Poul Klausen

C# 5

Assemblies, class libraries and IO

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 5: ASSEMBLIES, CLASS LIBRARIES AND IO

SOFTWARE DEVELOPMENT

C# 5: Assemblies, class libraries and IO: Software Development
1st edition

ISBN 978-87-403-3508-8

CONTENTS

FOREWORD

This book describes topics related to object oriented programming not described in the previous two books. The book starts with an introduction to binary operations, but the main topic is assemblies and including class libraries. Both private and shared assemblies are dealt with and especially how to create and install shared assemblies on the machine. The book also introduces reflection and how, at runtime, to retrieve information about an assembly's type and members and how to instantiate objects dynamic at runtime without knowing there types. There is also a chapter on IO, which in particular adds concepts of serialization that are not included in the previous books. Finally, the book ends with an appendix on binary and hexadecimal numbers.

The previous book deals with object-oriented programming in C# and including the core concepts such as classes and interfaces, and this book is in many ways a continuation such that the book describes a number of topics that were not accommodated in the previous book. The book describes generic types and in relation to that collection classes, among other things, but there are also a number of concepts that are basic prerequisites for C# programmers and including delegates and lambda and other concepts that may not be absolutely necessary but useful to know. After reading this book and working on the book's examples and exercises, you should have a good background for writing programs with C# as your programming language.

As the title says this series of books deals with software development, and the goal is to teach the reader how to develop applications in C#. It can be learned by reading about the subject and by studying complete sample programs, but most importantly by yourself to do it, and write your own programs from scratch. Therefore, an important part of the books is exercises and problems, where the reader has to write programs that correspond to the substance being treated in the books. All books in the series is built around the same skeleton and will consist of text and examples, exercises and problems that are placed in the text where they naturally belongs. The difference between exercises and problems is that the exercises largely deals with repetitions of the substance that is presented in the text, and furthermore it is relatively accurately described what to do. Problems are in turn more loosely described, and are typically a little bigger and there is rarely any clear best solution. These are books to be read from start to finish, but the many code examples, including exercises and problems plays a central role, and it is important that the reader predict in detail studying the code for the many examples and also solves the exercises and problems or possibly just studying the recommended solutions.

All books ends with a larger sample program, which focus primarily is on the process and an explanation of how the program is written. On the other hand appears the code only to a limited extent, if at all, and the reader should instead study the finished program code, perhaps while testing the program. In addition to show the development of programs that are larger than the examples, which otherwise is presented, the aim of the concluding examples also is to show program examples from varying fields of application.

All sample programs are developed and tested on a Windows machine, and the development tool is Visual Studio. Therefore, you must have Visual Studio installed and Microsoft provides a free version that is fully adequate. Visual Studio is an integrated development environment that provides all the tools needed to develop programs. I do not want to deal with the use of Visual Studio, which is quite straightforward, and it is a program that you quickly become familiar with.

Finally a little about what the books are not. It is not “a how to write” or for that matter a reference manuals to C#, but it is as the title says books on software development. It is my hope that the reader when reading the books and through the many examples can find inspiration for how to write good programs, but also can be used as a source collection with a number of examples of solutions to concrete everyday programming problems that you regularly face as a software developer.

1 INTRODUCTION

The last two books have dealt with the basic concepts of the programming language C# and this book is an immediate continuation, primarily with emphasis on

1. binary operators
2. assemblies
3. reflection
4. IO

The smallest unit on a computer is a bit which represents one of two values 0 or 1. All variables and objects consists of a number of bits, and how many and how this bits are used is determined by the datatype. Generally, the individual bits are not directly accessible, nor is it often necessary, but the examples exist. The most programming languages, including C#, therefore have special operators that allow the individual bits in a variable to be manipulated. Chapter two presents these operators and give examples on how to use the operators. The chapter assumes that you are familiar with binary and hex numbers and if not, you can read the book's appendix which gives an introduction to binary numbers and the data presentation on a computer.

When you in Visual Studio compile a project the result is an exe file or a dll, and then a program or a class library. It is a binary file that contains compiled classes, and such a file is called an assembly. A class library is a binary file containing one or more namespaces with compiled classes ready for use in programs, and you can under *References* in Visual Studio set a reference to a dll that contains a class library which means that all the classes can be used in your program. The books chapter 3 deals with assemblies and including how to create and use an assembly and associated with how to write your own class libraries.

When you compile a class, the compiled class contains the binary instructions for the class's methods, but the class's assembly also contains meta-data which are data about the class. This means that a program at runtime can retrieve information about a class's properties, including variables and methods, and you can even instantiate objects of a class without knowing the class name beforehand. This technique is called reflection and can be briefly described as the ability to determine a type's properties at runtime. Reflection is the topic of chapter 5.

IO and file processing are briefly discussed in the book C# 1, however, limited to how to read and write to a text file as well as how to serialize objects to a file. In practice, this is also the most important thing, but there is still a lot more to know, and that is the topic of Chapter 6.

2 BINARY OPERATORS

A data element (*int*, *double*, *string*, etc.) consists of bytes which are somehow the smallest unit on a machine. For example you specify how many gigabytes of RAM a machine has, and you can consider a machine's memory as a large array, where each element is a byte. As an example, an *int* fills 4 bytes while a *double* fills 8 bytes. A byte consists of 8 bits. If you do not have complete control over the binary numbers and the binary operations, refer to the book's appendix.

C# has a number of bit-level operators. The operators mentioned so far operate on simple types (*char*, *int*, *long*, *double*, etc.) that always consists of a whole number of bytes. However, there is a family of operators acting on the individual bits in one or more subsequent bytes. The following operators are available:

Operator	Used as
~	binary complement
&	binary and
|	binary or
^	binary xor
<<	left shift
>>	right shift

However, it should be noted that if the type of the left argument is an *int* or *long*, >> is an arithmetic shift so that it is the left bit that is inserted, but if the type is *uint* or *ulong* it is a regular right shift.

If for example

```
a = 01100111, b = 11001110          (binary)
```

are

```
a & b   = 01000110
a | b   = 11101111
a ^ b   = 10101001
~a      = 10011000
a << 2  = 10011100
a >> 3  = 00001100
```

You should note that a left shift is the same as multiply by 2, while a right shift is the same as division by 2.

The following method (the program *BinaryProgram*) shows how to use the binary operators:

```
static void Test1()
{
  AndOrXor();
  Complement();
  Left();
  Right();
}

static void AndOrXor()
{
  int a = 103;
  int b = 206;
  Console.WriteLine(Bitstring(a));
  Console.WriteLine(Bitstring(b));
  Console.WriteLine(Bitstring(a & b));
  Console.WriteLine(Bitstring(a | b));
  Console.WriteLine(Bitstring(a ^ b));
  Console.WriteLine();
}

static void Complement()
{
  int a = 12345678;
  Console.WriteLine(Bitstring(a));
  Console.WriteLine(Bitstring(~a));
  Console.WriteLine();
}

static void Left()
{
  int a = 12345678;
  Console.WriteLine(Bitstring(a));
  Console.WriteLine(Bitstring(a << 2));
  Console.WriteLine(Bitstring(a << 16));
  Console.WriteLine();
}

static void Right()
{
  int a = 12345678;
  int b = -12345678;
```

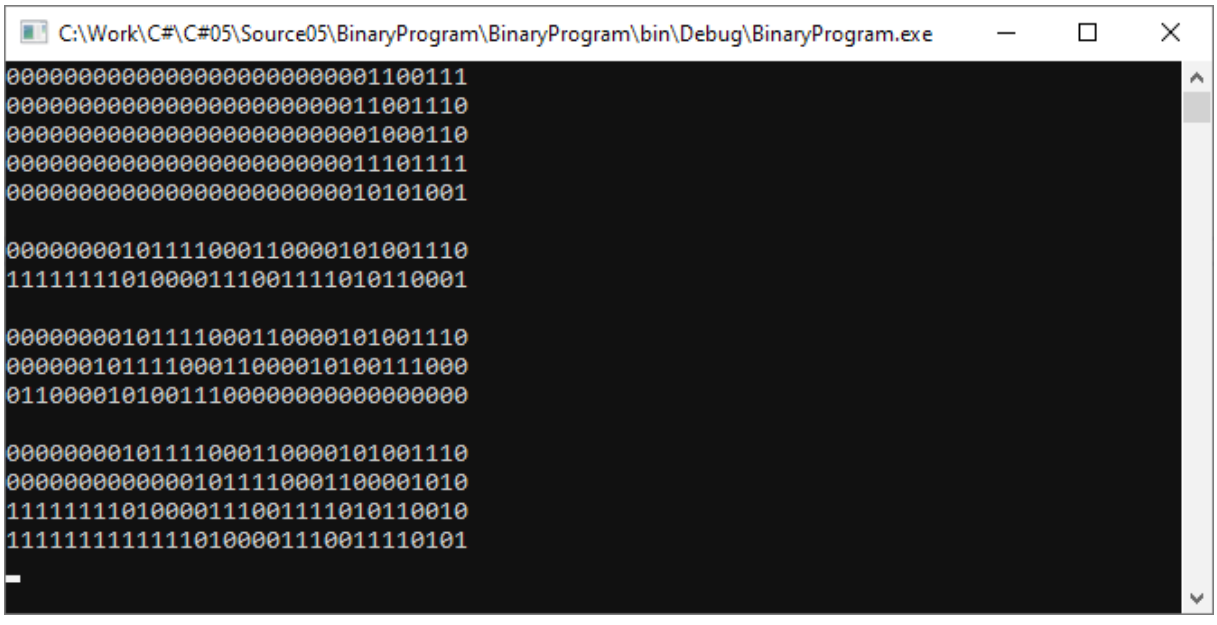
```
  Console.WriteLine(Bitstring(a));
  Console.WriteLine(Bitstring(a >> 5));
  Console.WriteLine(Bitstring(b));
  Console.WriteLine(Bitstring(b >> 5));
}

static string Bitstring(int a)
{
  StringBuilder builder = new StringBuilder(32);
  for (uint b = 0x80000000; b != 0; b >>= 1)
    builder.Append((a & b) == 0 ? '0' : '1');
  return builder.ToString();
}
```

I think most of the methods are self explanatory, but you should note the method *Bitstring()*. The method has an *int* as parameter and forms a string with the bit pattern that the number consists of, and here you must note that the variable *b* has the type *unit* and then the result of

```
b >> 1 and a & b
```

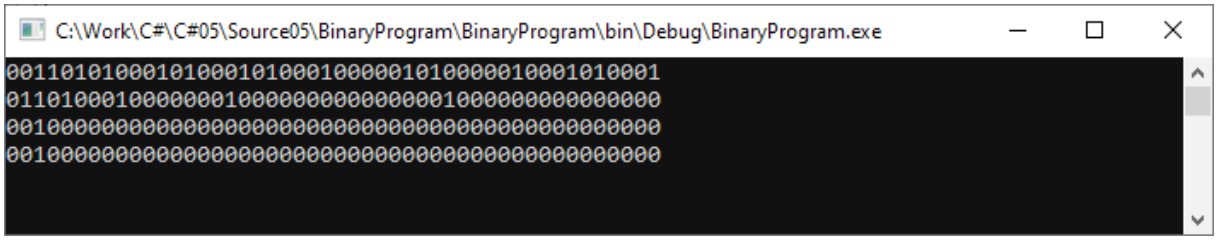
is a *unit*. In particular, also note the method *Right()* method, which shows that >> is an arithmetic shift on elements of the type int.

If the method is performed, you get the result as shown below:



C# has a class called *BitArray* which represents a bit pattern of arbitrary length and the following method shows how to use this class (the class is defined in the namespace *System.Collections*):

```
static void Test2()
{
  BitArray b1 = new BitArray(48);
  b1[2] = true;
  b1[3] = true;
  b1[5] = true;
  b1[7] = true;
  b1[11] = true;
  b1[13] = true;
  b1[17] = true;
  b1[19] = true;
  b1[23] = true;
  b1[29] = true;
  b1[31] = true;
  b1[37] = true;
  b1[41] = true;
  b1[43] = true;
  b1[47] = true;
  Print(b1);
  BitArray b2 = new BitArray(48);
  b2[1] = true;
  b2[2] = true;
  b2[4] = true;
  b2[8] = true;
  b2[16] = true;
  b2[32] = true;
  Print(b2);
  Print(b2.And(b1));
  Print(b2);
}

static void Print(BitArray arr)
{
  foreach (bool b in arr) Console.Write(b ? '1' : '0');
  Console.WriteLine();
}
```



There is not much to explain, but you should note the following:

- The constructor has a parameter that specifies how many bits a *BitArray* can hold, here it is 48. If you try to reference a bit with an index greater than 47, you get an exception. The class also has other constructors.
- A bit is represented as a *bool* where *true* corresponds to 1 while *false* corresponds to 0. After creating a bit array (as in this example), all bits are 0 (*false*).
- The class overrides the index operator, and here you get an exception if you try to index outside the array boundaries. In particular, note that a *BitArray* does not automatically expands as a list, but that it has the size (capacity) assigned when created.
- Note the next last statement that executes an *and* operation. The method changes the current object (the result is an *and* between the current object and the parameter) and it returns a reference to itself. It is in this case, *bm2* is changed after the method is executed. There are similar methods for *or* and *xor*.
- Finally, note the method *Print()* and that you can traverse a *BitArray* with *foreach*.

2.1 A BITMAP

I will try to write a class that is an alternative to a *BitArray*. There is absolutely no practical justification for the class and the example is included only to give examples of the use of the bit operators, but also to show the use of much of the substance treated in the previous book.

The code fills a portion, and I will only show parts of the code, but will otherwise refer to the full code on the book's website.

The class is defined as:

```
public class Bitmap : IComparable<Bitmap>, IEnumerable<int>
{
  delegate ulong BitOperation(ulong u1, ulong u2);

  private static ulong[] mask1 =
  {
    0x8000000000000000, 0x4000000000000000, 0x2000000000000000, ....
    ....
  };

  private static ulong[] mask2 =
  {
    0x7FFFFFFFFFFFFFFF, 0xBFFFFFFFFFFFFFFF, 0xDFFFFFFFFFFFFFFF, ....
    ....
  };

  private ulong[] map;
  public int Size { get; private set; }
```

The class implements the interface *IComparable<Bitmap>* and must then override the method *CompareTo()*, and the class also implements the iterator pattern. Next, a delegate is defined, which I will return to in a little while.

The next is two static arrays each with 64 (showing only three elements in each array) 64-bit bit patterns. The first array contains 64 patterns consisting of exactly one 1-bit in each of the 64 possible places, while the remainders are 0. The second array is similar, but it has exactly one 0-bit in each of the 64 possible places, while the rest are 1.

The data structure itself is represented as an array of the type *ulong* and thus as an array of 64 bit patterns. In addition, there is a property that keeps track of how many bits are available. For example, if you creates a 300-bit bitmap, *Size* will have a value of 300, while the array *map* will have space for 5 items. Thus, there are 20 bits that are not used.

The class has the following constructor that has a parameter that specifies the number of bits to accommodate. The constructor must primarily calculate the size of the array:

```
public Bitmap(int size)
{
  if (size < 1) throw new ApplicationException("The size of the
  bitmap... ");
  int n = size / 32;
  if (size % 32 > 0) ++n;
  map = new ulong[n];
  Size = size;
}
```

The class must override the index operator that way, that you can read and change a particular bit in an element in the *map*. It is here the two static arrays should be used:

```
public bool this[int n]
{
  get
  {
    if (n >= Size) throw new ApplicationException("Illegal index");
    int q = n / 64;
    int r = n % 64;
    return (map[q] & mask1[r]) != 0;
  }
  set
  {
    if (n >= Size) throw new ApplicationException("Illegal index");
    int q = n / 64;
    int r = n % 64;
    if (value) map[q] |= mask1[r]; else map[q] &= mask2[r];
  }
}
```

In particular, you should note how to calculate the bit position to be referenced and how to test with *mask1* whether a particular bit is set. In the *set* part, note how to apply *mask1* to set a bit to 1 and *mask2* to set a bit to 0. These are typical bit operations.

The class has a *private static* method:

```
private static Bitmap Operation(Bitmap bm1, Bitmap bm2, BitOperation opr)
{
  if (bm1.Size != bm2.Size) throw new ApplicationException("Bitmaps
  must ... ");
  Bitmap bm = new Bitmap(bm1.Size);
  for (int i = 0; i < bm.map.Length; ++i) bm.map[i] = opr(bm1.map[i],
  bm2.map[i]);
  return bm;
}
```

It uses the above delegate, which defines an operation that can be performed on two 64-bit patterns. Note in particular that the method requires the two bitmaps to be the same length, otherwise they are considered incompatible and the method raises an exception. The class *BitArray* works the same way. The method is used to implement the three basic bit operations (*and*, *or* and *xor*), and as an example is *and* shown below:

```
public static Bitmap operator &(Bitmap bm1, Bitmap bm2)
{
  return Operation(bm1, bm2, (u1, u2) => u1 & u2);
}
```

You should note that it is an operator override, and then you should note how to call the method *Operation()*, where you specify the last parameter with a lambda expression.

The class also overrides the comparison operators. Here I will refer to the finished code as they are all trivial and consist solely of a call to *CompareTo()*.

