{
8    string message;
9    bool isError;
10 
11   public string Message{
12     get {
13       return message;
14     }
15   }
16 
17   public bool IsError{
18     get {
19       return isError;
20     }
21     set {
22       isError = value;
23     }
24   }
25 
26   public MyObsoleteAttribute(){
27     message = ""; isError = false;
28   }
29 
```

```
  public MyObsoleteAttribute(string msg){
    message = msg; isError = false;
  }

  public MyObsoleteAttribute(string msg, bool error){
    message = msg; isError = error;
  }

}
```

Program 39.9   *A reproduction of class ObsoleteAttribute.*

In Program 39.10 we show a sample use of the attribute programmed in Program 39.9. The program does not compile because we attempt to associate the `MyObsolete` attribute to a class in line 3. As explained above, we have restricted `MyObsolete` to be connected with only methods and properties.

```
1  using System;
2  
3  [MyObsolete("Use class D instead")]
4  class C{
5    // ...
6  }
7  
8  class D{
9    [MyObsolete("Do not call this method.",IsError=true)]
10   public void M(){
11   }
12 }
13 
14 class E{
15   public static void Main(){
16     C c = new C();
17     D d = new D();
18     d.M();
19   }
20 }
```

Program 39.10   *Sample usage of the reproduced class - causes a compilation error.*

40. Patterns and Techniques

In relation to streams, which we discussed in Chapter 37 in the beginning of the IO lecture, it is relevant to bring up the ***Decorator*** design pattern. Therefore we conclude the IO lecture with a discussion of ***Decorator***.

40.1. The Decorator Pattern

Lecture 10 - slide 38

It is often necessary to extend an object of class `C` with extra capabilities. As an example, the `Draw` method of a `Triangle` class can be extended with the traditional angle and edge annotations for equally sized angles or edges. The typical way to solve the problem is to define a subclass of class `C` that extends `C` in the appropriate way. In this section we are primarily concerned with extensions of class `C` that do not affect the client interface of `C`. Therefore, the extensions we have in mind behave like specializations (see Chapter 25). The extensions we will deal with consist of adding additional code to the existing methods of `C`.

The decorator design pattern allows us to extend a class dynamically, at run-time. Extension by use of inheritance, as discussed above, is static because it takes place at compile-time. The main idea behind ***Decorator*** is a chain of objects, along the line illustrated in Figure 40.1. A message from `Client` to an instance of `ConcreteComponent` is passed through two instances of `ConcreteDecorator` by means of *delegation*. In order to arrange such delegation, a `ConcreteDecorator` and a `ConcreteComponent` should implement a common interface. This is important because a `ConcreteDecorator` is used as a stand in for a `ConcreteComponent`. This arrangement can for instance be obtained by the class hierarchy shown in Figure 40.2.

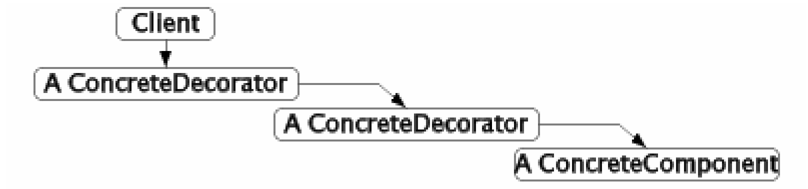

Figure 40.1 *Two decorator objects of a ConcreteComponent object*

In Figure 40.2 the `Decorators` and the `ConcreteComponent` share a common, abstract superclass called *`Component`*. When a `Client` operate on a `ConcreteComponent` it should do so via the type *`Component`*. This facilitates the object organization of Figure 40.1, because a `Decorator` can act as a stand in for a `ConcreteComponent`.

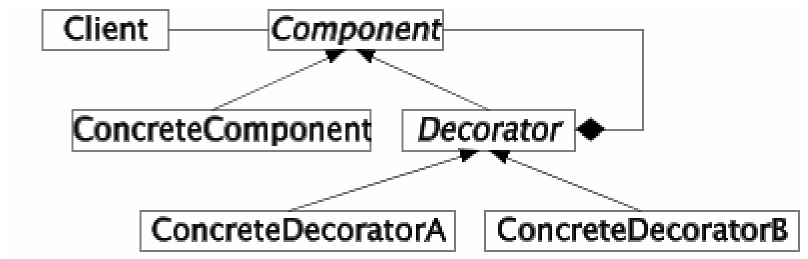

Figure 40.2 *A template of the class structure in the Decorator design pattern.*

- **Component**: Defines the common interface of participants in the Decorator pattern
- **Decorator**: References another Component to which it delegates responsibilities

The class diagram of ***Decorator*** is similar to ***Composite***, see Section 32.1. In Figure 40.2 a ***Decorator*** is intended to aggregate (reference) a single *`Component`*. In Figure 32.1 a ***Composite*** typically aggregate two or more *`Components`* . Thus, a ***Composite*** typically gives rise to trees, whereas a ***Decorator*** gives rise to a linear lists.

`Decorator` objects can be added and chained at run-time. A `Client` accesses the outer *`Component`* (typically a `ConcreteDecorator`), which delegates part of the work to another *`Component`*. While passing, it does part of the work itself.

Use of ***Decorator*** can be seen as a dynamic alternative to static subclassing

40.2. The Decorator Pattern and Streams

Lecture 10 - slide 40

The ***Decorator*** discussion above in Section 40.1 was abstract and general. It is not obvious how it relates to streams and IO. We will now introduce the stream decorators that drive our interest in the pattern. The following summarizes the stream classes that are involved.

We build a **compressed stream** on a **buffered stream** on a **file stream**

The **compressed stream** decorates the **buffered stream**

The **buffered stream** decorates the **file stream**

The idea behind the decoration of class `FileStream` (see Section 37.4) is to supply additional properties of the stream. The additional properties in our example are *buffering* and *compression*. Buffering may result in better performance because many read and write operations do not need to touch the harddisk as such. Use of compression means that the files become smaller. (Notice that class `FileStream` already apply buffering itself, and as such the buffer decoration is more of illustrative nature than of practical value).

Figure 40.3 corresponds to Figure 40.1. Thus, Figure 40.3 shows objects, not classes. A `FileStream` object is decorated with buffering and compression. A `Client` program is able to operate on `GZipStream` (a compressed stream) as if it was a `FileStream`.

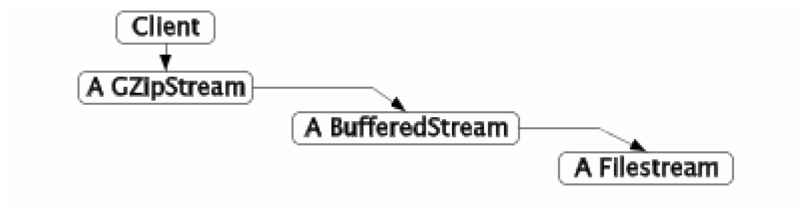

Figure 40.3 *Compression and buffering decoration of a FileStream*

In Program 40.1 we read a `FileStream` into a buffer of type `byte[]`. This is done in line 11-16. In line 18-27 we establish the decorated `FileStream` (see the **purple** parts). In line 27 we write the buffer to the decorated stream. In line 29-32 we compare the size of the original file and the compressed file. We see the effect in Listing 40.2 when the program is applied on its own source file.

