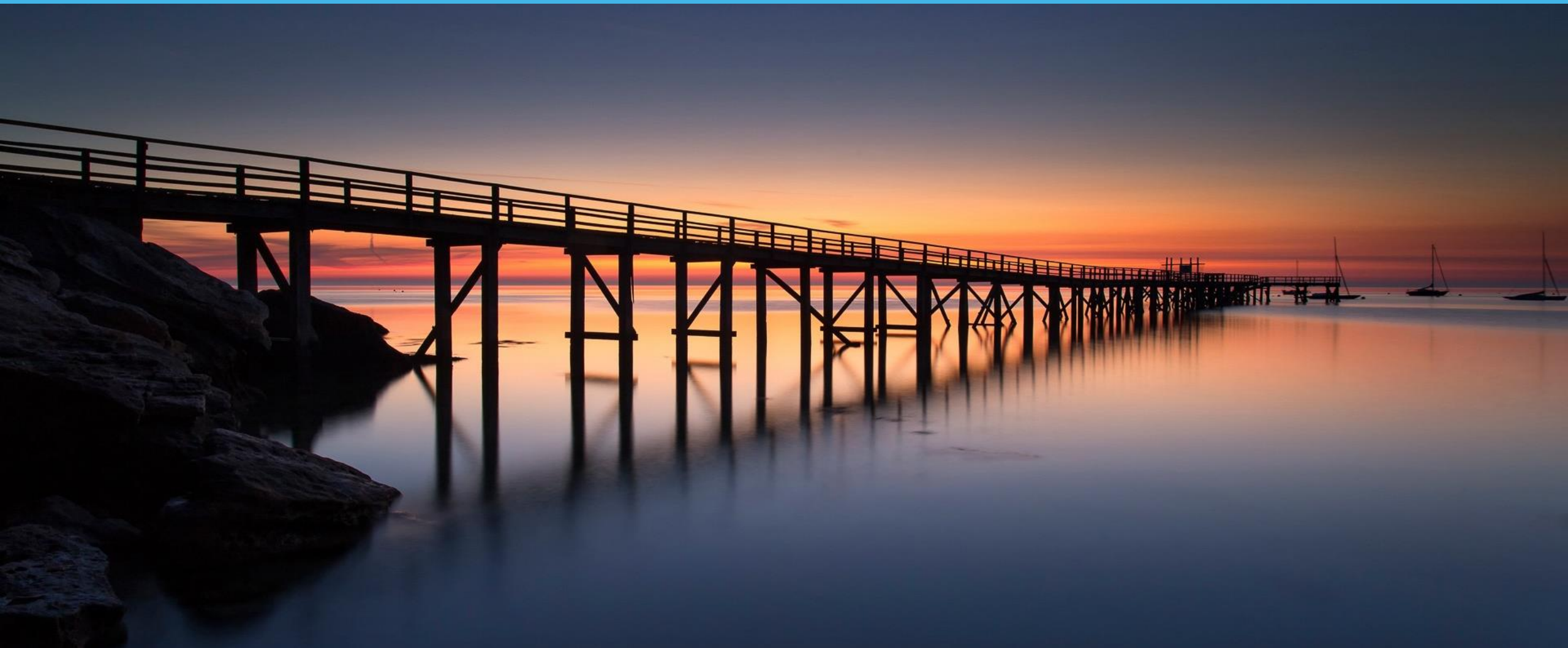
Input and output

7. november 2017



Agenda

oxygen

Streams

Directories and files

Serialization

Patterns and techniques

Streams

Streams

oxygen

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.

At the abstract level, the Stream class is the most important class in the IO landscape.

The concept of streams

oxygen

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.

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.

The concept of streams (cont.)

oxygen

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.

The abstract class Stream in C#

oxygen

Belong to the System.IO namespace

Supports both synchronous and asynchronous operations

Italic operation names indicate abstract operations

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()

The abstract class Stream in C# (cont.)

oxygen

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*, *pos*, *len*) reads at most *len* bytes, and stores them in buf[*pos*] ... buf[*pos*+*len-1*].

Read returns the actual number of characters read, which can be less than *len*.

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*, *pos*, *len*) writes *len* bytes, buf[*pos*] ... buf[*pos*+*len-1*], to the stream.