The class also overrides the shift operators. An example is the left shift below:

```
public static Bitmap operator <<(Bitmap bm, int n)
{
  Bitmap b = new Bitmap(bm.Size);
  if (n < bm.Size)
  {
    int q = n / 64;
    int r = n % 64;
    if (r == 0) for (int i = q, j = 0; i < bm.map.Length; ++i, ++j)
      b.map[j] = bm.map[i];
    else for (int i = q, j = 0; i < bm.map.Length; ++i, ++j)
    {
      if (j > 0) b.map[j - 1] |= (bm.map[i] >> 64 - r);
      b.map[j] = bm.map[i] << r;
    }
  }
  return b;
}
```

The second parameter *n* indicates the number of places to be shifted. If *n* is greater than the number of bits, it corresponds to the result being 0 (0 bits only). The calculations primarily consist of deciding where to start from, and here again there are two cases corresponding to whether to change a whole number of 64-bit words or to switch across the word boundary.

The class finally implements an implicit type of cast to a *BitArray*:

```
public static implicit operator BitArray(Bitmap bm)
{
  BitArray arr = new BitArray(bm.Size);
  for (int i = 0; i < bm.Size; ++i) if (bm[i]) arr[i] = true;
  return arr;
}
```

There is not much to explain and the implementation simply consists of a pass through the current bitmap. There is also an implicit type of cast that goes the other way and converts a *BitArray* into a *Bitmap*.

When working with binary data (often in connection with data communication), it will typically take the form of a byte array. The constructor in *BitArray* has a variant so you can create a *BitArray* based on a byte array. Instead, I have given my class an explicit type of cast:

```
public static explicit operator Bitmap(byte[] arr)
{
  Bitmap bm = new Bitmap(arr.Length * 8);
  for (int i = 0; i < arr.Length; ++i)
  {
    byte b = 0x80;
    for (int j = 0; j < 8; ++j, b >>= 1)
      if ((arr[i] & b) != 0) bm[i * 8 + j] = true;
  }
  return bm;
}
```

which converts a byte array into a *Bitmap* and there is also a type of cast that goes the other way. Note that this time this is an explicit type cast as it does not necessarily make sense to convert a byte array to a Bitmap, here the programmer should decide.

Below is a test method where you should note how the operations are used:

```
static void Test3()
{
  int n = 148;
  Bitmap bm1 = new Bitmap(n);
  bm1[79] = true;
  bm1[80] = true;
  bm1[n - 1] = true;
  Bitmap bm2 = new Bitmap(n);
  bm2[78] = true;
  bm2[80] = true;
  bm2[n - 2] = true;
  Console.WriteLine(bm1);
  Console.WriteLine(bm2);
  Console.WriteLine(bm1 & bm2);
  Console.WriteLine(bm1 | bm2);
  Console.WriteLine(bm1 ^ bm2);
  Console.WriteLine(~bm1);
  Console.WriteLine(bm1 << 5);
  Console.WriteLine(bm1 >> 5);
  Console.WriteLine(bm1 << 69);
  Console.WriteLine(bm1 >> 69);
  BitArray arr = bm1;
  Print(arr);
  Bitmap bm3 = arr;
  Console.WriteLine(bm3);
  byte[] b = (byte[])bm1;
  Print(b);
  Bitmap bm4 = (Bitmap)b;
  Console.WriteLine(bm4);
  bm2 = ~bm2;
  foreach (int bit in bm2) Console.Write(bit); Console.WriteLine();
}
```

EXERCISE 1: BIT OPERATIONS

The goal of this exercise is to write some more methods that work on the individual bits in an integer, partly for the sake of the exercise, but also because they can be useful in practice. Start with a new console application project that you can call *BinaryProgram*. To this project you must add an implement the following class:

```
public class Binary
{
  /// <summary>
  /// Method which convert an integer to a string representing an integer
  /// as a binary number. The integer has the type byte, sbyte, uint,
      int,
  /// ulong or long and is then a signed or unsigned integer on 8, 32 or
  /// 64 bits.
  /// </summary>
  /// <param name="t">The integer to be converted to a string</param>
  /// <returns>The string representation of the integer</returns>
  public static string ToBinString(byte t) { .. }
  public static string ToBinString(sbyte t) { .. }
  public static string ToBinString(uint t) { .. }
  public static string ToBinString(int t) { .. }
  public static string ToBinString(ulong t) { .. }
  public static string ToBinString(long t) { .. }

  /// <summary>
  /// Method which convert an integer to a string representing an integer
  /// as a hex number. The integer has the type byte, sbyte, uint, int,
  /// ulong or long and is then a signed or unsigned integer on 8, 32
or
  /// 64 bits.
  /// </summary>
  /// <param name="t">The integer to be converted to a string</param>
  /// <returns>The string representation of the integer</returns>
  public static string ToHexString(byte t) { .. }
  public static string ToHexString(sbyte t) { .. }
  public static string ToHexString(uint t) { .. }
  public static string ToHexString(int t) { .. }
  public static string ToHexString(ulong t) { .. }
  public static string ToHexString(long t) { .. }

  /// <summary>
  /// Method that returns the value of a bit in an integer at
  /// position n. The position starts from left, where the left
  /// most bit has position 0. The integer has the type byte,
  /// sbyte, uint, int, ulong or long and is then a signed or
```

```
/// unsigned integer on 8, 32 or 64 bits.
/// </summary>
/// <param name="t">The integer value</param>
/// <param name="n">The bit position</param>
/// <returns>0 or 1 as the value at bit position n</returns>
public static int GetBit(byte t, int n) { .. }
public static int GetBit(sbyte t, int n) { .. }
public static int GetBit(uint t, int n) { .. }
public static int GetBit(int t, int n) { .. }
public static int GetBit(ulong t, int n) { .. }
public static int GetBit(long t, int n) { .. }

/// <summary>
/// Method that sets the value of bit position n in an integer
/// to the value bit regardless of what value the bit in question
/// had before. The position starts from left, where the left
/// most bit has position 0. The integer has the type byte,
/// sbyte, uint, int, ulong or long and is then a signed or
/// unsigned integer on 8, 32 or 64 bits.
/// The method returns the integer after the bit is changed.
/// </summary>
/// <param name="t">The integer value</param>
/// <param name="n">The bit position</param>
/// <returns>The value after the bit is changed</returns>
public static sbyte SetBit(sbyte t, int n, int bit) { .. }
public static byte SetBit(byte t, int n, int bit) { .. }
public static int SetBit(int t, int n, int bit) { .. }
public static uint SetBit(uint t, int n, int bit) { .. }
public static long SetBit(long t, int n, int bit) { .. }
public static ulong SetBit(ulong t, int n, int bit) { .. }

/// <summary>
/// Ror() is a method which performs a right rotation on n bits on
    an integer
/// value with the type sbyte, byte, int, uint, long or ulong and then a
/// signed or unsigned integer on 8, 32 or 64 bits.
/// </summary>
/// <param name="t">The integer which bits should be rotated</param>
/// <param name="n">Number of bits to rotate</param>
/// <returns>The integer rotated n bits right</returns>
public static byte Ror(byte t, int n) { .. }
```

```
  public static sbyte Ror(sbyte t, int n) { .. }
  public static uint Ror(uint t, int n) { .. }
  public static int Ror(int t, int n) { .. }
  public static ulong Ror(ulong t, int n) { .. }
  public static long Ror(long t, int n) { .. }

  /// <summary>
  /// Rol() is a method which performs a left rotation on n bits on
      an integer
  /// value with the type sbyte, byte, int, uint, long or ulong and then a
  /// signed or unsigned integer on 8, 32 or 64 bits.
  /// </summary>
  /// <param name="t">The integer which bits should be rotated</param>
  /// <param name="n">Number of bits to rotate</param>
  /// <returns>The integer rotated n bits left</returns>
  public static byte Rol(byte t, int n) { .. }
  public static sbyte Rol(sbyte t, int n) { .. }
  public static uint Rol(uint t, int n) { .. }
  public static int Rol(int t, int n) { .. }
  public static ulong Rol(ulong t, int n) { .. }
  public static long Rol(long t, int n) { .. }
}
```

Once you have written the class, it is important that you test all the methods, there are 36 in total. To write test methods you can get problems, if you for example try to type cast a negative number to an unsigned. You can solve this problem if you place the type casts in an *unckecked* block as for example:

```
static void Test01()
{
  unchecked
  {
    Print1((byte)65);
    Print1((sbyte)65);
    Print1((byte)-65);
    Print1((sbyte)-65);
  }
}
```

Above I have shown a class *Bitmap* as an alternative to the C# class *BitArray*. This class requires two *Bitmaps* to have the same length to be compatible. For example you cannot perform an *and* operation on two bitmaps if they have different length. It doesn't always make sense, and you must to write another version of the class *Bitmap* that allows you to perform operations on bitmaps of different lengths.

There are some problems to write the class and for example what I should means if you for the indexer use an index which refers to a position outside the length of the current bitmap. If it happens the bitmap must be expanded to make room for the new position, and to avoid too many expansions, try to use a strategy where you double the capacity every time you expand the bitmap.

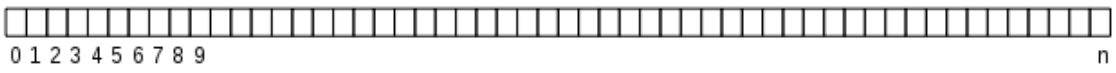
Another problem is what should happens, if you perform a binary operation on two bitmaps of different length. Here you should perceive missing bits as bits that are 0.

When you have written the class you must test it and you should be sure also to test the class on bitmaps of different length.

PROBLEM 1: ERATOSTHENES

As an example of using a bitmap, you must write a program that uses Eratosthenes's algorithm to determine the primes. The idea is the following:

In order to determine all prime numbers that are less than or equal to n, start by defining a bitmap with n+1 elements:



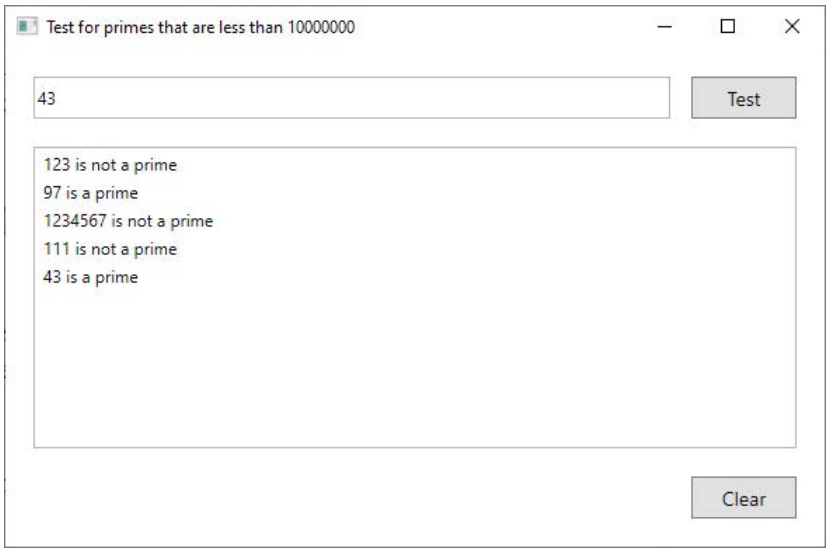
In the beginning all positions are 0. You then starts with putting a mark (a 1 bit) in position 0 and position 1 as well as all the positions where the index is divisible by 2 - though not position 2:

In this run has been set a mark in all the even numbers greater than 2, as they are not primes. You then repeats it all, but this time you put a mark on the positions where the index is divisible by 3 - though not position 3:

All positions where the index has 3 as the prime factor is now marked. Next time you repeat it all with the number 5 (the first position after 3 that is 0). All positions where the index has 5 as the prime factor is marked:

To resume. Next time start with the first position, which is not marked (it is position 7) and it is a prime, since it would otherwise be marked by a previous cycle. When you have been through it all, all positions with an index that is not prime are checked.

You must now write a program where the user can enter a number:



When you click the *Test* button, the application must insert a line in the list box that shows whether the number is a prime number. To determine if this is the case, the program must use the above algorithm and the class *Bitmap* from above. You can improve the algorithm a bit by observing the following:

1. If a number has a prime factor (a prime number that divides the number), it or another prime factor must be less than or equal to the square root of the number, and the above iteration can therefore stop when the starting index is greater than square root of the number.

2. If you have reached the index that is not marked in a previous cycle, it is a prime, and you have to mark all the numbers that has the number as divisor, and then it actually is enough to start from this index, because the position else would be marked by a previous cycle.
3. It is not necessary in the bitmap to track other than the odd numbers, because all the even numbers greater than 2 are not primes.

Eratosthenes algorithm is actually a very effective prime method, but the problem is of course that the bitmap take room in memory - even if you use the above observations.

EXERCISE 2: ENCODING TEXT

Create a console application project that you can call *Encoding*. Add the following two test methods, where the first write a text to a file, while the other reads the text again:

```
private static void Test1()
{
  try
  {
    StreamWriter writer = new StreamWriter("C:\\Temp\\Test1");
    writer.Write("Søren Sørensen\nPistolStræde 19\n6666 Borremose");
    writer.Close();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}

private static void Test2()
{
  try
  {
    StreamReader reader = new StreamReader("C:\\Temp\\Test1");
    for (string line = reader.ReadLine(); line != null; line = reader.
    ReadLine())
      Console.WriteLine(line);
    reader.Close();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
  Console.WriteLine();
}
```

Test the two methods from the *Main()* method. Do they have the expected result?

Write two other test methods:

```
private static void Test3()
{
  try
  {
    StreamWriter writer = new StreamWriter(new FileStream("C:\\Temp\\
    Test2",
      FileMode.Create), System.Text.Encoding.UTF8);
    writer.Write("Søren Sørensen\nPistolStræde 19\n6666 Borremose");
    writer.Close();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}

private static void Test4()
{
  try
  {
    StreamReader reader = new StreamReader("C:\\Temp\\Test2",
      System.Text.Encoding.UTF8);
    for (String line = reader.ReadLine(); line != null; line = reader.
    ReadLine())
      Console.WriteLine(line);
    reader.Close();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
  Console.WriteLine();
}
```

Note that the difference is that this time the file is opened in a different manner (with an additional parameter), and that the file has another name. Test also these two methods from *Main()* and note that you get the expected result.

Write two more test methods (which you can call *Test5()* and *Test6()*), where the differences only is that the file has a different name (this time *test3*) and a second parameter concerning encoding:

```
StreamWriter writer = new StreamWriter(new FileStream("C:\\Temp\\Test3",
  FileMode.Create), System.Text.Encoding.Unicode);

StreamReader reader = new StreamReader(new FileStream("C:\\Temp\\Test3",
  FileMode.Open), System.Text.Encoding.Unicode);
```

Test also these two methods. Write finally two test methods *Test7()* and *Test8()*, where the files are opened as follows:

```
StreamWriter writer = new StreamWriter(new FileStream("C:\\Temp\\Test4",
  FileMode.Create), System.Text.Encoding.GetEncoding("ISO-8859-1"));

StreamReader reader = new StreamReader(new FileStream("C:\\Temp\\Test4",
  FileMode.Open), System.Text.Encoding.GetEncoding("ISO-8859-1"));
```

and try also these two methods. If you uses *File Explorer* to examine the four files, you can see how much they fills on the disk:

- *test1* 48 bytes
- *test2* 51 bytes
- *test3* 92 bytes
- *test4* 45 bytes

Can you explains this 4 numbers?

EXERCISE 3: FLOATING POINTS

Create a project that you can call *FloatingPoint*. Add the class *Binary* from exercise 1 to this project. Remember to change the namespace. The class *Binary* has methods *GetBinString()* and *GetHexString()* which returns the binary and hex representations of integer values as strings. Add similar methods for the types *float* and *double*.

Add the following test method:

```
static void Test1()
{
  Print(-46030.32813F);
  Print(float.MaxValue);
  Print(float.MinValue);
  Print(float.NegativeInfinity);
  Print(float.PositiveInfinity);
  Print(float.Epsilon);
  Print(float.NaN);
}
```

where *Print()* is a method that print the parameter, its and hex binary representation. Check if it fits to what is said in the appendix.

Write a similar test method for the type *double*.

3 ASSEMBLIES

When you write a program and compile it, a binary file is created with the compiled code. This is the code that the runtime system executes when running the program. Under .NET, the file with the compiled code is called an assembly, and in the following I will take a closer look at what an assembly is, what it contains, but also how to write assemblies that are general and can be shared between multiple programs. In addition, I will look at how to configure assemblies using configuration files.

Some of the following is detailed and not absolutely necessary to learn how to program in C#, but conversely, issues like configuration files and shared assemblies are important in practice, at least if you need to write ready-made applications to install on clients' computers.

The current chapter is quite extensive and is organized in the following sections:

- Namespaces
- Class libraries
- Structure of an assembly
- Platform independence
- Private assemblies
- Shared assemblies
- Configurations files

3.1 NAMESPACES

Before addressing assemblies, I will look at namespaces a bit more. Primarily to explain how the term attaches to assemblies.