```
 1 using System;
 2 using System.IO;
 3 using System.IO.Compression;
 4 
 5 public class CompressProg{
 6 
 7   public static void Main(string[] args){
 8     byte[] buffer;
 9     long originalLength;
10 
11     // Read a file, arg[0], into buffer
12     using(Stream infile = new FileStream(args[0], FileMode.Open)){
13        buffer = new byte[infile.Length];
14        infile.Read(buffer, 0, buffer.Length);
15        originalLength = infile.Length;
16     }
17 
18     // Compress buffer to a GZipStream
19     Stream compressedzipStream =
20        new GZipStream(
21           new BufferedStream(
22                   new FileStream(
23                         args[1], FileMode.Create),
24                   128),
25           CompressionMode.Compress);
26     compressedzipStream.Write(buffer, 0, buffer.Length);
27     compressedzipStream.Close();
28 
29     // Report compression rate:
30     Console.WriteLine("CompressionRate: {0}/{1}",
31                       MeasureFileLength(args[1]),
32                       originalLength);
33 
34   }
35 
36   public static long MeasureFileLength(string fileName){
37     using(Stream infile = new FileStream(fileName, FileMode.Open))
38       return infile.Length;
39   }
40 
41 }
```

Program 40.1 *A program that compresses a file.*

```
> compress compress.cs out
CompressionRate: 545/1126
```

Listing 40.2 *Sample application together with program output (compression rate).*

When Program 40.1 is executed, a compressed file is written. In Program 40.3 we show how to read this file back again. In line 11-17 we set up the decorated stream, very similar to Program 40.1. In line 21-28 we read the compressed file into the buffer, and finally in line 32-35 we write the buffer back to an uncompressed file.

```
using System;
using System.IO;
using System.IO.Compression;

public class CompressProg{

  public static void Main(string[] args){
    byte[] buffer;
    const int LargeEnough = 10000;

    Stream compressedzipStream =
       new GZipStream(
         new BufferedStream(
                new FileStream(
                      args[0], FileMode.Open),
                128),
         CompressionMode.Decompress);

    buffer = new byte[LargeEnough];

    // Read and decompress the compressed stream:
    int bytesRead = 0,
        bufferPtr = 0;
    do{
      // Read chunks of 10 bytes per call of Read:
      bytesRead = compressedzipStream.Read(buffer, bufferPtr, 10);
      if (bytesRead != 0) bufferPtr += bytesRead;
    } while (bytesRead != 0);

    compressedzipStream.Close();

    // Write contens of buffer to the output file
    using(Stream outfile = new FileStream(args[1], FileMode.Create)){
       outfile.Write(buffer, 0, bufferPtr);
    }
  }

}
```

Program 40.3 *The corresponding program that decompresses the file.*

With this we are done with the IO lecture.