The abstract class Stream in C# (cont.)

oxygen

Only *WriteByte* and *ReadByte* are non-abstract operations.

ReadByte returns the integer value of the byte being read, or *-1* in case that the end of the stream is reached.

The two operations *ReadByte* and *WriteByte* rely on *Read* and *Write*.

ReadByte calls *Read* on a one-byte array, it accesses this byte, and returns this byte.

WriteByte works in a similar way.

Based on this information, it is obvious that overriding *ReadByte* and *WriteByte* is necessary in specialised stream implementations.

The default implementation is to inefficient.

The abstract class Stream in C# (cont.)

oxygen

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 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.

Subclasses of class Stream

oxygen

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.

Subclasses of class Stream (cont.)

oxygen

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...

Example: FileStreams

oxygen

Please notice, 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. (More on that later)

In this simple example, we write bytes directly to a file. We do not interpret the bytes.

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


    Stream s = new FileStream("myFile.bin", FileMode.Open);
    int i, j, k, m, n;
    i = s.ReadByte();  // O   79  01001111
    j = s.ReadByte();  // O   79  01001111
    k = s.ReadByte();  // P   80  01010000
    m = s.ReadByte();  // -1  EOF
    n = s.ReadByte();  // -1  EOF

    Console.WriteLine("{0} {1} {2} {3} {4}", i, j, k, m, n);
    s.Close();
    Console.ReadLine();
}
```

The *using* control structure

oxygen

The simple file reading and writing example 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...
}
```

The *using* control structure (cont.)

oxygen

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

Example: FileStream with *using*

oxygen

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


    using (Stream s = new FileStream("myFile.bin", FileMode.Open))
    {
        int i, j, k, m, n;
        i = s.ReadByte();  // O   79  01001111
        j = s.ReadByte();  // O   79  01001111
        k = s.ReadByte();  // P   80  01010000
        m = s.ReadByte();  // -1  EOF
        n = s.ReadByte();  // -1  EOF

        Console.WriteLine("{0} {1} {2} {3} {4}", i, j, k, m, n);
    }
    Console.ReadLine();
}
```

The *using* control structure (cont.)

oxygen

The following fragment shows what is actually covered by a using construct.

Most important, a try-finally construct is involved.

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.

We proved that during the previous lecture.

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

Example: Multiple filestreams

oxygen

```
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;
    }
}
```

The Encoding class

oxygen

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 rather it was 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.

The Encoding class (cont.)

oxygen

The naïve solution:

We want to be able to represent a maximum of 200000 different characters.

For this we need log2(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.

The Encoding class (cont.)

oxygen

As of 2007, the Unicode standard defines more than 100000 different characters.

Unicode organizes characters in a number of planes of up to 216 (= 65536) characters.

The Basic Multilingual Plane - BMP - contains the most common characters.

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 earlier 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.

The Encoding class (cont.)

oxygen

It was hypothesized that 16 bits was enough to hold all characters in the world, but as we discovered, this is not the case.

Char in C# is designed to hold 16 bits and therefore cannot represent the entire Unicode spectrum.

Solution?

Use multiple Char to represent a given character. History repreats 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*

The Encoding class (cont.)

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)

Exercise

oxygen

Download Encoding.cs and play around with encodings in the method.

Readers and Writers in C#

oxygen

The table 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 text (char/string) IO
and binary (bits structured as bytes) IO.

	Input	Output
Text	TextReader StreamReader StringReader …	TextWriter StreamWriter StringWriter …
Binary	BinaryReader	BinaryWriter

Readers and Writers in C# (cont.)

oxygen

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.

Readers and Writers in C# (cont.)

oxygen

We will now discuss how the reader and writer classes are related to the stream classes.

None of the classes in the table 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.

Exercise – StreamWriter.cs

oxygen

Open the file StreamWriter.cs

Implement 3 stream writers, called tw1, tw2, tw3
Use a FileStream to create a new file for each StreamWriter

Hint: **new StreamWriter(new FileStream(…), …);**

Make sure that:

- tw1 uses “iso-8859-1” encoding
- tw2 uses UTF8
- tw3 uses UTF16

Examine the files and check for differences

The class TextWriter

oxygen

Writing simple and complex types using a TextWriter:

Output simple:

5

5.5

5555

False