A completed .NET program consists of a family of objects that work together to solve the task to be solved. These objects are created on the basis of types that are primarily struct types or class types. Seen from the programmer, everything is thus types that are either written by the programmer and are part of the source code, or are types that come from the .NET framework (and this applies to the vast majority of types), or they are the types that come from a third party and may be purchased or otherwise are available. All of these types must have a unique name, and to control that no name clashes occurs, types are grouped into namespaces, and a type's full name is therefore the type name preceded by the namespace. For example is

```
System.Console
```

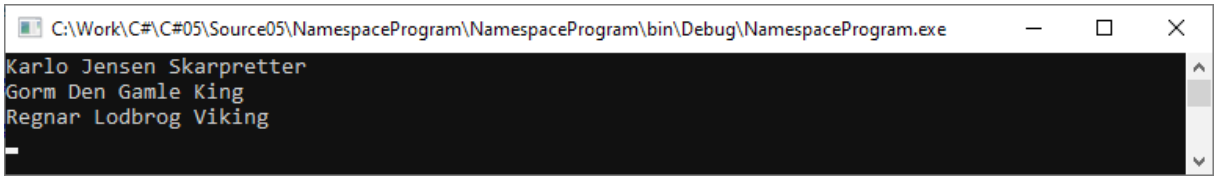
the full name of a type *Console* in the namespace *System*. A namespace is thus a very simple term, which is simply a matter of grouping types under a common name.

The following example shows a program with 5 classes, but where the classes are placed into three namespaces. The goal is to show how to create namespaces yourself, and the classes are all trivial. The program is a usual console application with the following code, where there are three namespaces in the same file:

```
using System;

namespace NamespaceProgram
{
  using NamespaceProgram.People;
  using NamespaceProgram.People.Danish;

  class Program
  {
    static void Main(string[] args)
    {
      Print(new Employee() { Firstname = "Karlo", Lastname = "Jensen",
        Position = "Skarpretter" });
      Print(new King() { Firstname = "Gorm", Lastname = "Den Gamle" });
      Print(new Viking() { Firstname = "Regnar", Lastname = "Lodbrog" });
      Console.ReadLine();
    }

    static void Print(Employee e)
    {
      Console.WriteLine("{0}  {1}  {2}",  e.Firstname,  e.Lastname,  e.
Position);
    }
  }
}

namespace NamespaceProgram.People
{
  public class Person
  {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
  }
```

```
  public class Employee : Person
  {
    public string Position { get; set; }
  }
}

namespace NamespaceProgram.People.Danish
{
  using NamespaceProgram.People;

  public class King : Employee
  {
    public King()
    {
      Position = "King";
    }
  }

  public class Viking : Employee
  {
    public Viking()
    {
      Position = "Viking";
    }
  }
}
```

If you run the program the result is:



Note first the two classes *Person* and *Employee*, the latter being a specialization of the first. The two classes are trivial, but they are located in a namespace named *NamespaceProgram.People*. The only thing required to create a namespace is to enter the name and the result is, that the full names of the two classes are:

- *NamespaceProgram.People.Person*
- *NamespaceProgram.People.Employee*

There are also two classes called respectively *King* and *Viking*, but they are located in another namespace, a sub namespace for *NamespaceProgram.People* named *NamespaceProgram.People.Danish*. Both classes are specializations of the class *Employee*, but since it is in a different namespace, a using statement is defined for that namespace. The classes are trivial, but the full name of the first is

- *NamespaceProgram.People.Danish.King*

The conclusion is that a namespace alone represents a prefix that is preceded by the name of a type. You should especially note that a namespace has nothing to do with a folder or a directory - it's just a name that is preceded by the name of a type. In practice, types are usually placed in corresponding folders, but it is another matter.

The main program with the *Main()* method is in its own namespace named *NamespaceProgram*. Main() creates objects of the above classes, but in order to reference the classes by their names alone, *using* statements is defined for the classes namespaces.

Below is an expanded version of the above program. The namespace *NamespaceProgram.People* is expanded with a new class. It means that there are two classes called *King*, but it is fine as the classes are in each there namespace. In the program, however, a problem arises, as there are now two things called *King*. That is, there is a name clash and the problem can of course be solved by using the full names of the two *King* classes everywhere. This means that you have to write a lot, something that you can avoid using an alias defined by a *using* statement.

```
using System;

namespace NamespaceProgram
{
  using NamespaceProgram.People;
  using NamespaceProgram.People.Danish;
  using King1 = NamespaceProgram.People.King;
  using King2 = NamespaceProgram.People.Danish.King;

  class Program
  {
    static void Main(string[] args)
    {
      Print(new Employee() { Firstname = "Karlo", Lastname = "Jensen",
        Position = "Skarpretter" });
      Print(new King2() { Firstname = "Gorm", Lastname = "Den Gamle" });
```

```
      Print(new King1("No") { Firstname = "Erik", Lastname = "Blodøkse"
});
      Print(new Viking() { Firstname = "Regnar", Lastname = "Lodbrog"
});
      Console.ReadLine();
    }

    static void Print(Employee e)
    {
      Console.WriteLine("{0}  {1}  {2}",  e.Firstname,  e.Lastname,  e.
Position);
    }
  }
}

namespace NamespaceProgram.People
{
  ...

  public class King : Employee
  {
    public string Country { get; set; }

    public King(string country)
    {
      Position = "King";
      Country = country;
    }
  }
}

namespace NamespaceProgram.People.Danish
{
  using NamespaceProgram.People;

  public class King : Employee
  {
    public King()
    {
      Position = "King";
    }
  }
```

3.2 CLASS LIBRARIES

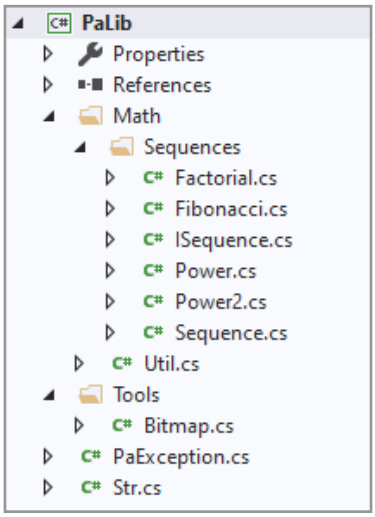
A class library is not a program, but is an assembly that contains compiled types and the types which can be used in programs. The .NET framework consists primarily of class libraries with the classes that we use to write among other C# programs. The file for a class library is an assembly has the extension .dll. In this section I will show how to write a class library. The library should contains 10 types, and it's pretty much all types written in the above books, but here added to a class library, so I have a dll available which can subsequently be used in programs. It is also a class library that later can be expanded (and will be) with several types.

A class library will usually consist of types for a particular purpose. It may be types that relate to a particular application, but it can also be types that are general and the class library then becomes a tool box that makes tools available, tools that are typically needed for the kind of programs a development environment develops. The following class library is of the last kind, and you should think of the library as classes which I often need.

The class library is called *PaLib* and it is created in Visual Studio as a *Class Library (.NET Framework)* project. Visual Studio will automatic add a class which you can delete and then add the classes you needs. In this case the 10 types in the class library must be added to four namespaces:

- PaLib
- PaLib.Tools
- PaLib.Math
- PaLib.Math.Sequences

and then a kind of hierarchy with nested namespaces. A namespace is a name and is not in itself a hierarchy, but to physically place the types separately according to the individual namespaces I have created folders and placed the types accordingly:



The types are as follows:

The namespace *PaLib* has two types. *PaException* is a new exception type:

```
using System;

namespace PaLib
{
  public class PaException : ApplicationException
  {
    public PaException(string message) : base(message)
    {
    }
  }
}
```

and is trivial. The type *Str* is the class from exercise 1 in C# 4. The type *Bitmap* in the namespace *PaLib.Tools* is the class from section 2.1 in this book. The types in the namespace *PaLib.Math.Sequences* are interface and classes from problem 1 in C# 3. Finally, the last type is a new class

```
namespace PaLib.Math
{
  public class Util
  {
    public static long Sqr(int n)
    {
      return ((long)n) * n;
    }

    public static double Sqr(double x)
    {
      return x * x;
    }

    public static bool IsPrime(long n)
    {
      if (n == 2 || n == 3 || n == 5 || n == 7) return true;
      if (n < 11 || n % 2 == 0) return false;
      for (long t = 3, m = (long)System.Math.Sqrt(n) + 1; t <= m; t += 2)
        if (n % t == 0) return false;
      return true;
    }
  }
}
```

The class has three static methods which are methods which I have used many times and thus methods that can be used in other contexts. Note the *namespace* statement. In all the other types the *namespace* statements are changed to the right name.

If you build the project the result is a binary file *PaLib.dll* which is the class library with the above types ready for use in other programs.

EXERCISE 4: ERATOSTHENES AGAIN

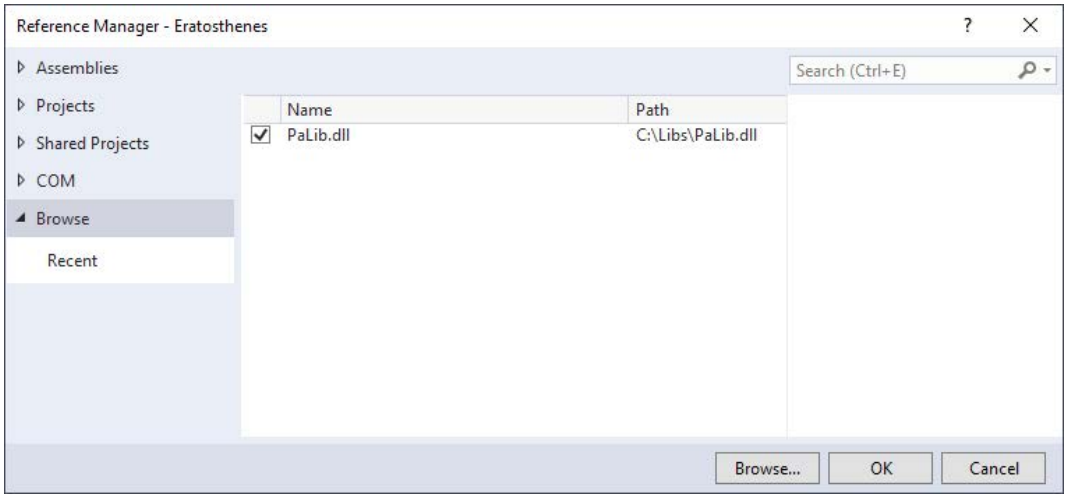
Create a copy of your project from problem 1. Open the copy in Visual Studio and remove the class *Bitmap*. Set a reference to the new class library:

- In Solution Explorer right click on *References*
- Select *Add reference*
- Click on the tab *Browse* and the click the button *Browse* (see below)
- Browse to the location for the class library and select it

When you click *OK* the project has a reference to the class library and can use the class *Bitmap*. For it to work, add a *using* statement

```
using PaLib.Tools;
```

and then the program can run again.



3.3 THE STRUCTURE OF AN ASSEMBLY

If you compile a simple program, the result is a single exe file. It is not a traditional exe file with binary code translated into the specific platform, but instead a so-called assembly, which contains managed code and thus code that is compiled, but must be performed by a runtime system. A program can very well consist of or use multiple assemblies, which if necessary are dlls. Put slightly differently, an assembly is a self-documenting binary file that has the extension *exe* or *dll* and contains types and possibly other resources. An assembly basically consists of six parts:

- a standard Windows header
- a CLR header that marks the file as managed code
- the CIL code
- type metadata
- an assembly manifest
- embedded resources

The Windows header is just a simple identification that identifies the file as executable code to the operating system and whether it is a console or a Windows program. The CLR header is a compiler-generated header that defines the layout of the assembly against the CLR in the form of a C-like structure. As a programmer, you rarely if ever have to take an interest in these two headers, but they are aimed solely at respectively operating system and CLR.

Central is the CIL code, which contains the binary code translated into a platform neutral language. When the CIL code is executed on a specific machine, it is interpreted on the fly by the just in time compiler and translated into the specific platform and CPU instructions.

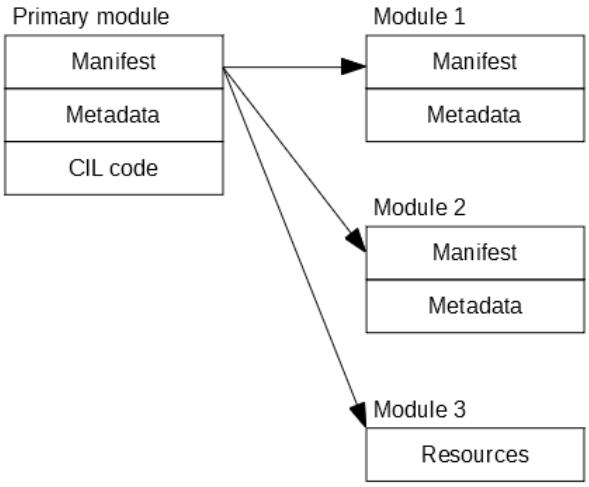
Metadata is a complete description of each type in the current assembly as well as all external references used by the assembly. This metadata is generated by the compiler and describes all attributes, methods, properties and events that the type defines. The CLR uses this metadata to determine a type's location and properties in the assembly.

Finally, the manifest documents all external assembly references, and it is the manifest that contains versioning information.

As the last thing an assembly can contain embedded resources as icons, images and so on.

In this context, a module is a term for a .NET file of some kind. An assembly can consist of one or more modules, and in that light an assembly can be perceived as a unit that can be versioned and installed on a given machine. In most cases, an assembly consists of a single module (file) and you are talking about a single file assembly. Here you can think of an assembly as a file with the following layout:

A multi file assembly consists of several modules, and the most important difference is that when a type is referenced, only the file containing the type is loaded. It has special interest in remote references. Technically, the individual modules are not linked together in a single entity, but the manifest contains references to the individual modules. The manifest may be stored in its own file, but more typically the manifest is part of the primary module. An example of the layout of a multi file assembly could therefore be as shown below.



Each assembly has a version identifier that applies to all types and resources of all modules in that assembly. On this basis, the runtime system can ensure that a client loads the correct assembly based on a versioning policy. An identification identifier consists of two parts: a text (called informational version) and a numeric value (called compatibility version). The last one could, for example be 1.0.70.3 where the first number is major version number, the second minor version number, the third is the build number and the last is the current revision number. However, the last two are not necessary.

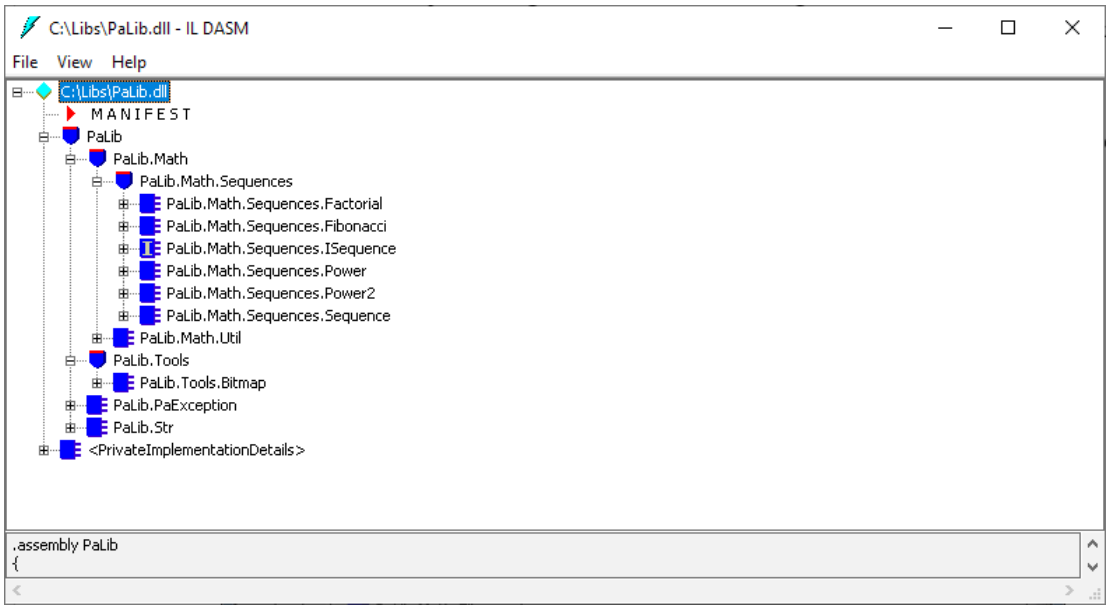
As an important feature of assemblies, multiple versions of the same assembly can be installed on the same machine. In an application configuration file you can control which version of a given assembly to load. It's the technique of solving the so-called dll hell.

3.4 ILDASM

Ildasm is a tool that comes with Visual Studio and can be used to analyze an assembly. The easiest way is to open a .NET prompt and start the program from there, as you have the right path for the program.

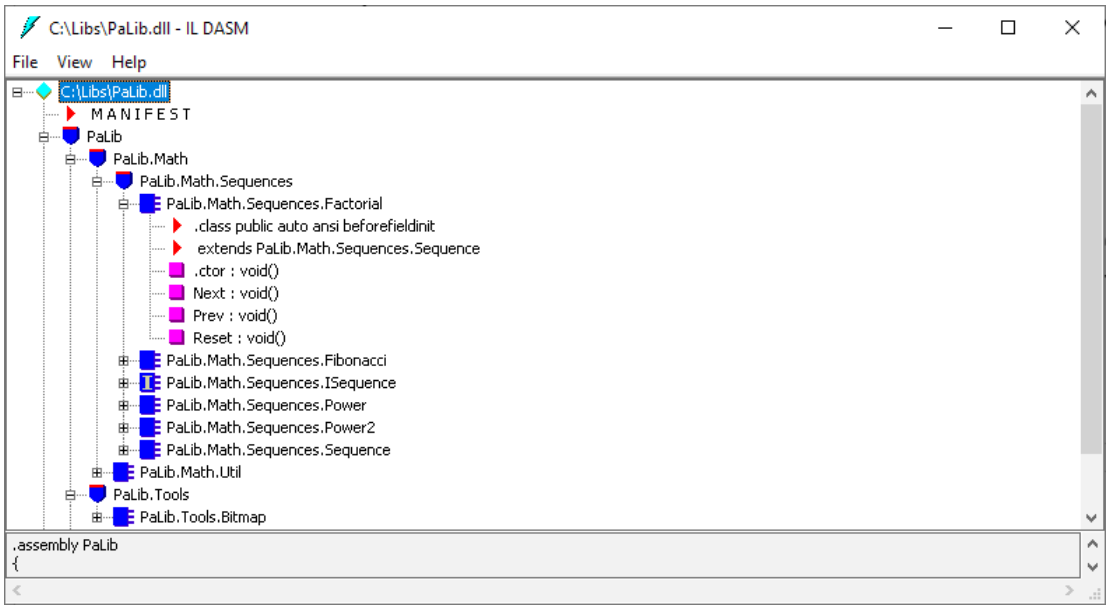
In the following, I will use Ildasm to analyze the dll *PaLib* and examine its contents. With Ildasm you can open and read an assembly's CIL code, metadata and manifest. It should be emphasized that Ildasm can open both a dll and an exe file.

Below is the screen shot of Ildasm with *PaLib* loaded:



The top level is the namespace *PaLib*, and below are two namespaces and two classes. Under the namespace *PaLib.Math* there are a class and a namespace with 6 types, while under the namespace *PaLib.Tools* there is one class. You should notice which symbols (icons) that Ildasm uses for the different types.

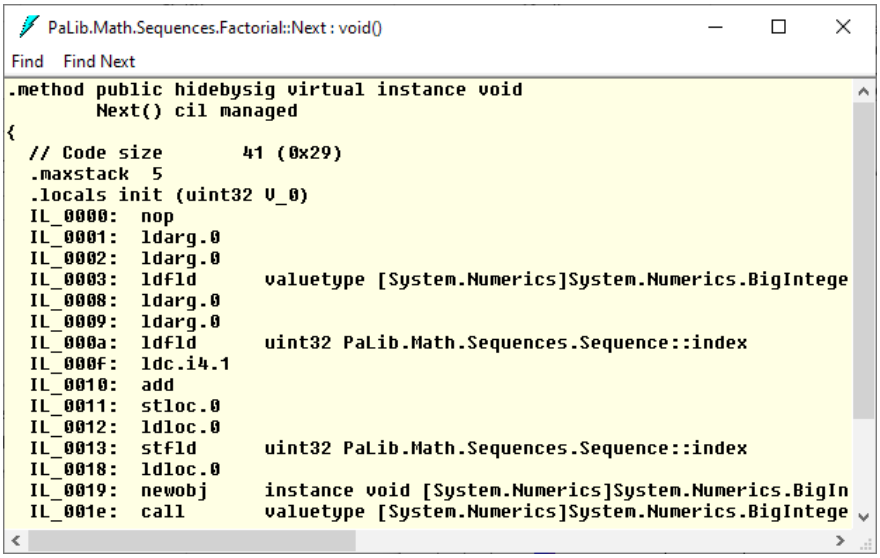
If you open one of the classes for example *Factorial*, you can see that the type is a class that inherits the class *Sequence*, that it has a default constructor, and that it has three other void methods without parameters:



So you can use Ildasm to examine the contents of an assembly

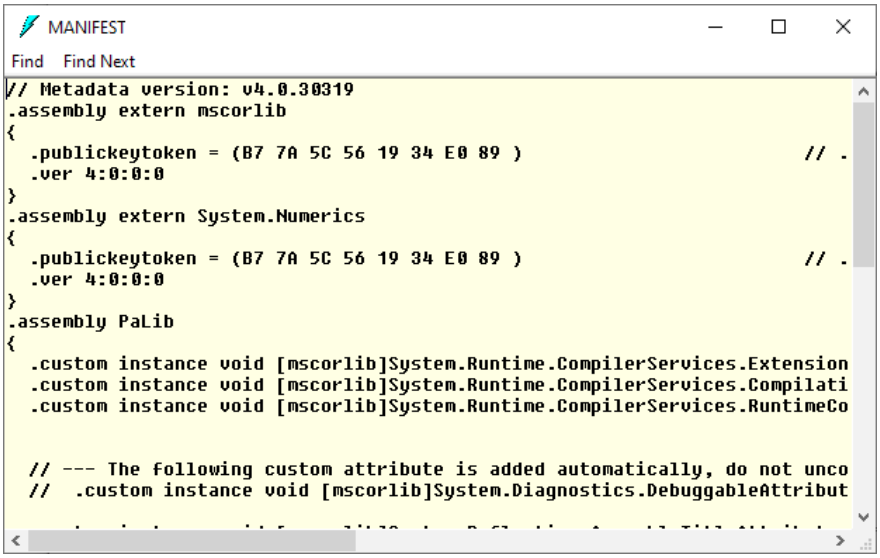
- what types there are
- what namespaces the types is divided into
- what public members a type has

If you double-click a method, for example the method *Next()* in the class *Factorial()*, you get a window that displays the CIL code:



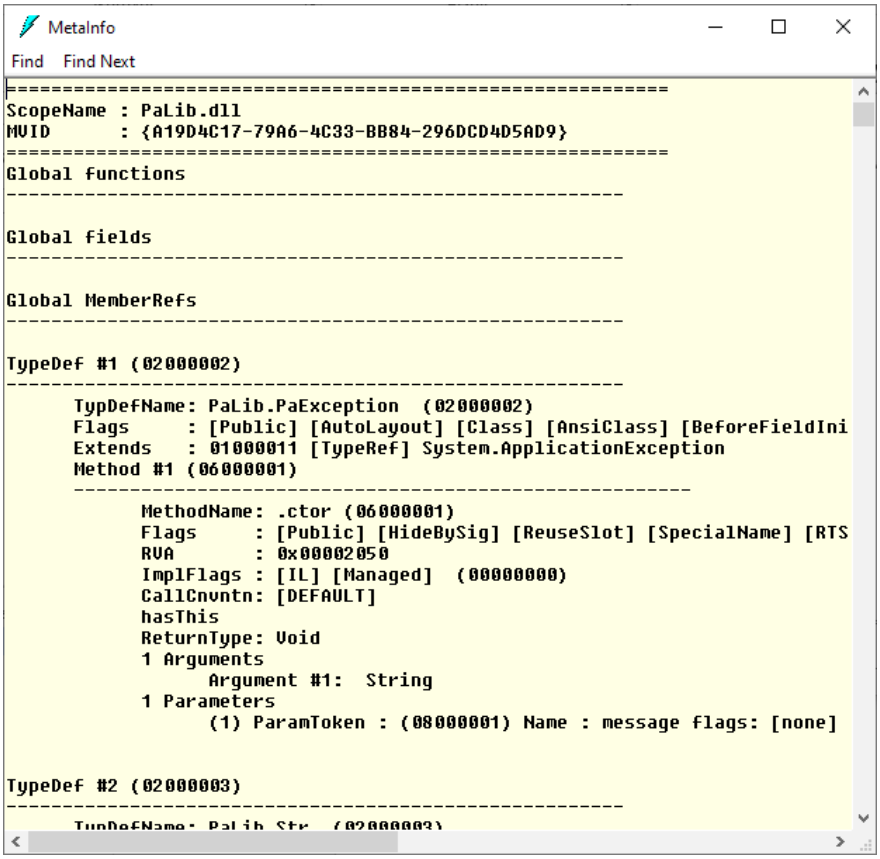
I will not try to interpret the code here, and how readable or unreadable the code is depends on whether you are used to looking at code at this low level. However, some of it can be immediately recognized, for example references to elements in the base class, but you should note how many CIL instructions even a very simple method such as *Next()* are compiled to.

At the top of the Ildasm window is an assembly manifest icon. Clicking on it will open a window with the manifest, and below is a minor part of it:



First, it may be noted that there is an external reference. It is for an assembly called *mscorlib* and is the assembly that contains all the basic classes in the .NET framework. This is an example of a shared dll, which will be explained later. Next comes a number of information regarding this assembly. If you open *Properties* under your project in *Solution Explorer*, there is a file called *AssemblyInfo.cs*. It defines a number of attributes that apply to the assembly, and it is these attributes that one can see in the manifest. Finally, in the manifest, there is some information that identifies the module that this assembly consists of, since it is a single file assembly, there is only one module.

If you press Ctrl-M in Ildasm, you get a window that shows the metadata and where the first part is shown below:



It is a very long list which is a description of all the types in the assembly. It is a good exercise to browse the file and although there are many details, the list is actually readable. It is worth noting that it is a complete description of all types and all members, and that this information is part of the assembly.

3.5 PLATFORM INDEPENDENCE

As an argument for using .NET, one will often hear that it is platform independent, and although one can discuss what it should mean, there is something about it. First, the runtime system ensures platform independence so that a .NET program can run on any machine that has a CLR running, and second and more importantly, a program can consist of parts written in different programming languages when there is a compiler which can translate each part into binary .NET code.

As an example I will write a program that uses four classes where two of the classes are written in C# and the other two in VB. Even if you don't know anything about VB, you can easily read the following code.

I starts with an ordinary *Consol Application project* called *AccountProgram*. Next, in the same solution, I created a *Class Library* project, which I have called *CSAccount*. This project has two classes. The first is an abstract class, which represents an account (a loan) with a principal, an interest rate and a number of periods:

```
namespace CSAccount
{
  public abstract class Account
  {
    protected double principal;
    protected double interestRate;
    protected int period;

    public Account(double principal, double interestRate, int period)
    {
      this.principal = principal;
      this.interestRate = interestRate;
      this.period = period;
    }

    public double Principal
    {
      get { return principal; }
    }

    public double InterestRate
    {
      get { return interestRate; }
    }

    public int Period
    {
      get { return period; }
    }

    abstract public double Payment(int n);
    abstract public double Interest(int n);
    abstract public double Repayment(int n);
    abstract public double Outstanding(int n);
  }
}
```

The class does not require special explanation, but note that there are abstract methods that return respectively the payment, the interest and repayment at the *n*th period, as well as an abstract method that returns the outstanding debt immediately after the *n*th payment is paid.

The second class in the *CSAccount* project is called *Annuity* and is a class that inherits *Account* and thus implements the abstract methods. The class represents as the name says an annuity loan:

```
namespace CSAccount
{
  public class Annuity : Account
  {
    public Annuity(double principal, double interestRate, int period) :
      base(principal, interestRate, period)
    {
    }

    public override double Payment(int n)
    {
      return principal * interestRate / (1 - Math.Pow(1 + interestRate,
      -period));
    }

    public override double Outstanding(int n)
    {
      return principal * Math.Pow(1 + interestRate, n) -
             Payment(0) * (Math.Pow(1 + interestRate, n) - 1) /
interestRate;
    }

    public override double Interest(int n)
    {
      return Outstanding(n - 1) * interestRate;
    }

    public override double Repayment(int n)
    {
      return Payment(n) - Interest(n);
    }
  }
}
```

In this same solution I then create another class library project, which I call *VBAccount*, but this time I choose VB as language. Next, I reference the assembly that contains the classes from *CSAccount*, that is the file *CSAccount.dll*. The first class in the new class library is an account that represents a serial loan:

```
Imports CSAccount

Public Class Serial
  Inherits Account

  Public Sub New(ByVal principal As Double, ByVal interestRate As Double,
    ByVal period As Integer)
    MyBase.New(principal, interestRate, period)
  End Sub

  Public Overrides Function Repayment(ByVal n As Integer) As Double
    Return Principal / Period
  End Function

  Public Overrides Function Interest(ByVal n As Integer) As Double
    Return Outstanding(n - 1) * InterestRate
  End Function

  Public Overrides Function Outstanding(ByVal n As Integer) As Double
    Return Repayment(0) * (Period - n)
  End Function

  Public Overrides Function Payment(ByVal n As Integer) As Double
    Return Repayment(n) + Interest(n)
  End Function
End Class
```

The class is simple, but you should note that the language is VB and that the class inherits the class *Account*. The last thing is possible, though this class is written in the C#, the compiler does not need to know what language *Account* is written in - CIL code is CIL code whether or not the one programming language or the other is used.

The *VBAccount* project has another class that implements an amortization plan, and this class is also written in VB:

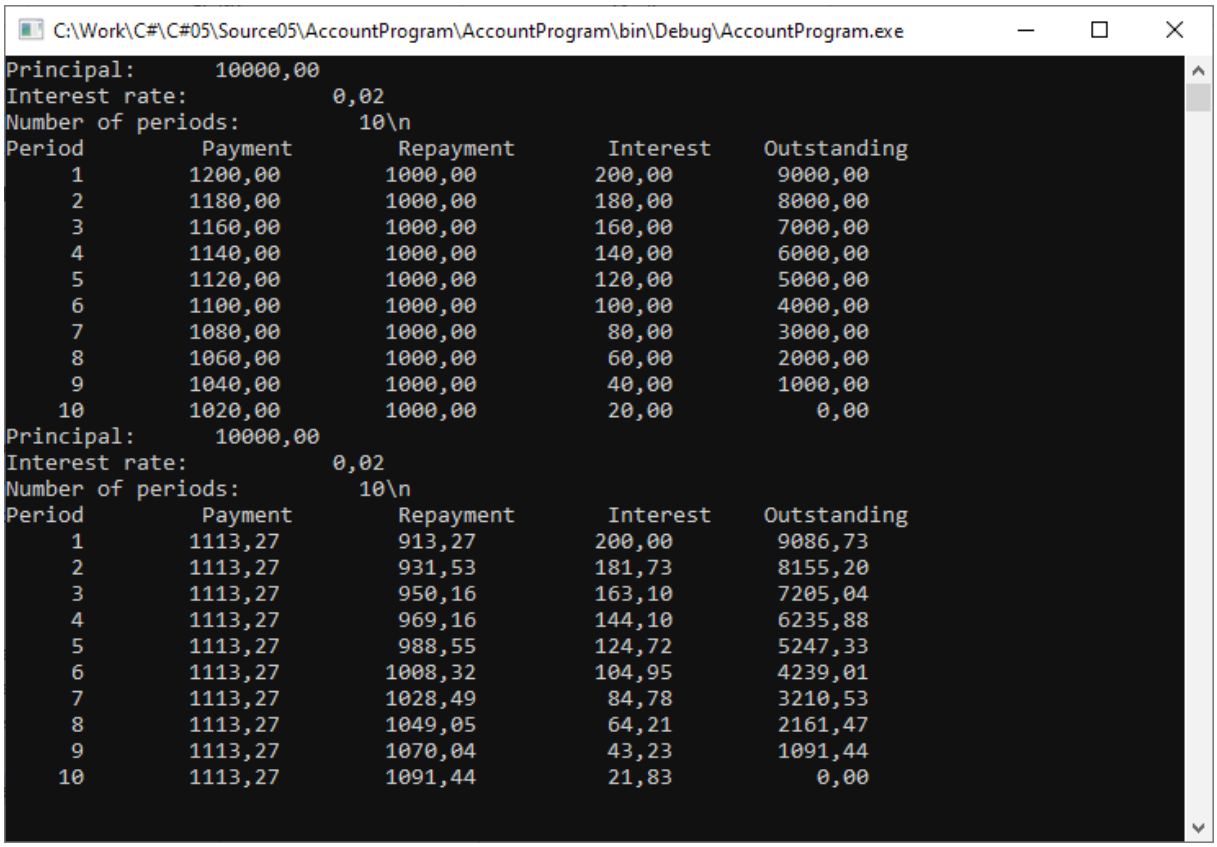
```
Imports CSAccount

Public Class Amortization
  Private loan As Account

  Public Sub New(ByVal loan As Account)
    Me.loan = loan
  End Sub

  Public Sub Print()
    Console.WriteLine("Principal: {0, 10:F}", loan.Principal)
    Console.WriteLine("Interest rate: {0, 10:F}", loan.InterestRate)
    Console.WriteLine("Number of periods: {0, 10:D}\n", loan.Period)
    Console.WriteLine(
      "Period Payment Repayment Interest Outstanding")
    For n As Integer = 1 To loan.Period
      Console.WriteLine("{0, 6:D}{1, 15:F}{2, 15:F}{3, 15:F}{4, 15:F}", n,
        loan.Payment(n), loan.Repayment(n), loan.Interest(n), loan.
        Outstanding(n))
    Next
  End Sub
End Class
```

Back there is the main program, which will use the 4 classes to print an amortization plan for respectively a serial loan and an annuity loan. The main program is a C# program, and it must therefore have a reference to both assemblies with the two class libraries:

- *CSAccount.dll*
- *VBAccount.dll*

The the program can be written as

```
using CSAccount;
using VBAccount;

namespace AccountProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      Amortization plan1 = new Amortization(new Serial(10000, 0.02, 10));
      Amortization plan2 = new Amortization(new Annuity(10000, 0.02, 10));
      plan1.Print();
      plan2.Print();
    }
  }
}
```



3.6 PRIVATE ASSEMBLY

A private assembly is only used by the application to which it is associated. For example is the assembly *CSAccount* above a private assembly for the program *AccountProgram*. A private assembly must be in the same directory as the application or in a sub-directory. An important consequence of this is that a private assembly should not be registered in the registry. That means that one can easily copy an application and all its private assemblies to another folder, and the application can still run. Similarly, it is trivial to delete such an application: You simply delete the exe file and all the assemblies.