```
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();
    }
}
```

Output complex:

Point: (1, 2)

Die[6]: 3

Explain the Complex output!

The class TextWriter (cont.)

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*

The TextReader class

oxygen

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.

Exercise – Extend StreamWriter.cs

oxygen

Extend your main method to include reading of the newly created files.

Implement 3 stream readers, called tr1, tr2, tr3

Use a FileStream to create a new file for each StreamReader

Hint: **new StreamReader(new FileStream(…), …);**

Make sure that you read the files with appropriate encodings

Finish off with:

```
Console.WriteLine(tr1.ReadLine());  Console.WriteLine(tr1.ReadLine());
Console.WriteLine(tr2.ReadLine());  Console.WriteLine(tr2.ReadLine());
Console.WriteLine(tr3.ReadLine());  Console.WriteLine(tr3.ReadLine());

tr1.Close();
tr2.Close();
tr3.Close();
```

The TextReader class (cont.)

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

BinaryWriter

oxygen

As such, a binary writer is similar to a FileStream used in write access mode

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.

BinaryWriter (cont.)

oxygen

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*

Exercise – Modify StreamWriter.cs

oxygen

Modify StreamWriter.cs to use binary writers instead.

BinaryReader

oxygen

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).

BinaryReader (cont.)

oxygen

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

- Two public constructors
  - BinaryReader(Stream)
  - BinaryReader(Stream, Encoding)
- 15 individually name Readtype 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.

StringWriter and StringReader

oxygen

StringReader is a non-abstract subclass of TextReader.

Similarly, StringWriter is a non-abstract subclass of TextWriter.

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.

A StringReader can be constructed on a string.

A StringWriter, however, cannot be constructed on a string, strings are non-mutable in C#.

Therefore a StringWriter object is constructed on an instance of StringBuilder.

StringWriter example

oxygen

```
public static void Main()
{
    StringBuilder sb = new StringBuilder();   // A mutable string

    using(TextWriter tw = new StringWriter(sb)){
      for (int i = 0; i < 5; i++){
        tw.Write(5 * i);  tw.WriteLine();
        tw.Write(5.5 * i);  tw.WriteLine();
        tw.Write(5555M * i); tw.WriteLine();
        tw.Write(5 * i == 6); tw.WriteLine();}
    }

    Console.WriteLine(sb);
}
```

StringReader example

oxygen

```
public static void Main()
{
    string str = "5" + "\n" +
                 "5,5" + "\n" +
                 "5555,0" + "\n" +
                 "false";

    using(TextReader tr = new StringReader(str))
    {
        int i = Int32.Parse(tr.ReadLine());
        double d = Double.Parse(tr.ReadLine());
        decimal m = Decimal.Parse(tr.ReadLine());
        bool b = Boolean.Parse(tr.ReadLine());

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

StringWriter and StringReader (cont.)

oxygen

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.

Directories and files

Directories and files

oxygen

Now 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 discuss directories.

The File and FileInfo class

oxygen

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.

File is static 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.

Exercise: Try FileInfo

oxygen

Take a look at FileInfo.cs

Create a file called **file-info.txt** and place it in the right directory (Find out where…)

Add some text to **file-info.txt** and run the program.

The File and FileInfo class (cont.)

oxygen

Against our intuition, we realize that the copy of the file still exists after its deletion.

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.

The File and FileInfo class (cont.)

oxygen

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 File and FileInfo class (cont.)

oxygen

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.

Exercise: Refactor to File

oxygen

Take the result of the previous exercise and refactor the code to use File instead of FileInfo.

This is not recommended to be used anywhere, but works well for a classroom assignment.

Directory and DirectoryInfo class

oxygen

The classes DirectoryInfo and Directory are natural directory counterparts of the classes FileInfo and File.

It is recommended 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.

Directory and DirectoryInfo class (cont.)

oxygen

```
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);
}
```

Directory and DirectoryInfo class (cont.)

oxygen

The following shows an overview of the instance properties and methods in 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, ...

Directory and DirectoryInfo class (cont.)

oxygen

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.

Directory and DirectoryInfo class (cont.)

oxygen

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

oxygen

Serialization

Serialization

oxygen

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.