When the runtime system needs to find a specific private assembly, a simple technique called probing is used: When the manifest has a reference to an external assembly named *CSAccount*, it first searches the application folder for a file named *CSAccount.dll*. If it is not found is searched for a *CSAccount.exe* file, and if it does not exist, you get an exception. However, you can specify in the config file that the runtime system should search other directories.

In the manifest, an assembly is identified by a name and a version number. For example will the manifest for the program *AccountProgram* contains

```
.assembly extern CSAccount
{
  .ver 1:0:0:0
}
```

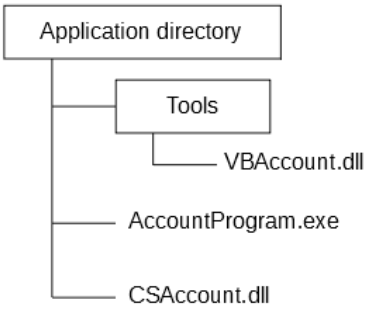
In the case of a private assembly, however, only the name matters, as no version control is necessary here. If there are multiple applications using a given assembly, a copy of the assembly will be found for each application.

An application may have associated a file with extension *config* that has the same name as the application, for example *AccountProgram.config*. It is an XML file, which among other things can be used to specify in which subdirectories the runtime system should look for (probe) external assemblies.

If you look at the program *AccountProgram* it consists of three assemblies

- *AccountProgram.exe* which is the applicationen itself
- *CSAccount.dll* which contains two typers
- *VBAccount.dll* which contains two types (written in VB)

These are all three in the same directory. I will now move one (for example *VBAccount.dll*) to a subdirectory that I will call *Tools*. If you create the subdirectory *Tools* and move the file *VBAccount.dll*, you will find that the program cannot run, but crashes with an exception. A config file is required. It is an XML document which must be in the application directory and have the same name as the program followed by *config* - that is in this case *AccountProgram.exe.config*. Here you can specify which subdirectories that the runtime system should probe if an external assembly is not found in the application directory.



Visual Studio has probably already added a config file named *App.config* to the project. If not, you can add a new Application configuration file yourself. Note that the file name is *App.config* - it must not be changed. If you open the file, you have the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
</configuration>
```

You need to add some XML that tells where the system must searches the dll in question:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="Tools"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

That, in turn, is all it takes. However, there is a slight oddity. Even though there is a config file with a probing tag, the compiler will still copy that assembly to the application directory. To avoid this, in Visual Studio, right-click on the current assembly, select properties and set *Copy Local* to *false*.

As a final note: If you want to probe multiple sub-directories, you can specify multiple names separated by a semicolon, e.g.

```
<probing privatePath="Tools;Util"/>
```

3.7 SHARED ASSEMBLIES

A shared assembly is an assembly that can be used by multiple applications on the same machine. For example all the assemblies containing the .NET types are shared assemblies. A shared assembly is not copied to the application directory, but is at a central location called GAC - *Global Assembly Cache*. In order for a shared assembly to be installed in the GAC, it must have a strong name so that it can be uniquely identified. This means, among other things, that the GAC can contain the same assembly in several versions.

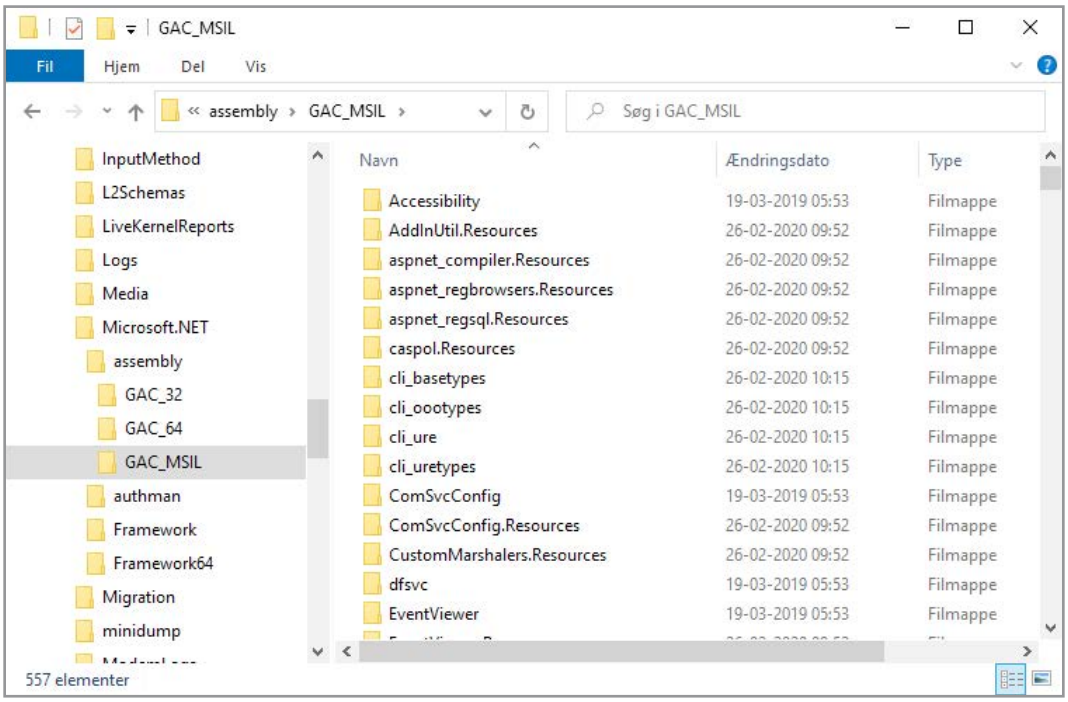
Prior to .NET 4.0, the GAC is a subdirectory called *Assembly* and located under the *Windows* directory:

```
C:\Windows\Assembly
```

but from version 4.5 the GAC is moved to:

```
C:\Windows\Microsoft.NET\assembly\GAC_MSIL
```

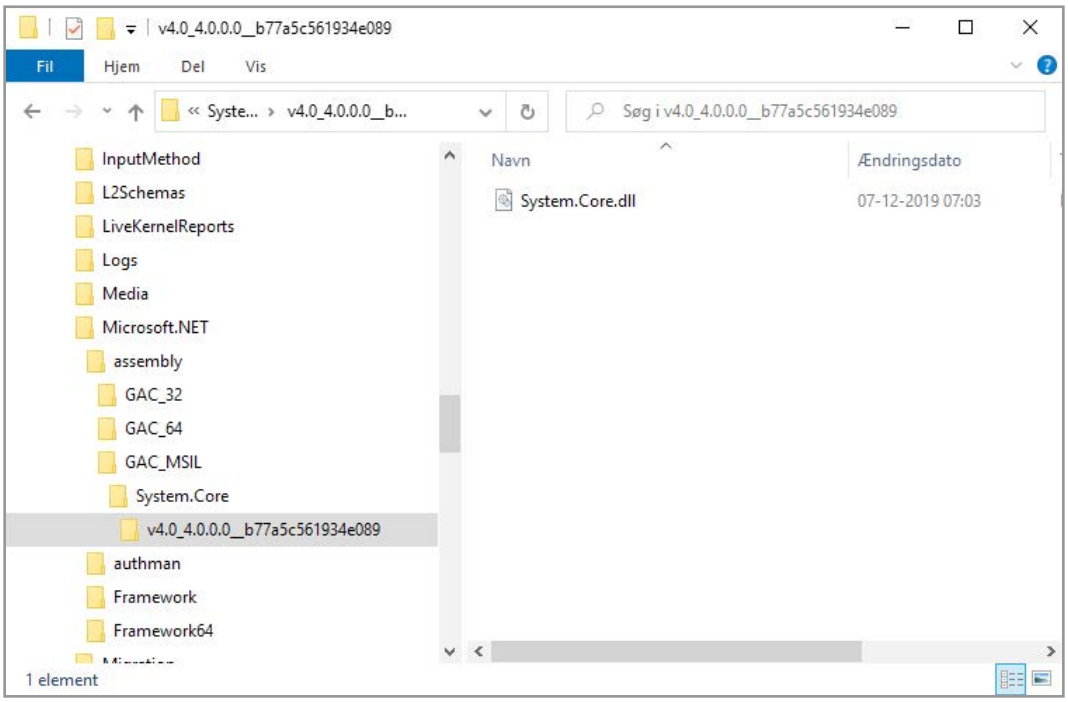
If you open this directory in the Explorer you will get something like the following:



showing shared assembly on my machine. If you find the library *System.Core* and open it, there is a subdirectory named

```
v4.0_4.0.0.0_b77a5c561934e089
```

and below that, there is a shared dll:



The name *v4.0_4.0.0.0_b77a5c561934e089* looks a bit strange, but the first means that the assembly is translated to .NET version 4.0 (or 4.5), followed by the version number and finally the assembly's public key.

3.8 A STRONG NAME

You cannot install an exe file in the GAC, but only an assembly of the type dll. To install a dll in the GAC, the assembly must be identified by a strong name that consist of

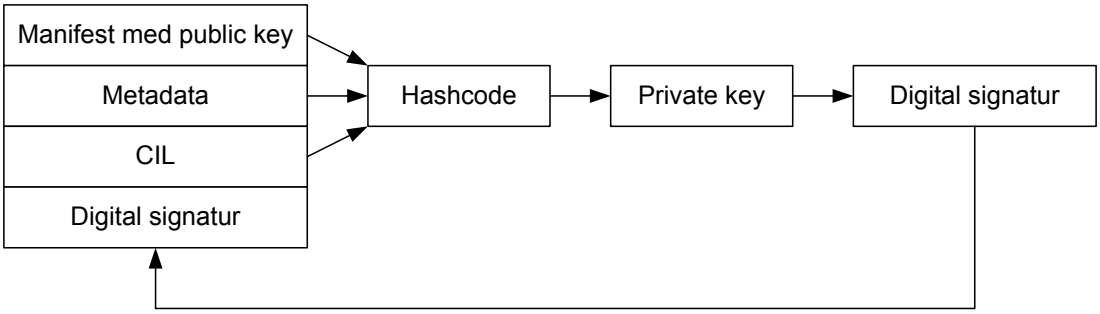
- the name of the assembly (without extension)
- a version number
- a public key
- a culture identification (it is not a requirement)

The goal of this strong name is two:

- to ensure that the assembly has a unique name which only this one assembly has
- to increase security so that a user can verify that the assembly has not changed after leaving the development department

To create a strong name, the first step is to create a key file with a private / public key. I'll show you how in a little while. A private / public key is a pair of keys that can be used to encrypt data. The idea is that you can encrypt data with both keys - either with the private key or the public key, but if you have encrypted data with one of the keys, you cannot decrypt data unless you have the other key.

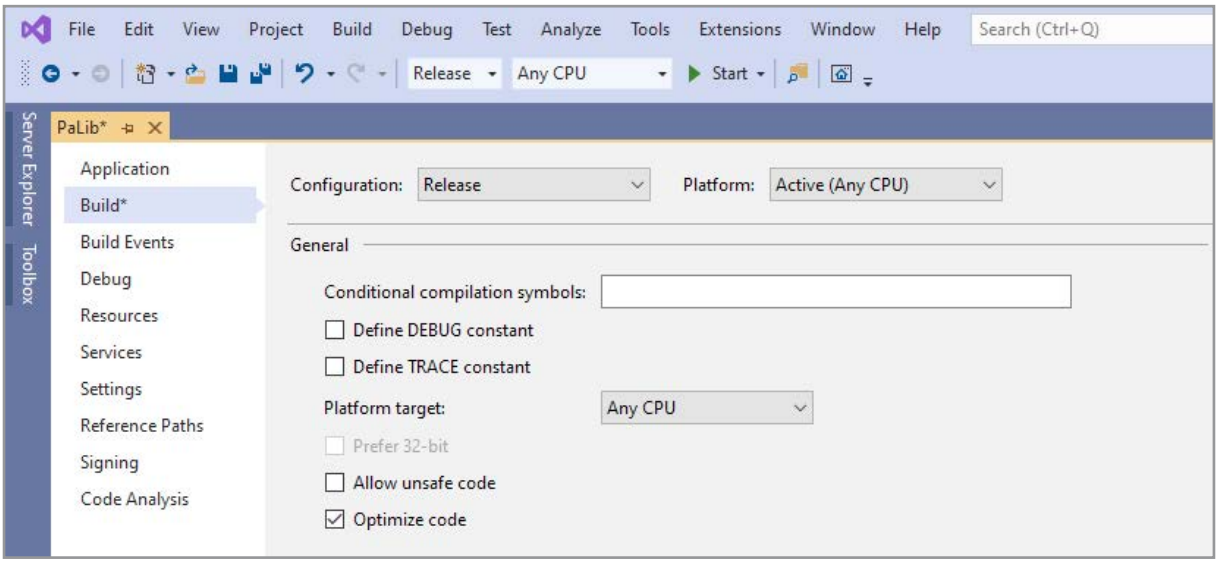
As soon as you have created the key file and linked it to the project, the compiler will insert the public key into the assembly manifest. Next, the compiler will generate a hash code based on the entire assembly (CIL code, metadata and manifest). This hash code is combined (encrypted) with the private key (which only the development department has access to) for a digital signature that is inserted into the assembly's CLR header.



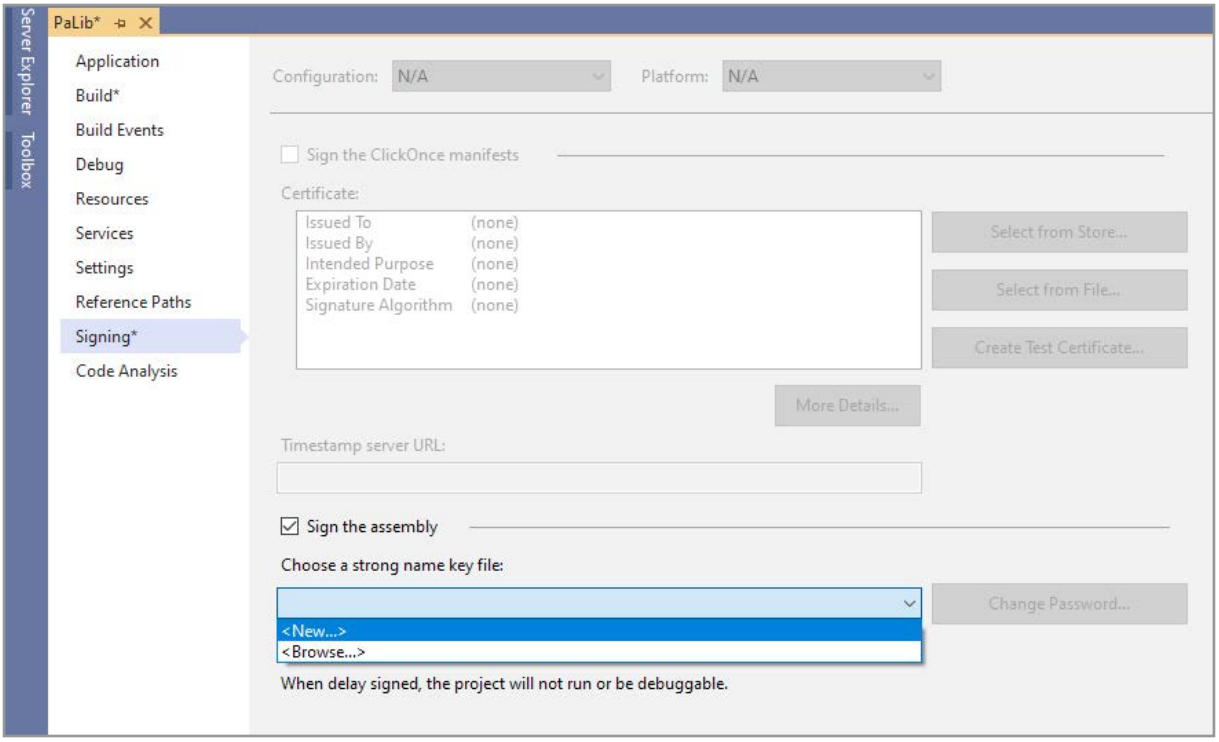
Here it is important to note that the private key is not part of the assembly and is used solely by the compiler to create the digital signature. This ensures that it is only the development department that can have created the digital signature. This has achieved that a unique identifier is attached to an assembly in the form of the digital signature. Since the public key is part of the manifest, everyone can determine the hash code, but you can only decrypt the hash code if the digital signature is created with the corresponding private key. Everyone can calculate a new hash code, but if the assembly has changed since leaving the development department, the two hash codes will not vote and you can see that the assembly has changed. The digital signature thus also provides security against the fact that someone has changed the assembly.

To build a shared assembly, I will use the class library *PaLib* that I created above.

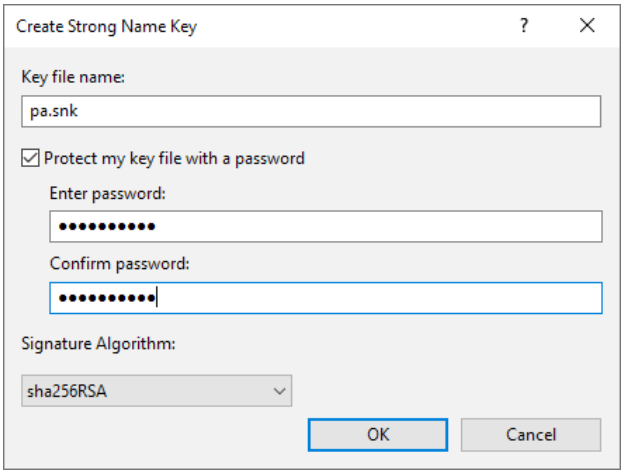
Since a shared assembly is a finished class library, I don't want it to contain debug information. Therefore, in the project properties and under *Build* and *Configuration*, I would like to specify that a *Release* assembly should be formed:



I then reopen the project properties, but this time I select the *Signing* tab:

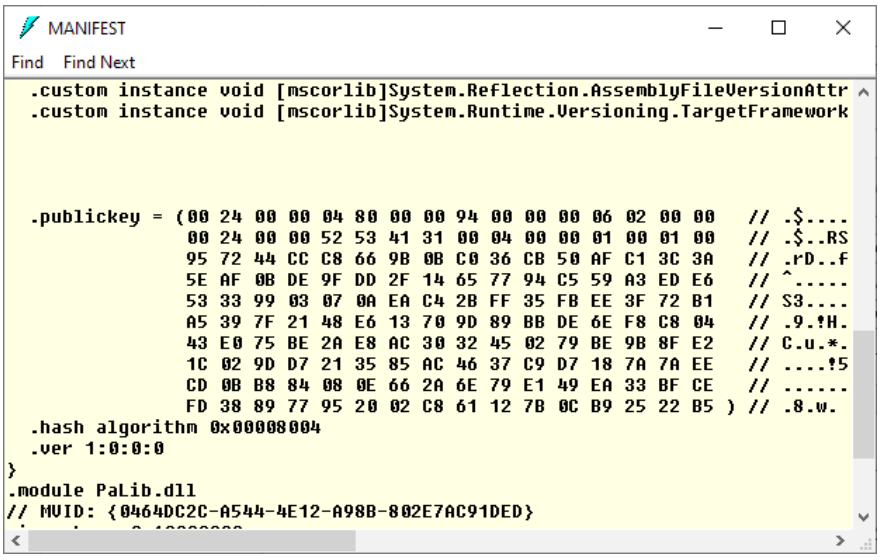


Next, I need to make sure that the *Sign the assembly* check box is selected, and in the combo box I need to select <New ...>:

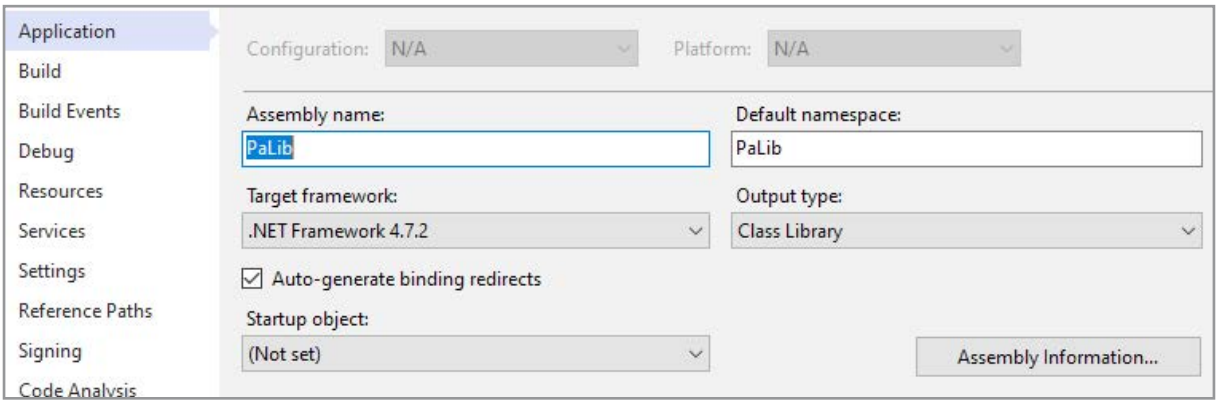


Here I enter the name of the key file (*pa.snk*) and a password. The algorithm tells which private / public key algorithm to use, but here you just have to keep the default. After clicking OK, Visual Studio has created a key file, and if you then build the project, the result is an assembly with a strong name.

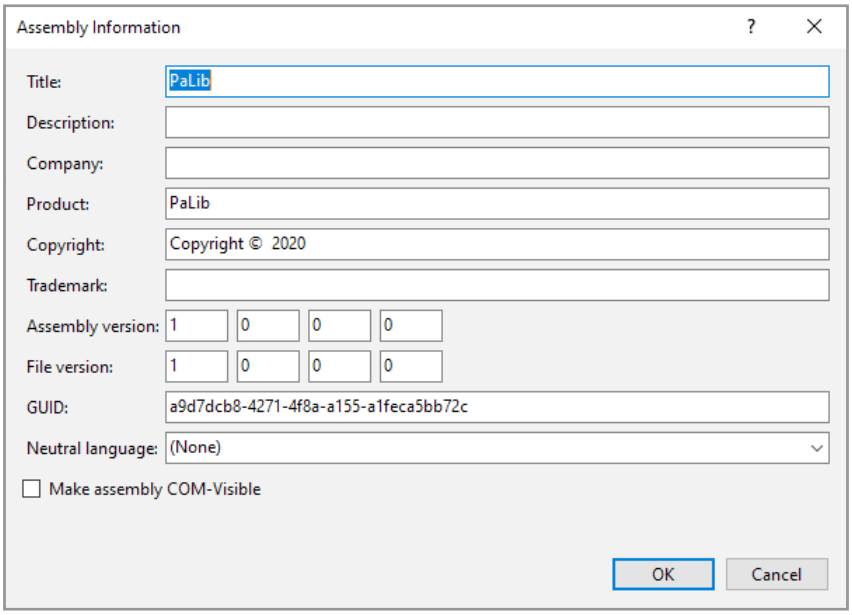
If you now open the assembly manifest in ILDASM (remember it must be the dll under *Release*), you can see that it is now an assembly with a strong name, since a public key is generated:



Also note that if you opens the project properties and the *Application* tab, there is a button called *Assembly Information*:



If you click on it, you will get a window where you can maintain the version number:



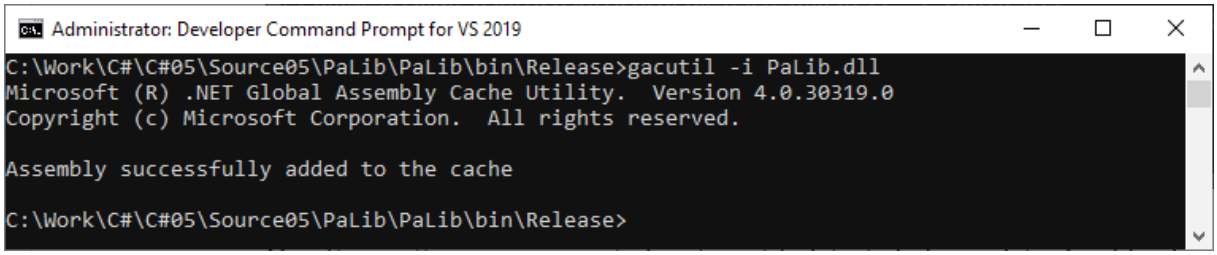
The version number can also be added to the project's *AssemblyInfo* file:

```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the information
// associated with an assembly.
[assembly: AssemblyTitle("PaLib")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("")]
[assembly: AssemblyProduct("PaLib")]
[assembly: AssemblyCopyright("Copyright © 2020")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]

// Setting ComVisible to false makes the types in this assembly not visible
// to COM components. If you need to access a type in this assembly from
// COM, set the ComVisible attribute to true on that type.
[assembly: ComVisible(false)]

// The following GUID is for the ID of the typelib if this project is
   exposed to COM
[assembly: Guid("a9d7dcb8-4271-4f8a-a155-a1feca5bb72c")]

// Version information for an assembly consists of the following four
   values:
//
//     Major Version
//     Minor Version
//     Build Number
//     Revision
//
// You can specify all the values or you can default the Build and
   Revision Numbers
// by using the '*' as shown below:
// [assembly: AssemblyVersion("1.0.*")]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

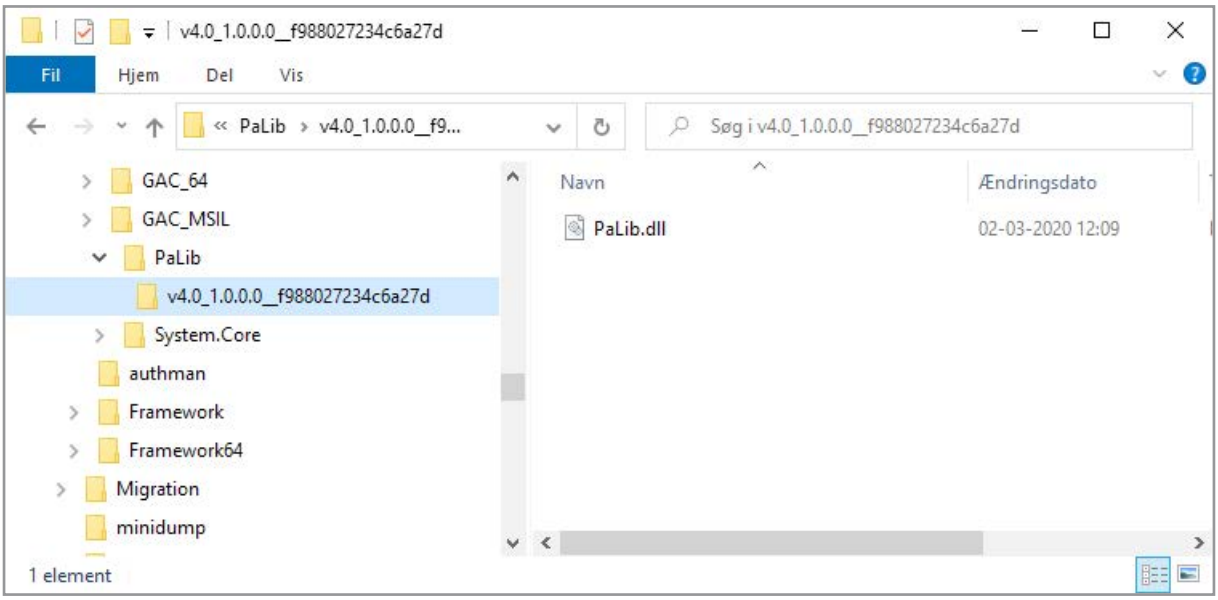
3.9 INSTALL A SHARED ASSEMBLY

Now I have a shared assembly, so as the next point it must be installed in the GAC. This is typically done with an MSI install package, but on the local machine you can do it with a command called *gacutil.exe*. The easiest way is to open a *Developer Command Prompt* (you need to run the program as administrator). Next, you set the current directory to the location of the dll in question and execute the command

```
gacutil -i PaLib.dll
```



Then the dll is installed in the GAC, and if you now open the GAC in Explorer, you can find the assembly *PaLib*:



The command *gacutil* can also be used to uninstall a shared assembly and the command has other options, for example to determine where an assembly is installed in the GAC.

EXERCISE 5: ERATOSTHENES A LAST TIME

Create a copy of the project *Eratosthenes1* and call the copy *Erastosthenes2*. Remove the file *PaLib.dll* from the project. Open the project in Visual Studio and:

1. Remove the reference to the class library *PaLib* (it is the reference to the private assembly)
2. Set a reference to the shared assembly *PaLib* in the GAC
3. Set the property *Copy Local* for *PaLib* to *false*
4. Build the project

The program should run again. When you are sure that it all works use Explorer to test that the project do not have any copy of *PaLib*.

3.10 APP.CONFIG

In connection with private assemblies, I have discussed how to specify in an *App.config* file where an assembly is located. However, you can also use configuration files in connection with shared assemblies, and the following are a few remarks about what it is about, but basically it is a matter that an administrator can specify the properties of a shared assembly via an XML configuration file. If you need something, you need to look at the documentation for the configuration file.

A typical use of a configuration file is that you want the runtime system to bind to a different version of an assembly than what is stated in the program's manifest. If a program such as *Erastosthenes* uses an assembly such as *PaLib*, and it changes to a new version (with a new version number) and is installed in the GAC, the program will not see it, since the manifest indicates something else. This can be specified in a config file where you specify the strong name of the assembly and which alternative version to use.

It is also possible to specify a so-called publisher policy, which is actually a compiled config file that is installed in the GAC. With such a file it is possible to specify that all programs using a particular assembly must use the latest version. The policy file itself must have a strong name (it must be installed in the GAC) and the file is created with a special tool called *al.exe* (assembly links). The principle is that when CLR loads a particular assembly, it will check if there is a policy file and, if applicable, select the version that the file specifies.

Finally, I want to mention codebase as a tag that can appear in the config file. Hereby one can indicate that an assembly exists elsewhere and possibly on a remote machine. In the latter case, the assembly is downloaded to the local machine.

PROBLEM 2: A CALENDAR

In this problem you must create a class library, install the class library in the GAC and then create the program that uses the class library. The task is relatively comprehensive and you are encouraged to follow the guidelines below.

1) Create a new class library which you can call *PaMath*. In exercise 11 in the previous book you should write a struct *Time* which represents a time for a day. Add this class to the new project. Remember to change the namespace definition.

Add also an exception class, which you can call *DateException*.

2) You should then create type that represents a date from year 0 to year 9999. The type must take account of the shift from the Julian to the Gregorian calendar. The switch to the Gregorian calendar has not happened the same year in all countries, but in this type you should use the year 1582, where October 4 is followed by October 15. The means there is 10 days that does not exists.

The type should know the main holidays. You decide which ones, but at least it should know Easter Day. It is not quite easy to calculate when it is Easter, but you probably can find examples on the web or somewhere else that shows how to do it.

As a sketch below is a proposal for how the class can be written and what methods it should have. You should note that it is an immutable class and that the state of an object cannot be changed.

```
namespace PaDate
{
  public class Date : IComparable<Date>
  {
    private readonly int number; // represents a date as an int YYYYMMDD

    // Default constructor that initializes a date to the current date.
    public Date() { ... }
    public Date(int year, int month, int day) // the parameters must
    represents a legal date

    public override bool Equals(object obj) { ... }
    public override int GetHashCode(){ ... }
    public int CompareTo(Date date) { ... }
    public override string ToString() { ... }

    public int Year { get { ... } }
    public int Month { get { ... } }
    public int Day { get { ... } }
    public int DaysInMonth { get { ... } } // number of days in the
    date's month
    public int DaysInYear { get { ... } } // number of days in the
    date's year
    public string Dayname { get { ... } } // name of week day
    public string Monthname { get { ... } }
    public int WeekDay { get { ... } } // 1 = monday, 2 = tuesday,
...
    public Date Next { get { ... } } // the date after this date
    public Date Prev { get { ... } } // the day before this date
    public Date NextMonth { get { ... } } // the date a month later
    public Date PrevMonth { get { ... } } // the date a month before
    public Date NextYear { get { ... } } // the date a year later
    public Date PrevYear { get { ... } } // a date year before
    public int Weeknumber { get { ... } } // week number for this date
    public bool IsHoliday { get { ... } } // true if this date is a
holiday
    public string Holiday { get { ... } } // name of a holiday or blank
    public bool Leapyear { get { ... } } // true is this date is for
    a leap year

    public static bool operator ==(Date date1, Date date2) { ... }
```

```
    public static bool operator <=(Date date1, Date date2) { ... }
    public static bool operator >(Date date1, Date date2) { ... }
    public static bool operator >=(Date date1, Date date2) { ... }
    public static Date operator ++(Date date) { ... } // next date
    public static Date operator --(Date date) { ... } // previous date
    // Number of days between to dates, negative if date1 is after date2
    public static int operator -(Date date1, Date date2) { ... }
    // This date n days later, if n < 0 the operation is ignored.
    public static Date operator +(Date date, int n) { ... }
    // This date n days before, if n < 0 the operation is ignored.
    public static Date operator -(Date date, int n) { ... }

    public static implicit operator int(Date date) { ... }
    public static explicit operator Date(int n) { ... }
    public static implicit operator string(Date date) { ... }
    public static explicit operator Date(string text) { ... }
    public static implicit operator DateTime(Date date) { ... }
    public static implicit operator Date(DateTime date) { ... }

    public static string MonthName(int n) { ... } // month name
    public static bool IsLeapyear(int year) { ... }
    public static int GetDaysInMonth(int year, int month) { ... }
    public static int GetDaysInYear(int year) { ... }
    public static int GetWeekday(Date d) { ... } // 1 = Monday, 2 =
    Tuesday, ...
    public static int GetWeeknumber(Date d) { ... } // week number for
    the date d
    public static bool IsDate(int date) { ... } // true if date =
    YYYMMDD is legal
    public static bool IsDate(int year, int month, int days) { ... }
  }
}
```

When you have written the class you must compile the class library and it should be ready for use. Note that for the moment it is a private assembly.