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).

Serialization (cont.)

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 (cont.)

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.

Examples of Serialization in C#

oxygen

See PersonAndDate.cs file

The redundancy is class Person and class Date is introduced on purpose, because it helps us illustrate the serialization control.

In most circumstances we would avoid such redundancy, at least in simple classes.

Try running the code.

Custom serialization

oxygen

In the Person and Date classes, 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.

Custom serialization (cont.)

oxygen

The serialization is controlled by marking some fields as **[NonSerialized]**

The deserialization is controlled by a method marked with the attribute **[OnDeserialized]**

```
[Serializable]
public class Person
{
    private string name;

    [NonSerialized]
    private int age;

    private Date dateOfBirth, dateOfDeath;

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

    [OnDeserialized]
    internal void FixPersonAfterDeserializing(
        StreamingContext context)
    {
        age = Date.Today.YearDiff(dateOfBirth);
    }
    ...
```

Exercise: Add attributes to Date

oxygen

With outset in the previous slide, implement **[NonSerializable]** and **[OnDeserialized]** in Date, so that nameOfDay is calculated on deserilization.

```
[Serializable]
public class Person
{
    private string name;

    [NonSerialized]
    private int age;

    private Date dateOfBirth, dateOfDeath;

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

    [OnDeserialized]
    internal void FixPersonAfterDeserializing(
        StreamingContext context)
    {
        age = Date.Today.YearDiff(dateOfBirth);
    }
    ...
```

Considerations about Serialization

oxygen

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

Serialization and Alternatives

oxygen

- 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 of this course
- Objects in Document databases
  - More or less serialization
  - More performant, optimized structure.
  - Not a topic of this course

Patterns and techniques

Decorator pattern

oxygen

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.

The extensions we deal with consist of adding additional code to the existing methods of C.

Decorator pattern (cont.)

oxygen

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.

A message from Client to an instance of ConcreteComponent is passed through instances of ConcreteDecorator by means of delegation.

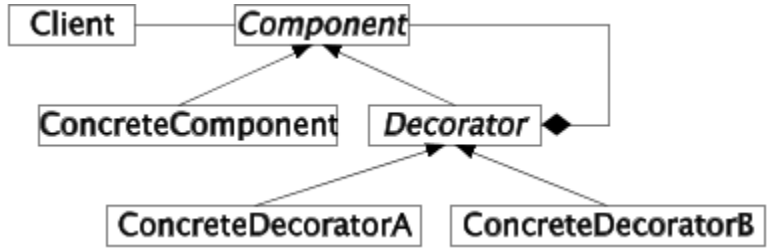
Decorator pattern (cont.)

oxygen

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 be obtained by use of the class hierarchy shown below



Decorator pattern (cont.)

oxygen

On the previous slide 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, because a Decorator can act as a stand in for a ConcreteComponent.

Component: Defines the common interface of participants in the Decorator pattern

Decorator: References another Component to which it delegates responsibilities

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

Decorator pattern and Streams

The Decorator discussion before 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**

Decorator pattern and Streams (cont.)

oxygen

The idea behind the decoration of class FileStream 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).

Decorator pattern and Streams (cont.)

oxygen

```
public static void Main(string[] args)
{
    byte[] buffer;
    long originalLength;

    // Read a file, arg[0], into buffer
    using (Stream infile = new FileStream(args[0], FileMode.Open))
    {
        buffer = new byte[infile.Length];
        infile.Read(buffer, 0, buffer.Length);
        originalLength = infile.Length;
    }

    // Compress buffer to a GZipStream
    Stream compressedzipStream =
        new GZipStream(
            new BufferedStream(
                new FileStream(
                    args[1], FileMode.Create),
                128),
            CompressionMode.Compress);
    compressedzipStream.Write(buffer, 0, buffer.Length);
    compressedzipStream.Close();

    // Report compression rate:
    Console.WriteLine("CompressionRate: {0}/{1}",
        MeasureFileLength(args[1]),
        originalLength);
}
```

```
public static long MeasureFileLength(string fileName)
{
    using (Stream infile = new FileStream(fileName, FileMode.Open))
        return infile.Length;
}
```

oxygen

Questions?