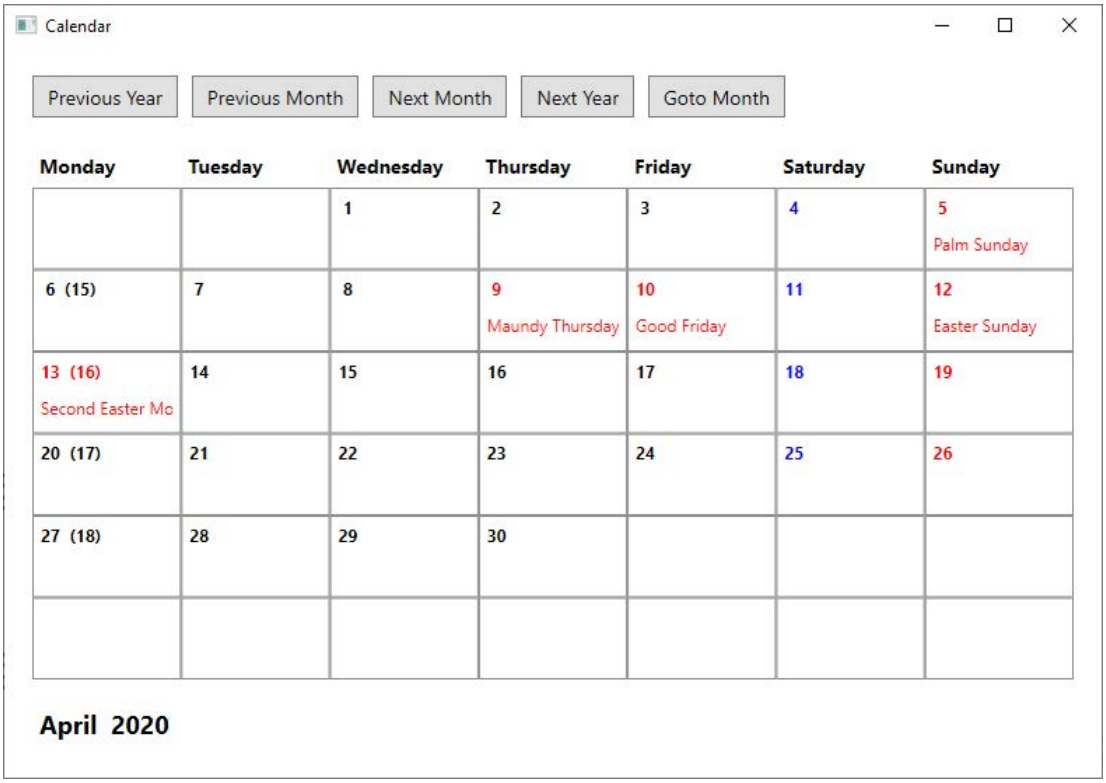
3) You must then write a program that uses the class *Date* and then your class library. Create a new WPF project which you can call *Calendar*. When you opens the program it must show a window like shown below, that is a window which shows a calendar for a month. Below is shown a month with holidays, but for a particular day, only the day's number of the month as well as if it is a Monday, the week number is displayed. In addition, the name appears if the day is a public holiday. The user interface is proven simple (a *UniformGrid* in a *Grid*) as it is not the primary goal of the task.

The window has 5 buttons:

1. *Previous Year* which navigate the calendar a year back
2. *Previous Month* which navigate the calendar a month back
3. *Next Month* which navigate the calendar a month ahead
4. *Next Year* which navigate the calendar a year ahead
5. *Goto Month* where the user must enter a year and a month for the month to be shown in the calendar

The last feature requires, that you to add a dialog to the program so the user can enter the values for a month.

Note that the program requires a reference to the assembly with your class library.



When you have written the program you must test it to ensure that the calendar works at it should.

4) You must now change your class library to a shared assembly:

1. Open the project with your class library.
2. Change the project setting such the library is compiled to a *Release* version.
3. Sign the project with a strong name.
4. Build the project.
5. Use Gacutil to install your class library in the GAC. Remember to install the Release version, the version with the strong name.

Then you must test your Calendar program with your shared assembly:

1. Open the project in Visual Studio.
2. Remove the reference to the class library (it is the private assembly).
3. Add a reference to the class library in the GAC.
4. Remember to set the project's *Copy Local* (for the class library) to *false*.
5. Test the program, it should all work again.
6. Use Explorer to test that the program folder not have a copy of the assembly.

4 REFLECTION

As shown above, an assembly contains metadata for the assembly's types. It is a complete description of the types and including their methods, properties, variables and son on. You can see an assembly's metadata by opening it in Ildasm, and the description can be quite comprehensive. There are many details and I have no way to explain the content here, but in fact the content is readable and one can interpret most of it without any specific explanation.

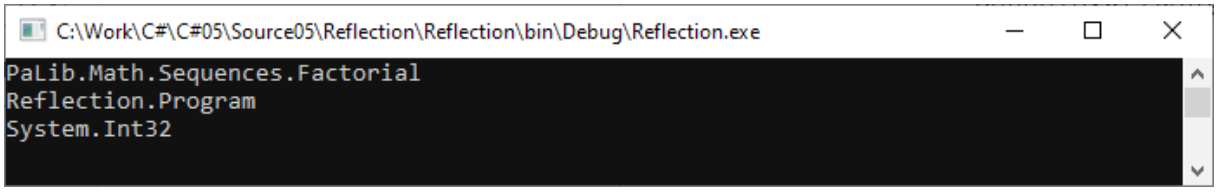
Reflection deals with how to determine a type's properties at runtime. It is possible, at runtime, to load an assembly and determine information about the assembly types, which of course happens by loading its metadata. It requires some types that are defined in the namespace *System.Reflection* as well as the type *System.Type*. As the name says, it is a class that represents a type and it is a very comprehensive class with many methods. It is an abstract class, so you cannot create an object with *new*, but the class *Object* has a method *GetType()*, which returns a *Type* object for the type of the specific object. The class *Type* also has a static method that returns a *Type* object from a string containing the type name, and finally, you can use the operator *typeof()*, that returns a *Type* object on the basis of a type name.

As an example I will use the assembly *PaLib.dll* to show how to use reflection. I have created a new console application project and added a reference to the private version of the class library. The method *Test01()* determines a *Type* object for three types

- the type *Factorial* in the class library *PaLib*
- *the* type of the program, that is the class *Program*
- the type *int*

```
using System;

using PaLib.Math.Sequences;

namespace Reflection
{
  class Program
  {
    static void Main(string[] args)
    {
      Test01();
      Console.ReadLine();
    }

    static void Test01()
    {
      Factorial fac = new Factorial();
      Type t1 = fac.GetType();
      Type t2 = Type.GetType("Reflection.Program");
      Type t3 = typeof(int);
      Console.WriteLine(t1);
      Console.WriteLine(t2);
      Console.WriteLine(t3);
    }
  }
}
```

If you run the program the result is:



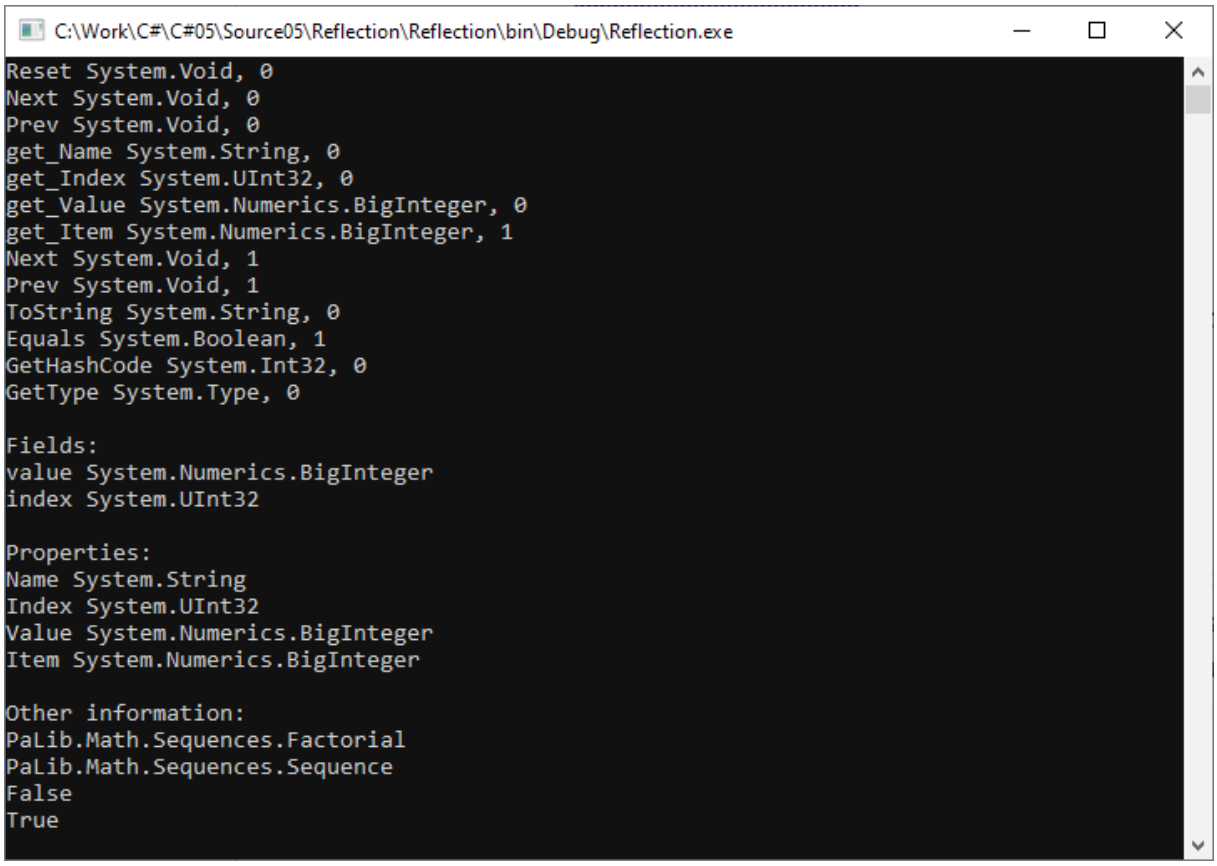
You should note that the types are determined in three different ways.

As mentioned, using types in the namespace *System.Reflection* you can read an assembly's metadata. The following method *Test02()* shows how:

```
using System;
using System.Reflection;
using PaLib.Math.Sequences;

namespace Reflection
{
  class Program
  {
    ...

    static void Test02()
    {
      Factorial fac = new Factorial();
      Type type = fac.GetType();
      ShowMethods(type);
      ShowFields(type);
      ShowProperties(type);
      ShowOther(type);
    }

    static void ShowMethods(Type type)
    {
      Console.WriteLine("Methods:");
      MethodInfo[] methods = type.GetMethods();
      foreach (MethodInfo info in methods)
        Console.WriteLine("{0} {1}, {2}", info.Name, info.ReturnType,
                                          info.GetParameters().Length);
    }

    static void ShowFields(Type type)
    {
      Console.WriteLine("\nFields:");
      FieldInfo[] fields =
        type.GetFields(BindingFlags.Instance | BindingFlags.NonPublic);
      foreach (FieldInfo info in fields)
        Console.WriteLine("{0} {1}", info.Name, info.FieldType);
    }

    static void ShowProperties(Type type)
    {
      Console.WriteLine("\nProperties:");
      PropertyInfo[] props = type.GetProperties();
```

```
      foreach (PropertyInfo info in props)
        Console.WriteLine("{0} {1}", info.Name, info.PropertyType);
    }

    static void ShowOther(Type type)
    {
      Console.WriteLine("\nOther information:");
      Console.WriteLine(type.FullName);
      Console.WriteLine(type.BaseType);
      Console.WriteLine(type.IsAbstract);
      Console.WriteLine(type.IsClass);
    }
  }
}
```

The method determines the methods, properties and variables for the class *Factorial* in the library *PaLib*. If the method is executed, you get the result:



The code is self-explanatory, and the important thing is that you via a *Type* object at runtime you can retrieve all information about a given type. However, especially note the method *ShowFields()*, which prints information about the class's member variables. By default, private member information is not returned, but it can be specified as a parameter to for example *Getfields()*.

The namespace *System.Reflection* has several important types, and the above program uses several types from this namespace. The most important are probably

Assembly	which represents an assembly and has methods for loading an assembly at runtime
AssemblyName	which represents the name of an assembly and including its strong name
EventInfo	which represents properties of an event
FieldInfo	which represents properties of a member variable
MemberInfo	there is an abstract base class for the other info types
MethodInfo	which represents properties of a method
ModuleInfo	which represents properties of a module
Module	which represents a module in a multi-file assembly
ParameterInfo	which represents properties of a parameter to a method
PropertyInfo	which represents properties of a property

4.1 A SIMPLE DLL

In order to have a simple dll with few types I have created a class library, called *MathLib*, and which has two types only:

```
using System;

namespace MathLib
{
  public struct Point
  {
    public double X { get; set; }
    public double Y { get; set; }

    public double Distance(Point p)
    {
      return Math.Sqrt(Sqr(X - p.X) + Sqr(Y - p.Y));
    }
```

```
    private double Sqr(double x)
    {
      return x * x;
    }
  }
}

namespace MathLib
{
  public static class Functions
  {
    public static ulong Factorial(uint n)
    {
      if (n < 2) return 1;
      return n * Factorial(n - 1);
    }

    public static ulong Fibonacci(uint n)
    {
      if (n < 2) return n;
      ulong f1 = 0;
      ulong f2 = 1;
      while (n-- > 1)
      {
        ulong f3 = f1 + f2;
        f1 = f2;
        f2 = f3;
      }
      return f2;
    }
  }
}
```

I have then in the test program (the program *Reflection*) added a reference to the assembly with the new class library. Note that when it is a private assembly Visual Studio will make a copy of the assembly in the application directory. As the next I have added a test method *Test03()* and three other new methods:

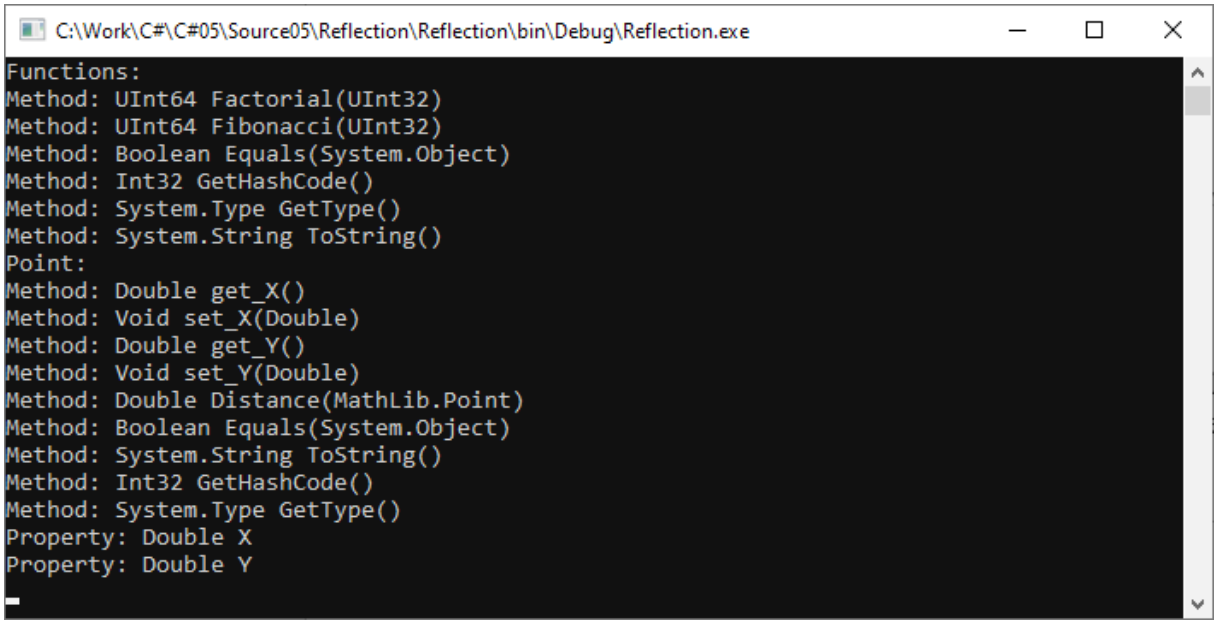
```
namespace Reflection
{
  class Program
  {
    ...

    static void Test03()
    {
      Assembly asm = LoadAssembly("MathLib");
      if (asm != null) ShowTypes(asm);
    }

    private static void ShowTypes(Assembly asm)
    {
      Type[] types = asm.GetTypes();
      foreach (Type type in types) ShowMembers(type);
    }

    private static void ShowMembers(Type type)
    {
      Console.WriteLine(type.Name + ":");
      MemberInfo[] members = type.GetMembers();
      foreach (MemberInfo info in members)
        Console.WriteLine("{0}: {1}", info.MemberType.ToString(), info);
    }

    private static Assembly LoadAssembly(string name)
    {
      try
      {
        return Assembly.Load(name);
      }
      catch
      {
        Console.WriteLine("Assembly ikke fundet...");
        return null;
      }
    }
  }
}
```

The method *LoadAssembly()*loads a private assembly named *name*. Here, the usual probing mechanism is used. *Load()* is a static method in the class *Assembly*, and for a private assembly, the name simply needs to be a string containing the assembly name. However, the string can also consist of a comma-separated list of key / value pairs, where you can specify version

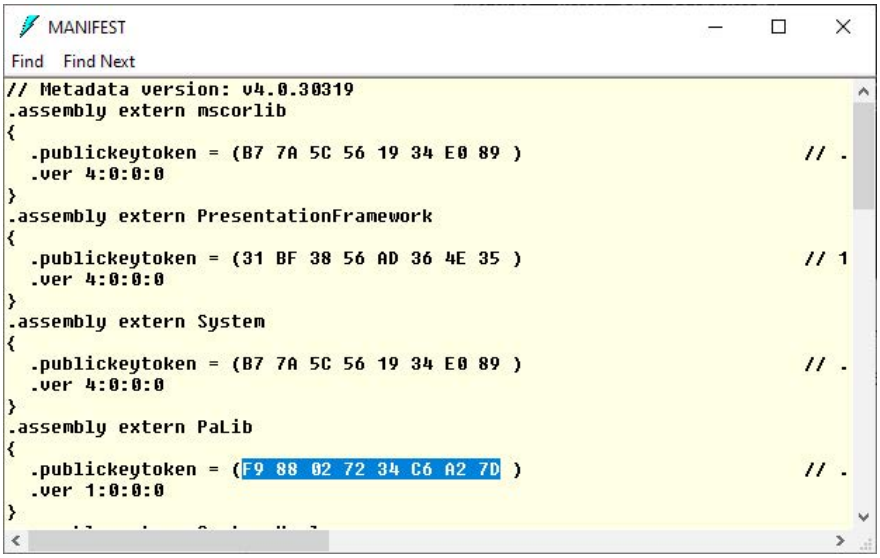
number, etc. The method has several overrides, and there is one that has an *AssemblyName* parameter. It is a type that represents an assembly with its strong name.

The method *ShowTypes()* runs over all types in an assembly, and for each type, the method calls *ShowMembers()*, which prints the name of the type and loops over the types of public members. For each member, the method prints what it is for a member as well as its type.

The method *Test03()* thus shows how to load an assembly and examine its metadata at runtime. If you performs the method, the result is:



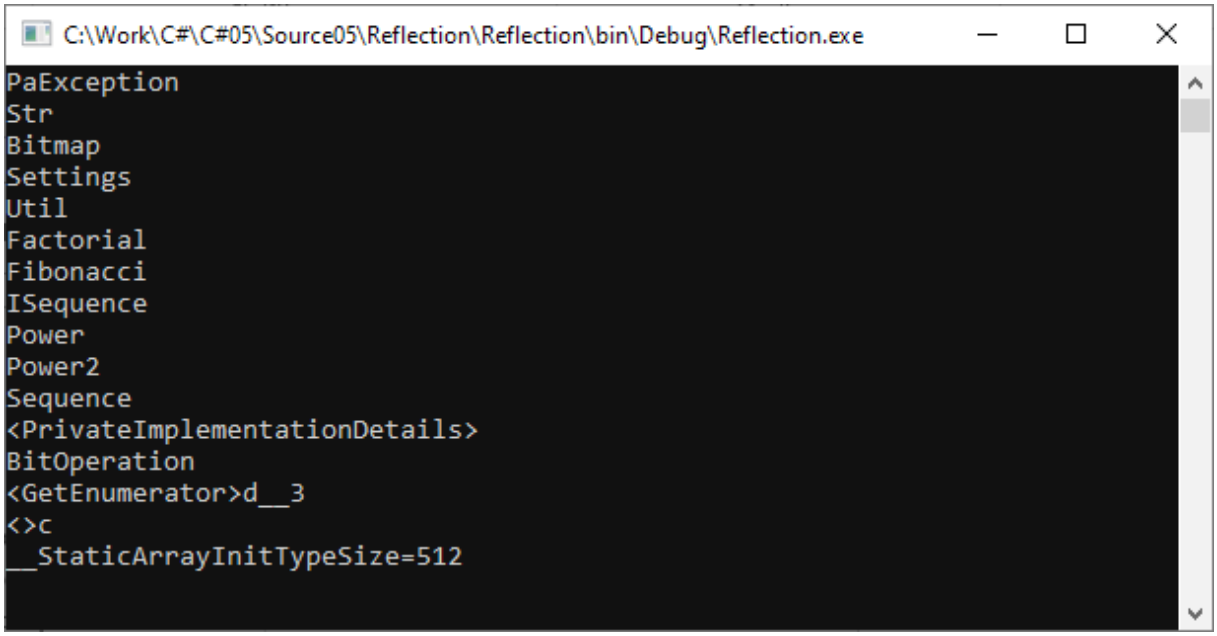
You can also load a shared assembly. If you opens a program in Ildasm which uses *PaLib* (for example *Eratosthenes2.exe*) you can see the assembly's strong name:



and the public key F988027234C6A27D and the version number 1:0:0:0. If I then expands the test program with the following methods

```
namespace Reflection
{
  class Program
  {
    ...

    static void Test04() {
      Assembly asm = LoadAssembly();
      if (asm != null) ShowTypeNames(asm);
    }

    private static void ShowTypeNames(Assembly asm)
    {
      Type[] types = asm.GetTypes();
      foreach (Type type in types) Console.WriteLine(type.Name);
    }

    private static Assembly LoadAssembly()
    {
      String name =
        "PaLib,Version=1.0.0.0,PublicKeyToken=
        F988027234C6A27D,Culture=\"\"";
      try
      {
        return Assembly.Load(name);
      }
      catch
      {
        Console.WriteLine("Assembly ikke fundet...");
        return null;
      }
    }
  }
}
```

and performs the method *Test04()* the program loads the shared assembly:



4.2 LATE BINDING

The above examples show how at runtime you can load an assembly and read its metadata. The next step is how to run assembly types and instantiate objects as well as call their methods at runtime. It is simple in principle, but the syntax is not entirely obvious.

I will again apply my shared assembly *PaLib* and the program instantiates a *Factorial* object and prints the factorials. The program also uses a method in the *String* class and the difference is that this time the program is written without knowledge of the assembly, but it is loaded dynamically at run time. The starting point is a usual console application, but this time there is no reference to the assembly - there must not be, because the meaning is precisely to write the program and translate it, without the metadata of the assembly is known. The code can be written as follows:

```
using System;
using System.Reflection;

namespace LateBinding
{
  class Program
  {
    static void Main(string[] args)
    {
      Assembly asm = LoadAssembly();
      Test1(asm);
      Test2(asm);
      Console.ReadLine();
    }

    private static void Test1(Assembly asm)
    {
      string name = "PaLib.Math.Sequences.Factorial";
      object fact = CreateObject(asm, name);
      MethodInfo Next = CreateMethod(asm, name, "Next");
      PropertyInfo Index = CreateProperty(asm, name, "Index");
      PropertyInfo Value = CreateProperty(asm, name, "Value");
      for (int i = 0; i <= 20; ++i)
      {
        Console.WriteLine("{0, 4}! = {1}",
          Index.GetValue(fact), Value.GetValue(fact));
        Next.Invoke(fact, null);
      }
    }

    private static void Test2(Assembly asm)
    {
      string name = "PaLib.Str";
      Type type = asm.GetType(name);
      object[] args = { "1234", 10, '*' };
      MethodInfo Fill = CreateMethod(asm, name, "Right", typeof(string),
        typeof(int), typeof(char));
      string text = Convert.ToString(Fill.Invoke(type, args));
      Console.WriteLine(text);
    }
```

```
    private static PropertyInfo CreateProperty(Assembly asm, string name1,
      string name2)
    {
      Type type = asm.GetType(name1);
      return type.GetProperty(name2);
    }

    private static MethodInfo CreateMethod(Assembly asm, string name1,
      string name2, params Type[] args)
    {
      Type type = asm.GetType(name1);
      return type.GetMethod(name2, args);
    }

    private static object CreateObject(Assembly asm, string name)
    {
      Type type = asm.GetType(name);
      return Activator.CreateInstance(type);
    }

    private static Assembly LoadAssembly()
    {
      String name =
        "PaLib,Version=1.0.0.0,PublicKeyToken=
        F988027234C6A27D,Culture=\"\"";
      try
      {
        return Assembly.Load(name);
      }
      catch
      {
        Console.WriteLine("Assembly not found...");
        Environment.Exit(0);
        return null;
      }
    }
  }
}
```

The method *LoadAssembly()* is as in the previous example, and does nothing but load the assembly *PaLib* from the machine's GAC. If it fails, the program ends with an error message.

The method *Test1()* creates a *Factorial* object that occurs in the method *CreateObject()* which based on the assembly information and the full name of the type, creates objects with the class *Activator*, which has a static method *CreateInstance()* that creates an object. Note that the object in the program (the method *Test1()*) is known as an object and you cannot use it as a *Factorial* object as this type is not known.

After the object is created, *Test1()* uses two additional help methods to obtain objects (*MethodInfo* and *PropertyInfo* types) for respectively methods and properties. Note that this is done by means of reflection, which reads metadata in the assembly represented by the object *asm*. For example the object *Next* represents a method in the class *Factorial.* You should note that the method *CreateMethod()* has a last parameter that is a *param* parameter which is used to tell the types of the methods parameters, but in this case *Next()* (in the class *Factorial*) has no parameters. The method *Test1()* also creates references for the two properties *Index* and *Value*.

The next loop prints the first 21 factorials, and here you must note how to use a property and a method. For a property as for example *Index*, which has the type *PropertyInfo* is used a method *GetValue()* that returns the property's value, but it requires a parameter that is the object where to invoke the property. For a method as *Next* that has the type *MethodInfo* you call a method *Invoke()*. The first parameter is the object where to invoke the method, and the next (in this case *null*) is an array with arguments for the method.

The method *Test1()* thus primarily shows:

- how you dynamic on runtime can instantiate an object of a type in a dynamic loaded assembly
- how to perform an instance method on the object
- how to use a property on the object

The method *Test2()* uses the same dynamically loaded assembly, and it should show how to execute a static method and how to pass parameters to the method (the same syntax applies to an instance method).

If you need to pass parameters, it is as an array of the type *object*. In this case, the method is *Right()*, which has three parameters:

```
object[] args = { "1234", 10, '*' };
```

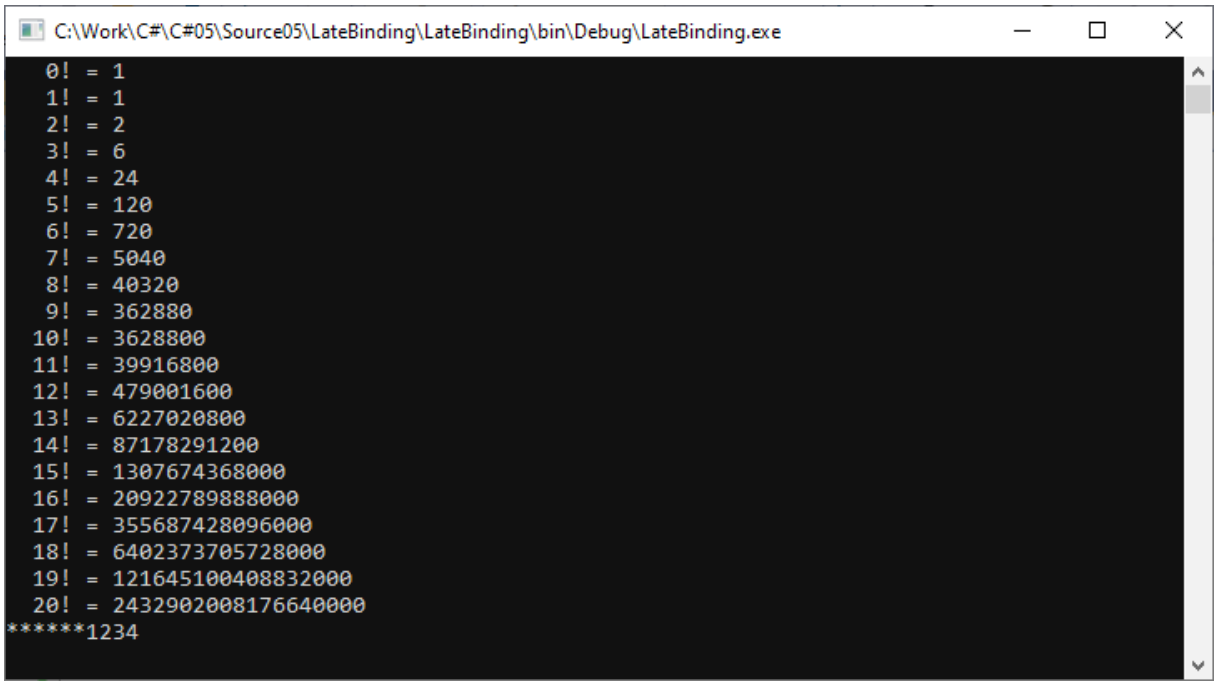
Then the method can be performed as:

```
string text = Convert.ToString(Fill.Invoke(type, args));
```

and here you should primarily notice that the first parameter of the method *Invoke()* is the type and not an object as for an instance method. In addition, the last parameter could be defined inline as an array of arguments:

```
string text =
Convert.ToString(Fill.Invoke(type, new object[] { "1234", 10, '*' }));
```

To sum it all up, the program uses the types in the assembly *PaLib* without the compiler knowing these types. The entire type information is determined at runtime. Of course, there is no benefit in this case, and the contrary, since the compiler cannot type check the code, but it opens up to be able to use code where there is no type description. Reflection is used for example in connection with what is called interop, where a .NET program must use COM objects. If you run the above program the result is:



4.3 ATTRIBUTES

Lastly, I will refer to attributes that can best be described as a way in which the programmer can manually add information to an assembly in the form of metadata. Attributes allow you to decorate program elements such as types, methods, variables, and so on and thus insert information into the assembly. An attribute is actually a class (translated into a class) that inherits the class *System.Attribute*.

There are basically two kinds of attributes

- predefined attributes in the .NET framework
- user defined attributes

There are many of the first kind and, for example, can be mentioned *Serializable*, which is mentioned in C# 1 in connection with files and is an attribute that tells a *Formatter* that an object can be serialized. You must use a special naming convention where the name of an attribute ends with *Attribute*, for example *SerializableAttribute*, but to simplify the notation, the C# compiler allows you to simply write *Serializable*, for example

```
[Serializable]
public class Person
{
  ....
```

instead of

```
[SerializableAttribute]
public class Person
{
  ....
```

Attributes are nothing in themselves, and as mentioned, it is simply a matter of inserting metadata into the assembly. This data is not used for anything unless there is a program that uses it, and for example, a binary formatter examines whether an object can be serialized. The .NET framework has a number of other tools that utilize various attributes and especially the compiler. There is for example an attribute called *CLSCompliant* that specifies that a particular program element complies with the CLS (*Common Language Specification*) specification, and if an element is decorated with that attribute, the compiler will check whether the specification is respected and if not provide a warning.

As an example I have created a class library called *DataLib* and the library contains the following class which represents a Danish King:

```
[Serializable]
public class King : IComparable<King>
{
  public int From { get; private set; }
  public int To { get; private set; }
  public string Name { get; private set; }

  public King(string name, int from = 0, int to = 9999)
  {
    Name = name;
    From = from;
    To = to;
  }

  public int CompareTo(King king)
  {
    if (From == 0 || king.From == 0) return To.CompareTo(king.To);
    return From.CompareTo(king.From);
  }

  public override bool Equals(object obj)
  {
    if (!(obj is King)) return false;
    King king = (King)obj;
    return Name.Equals(king.Name) && From == king.From && To == king.To;
  }

  public override int GetHashCode()
  {
    return Name.GetHashCode() ^ From.GetHashCode() ^ To.GetHashCode();
  }

  public override string ToString()
  {
    return string.Format("{0, -40} {1, 4} - {2, 4}",
      Name, From > 0 ? "" + From : "", To < 9999 ? "" + To : "");
  }
}
```

There is not much to explain as the class has just three properties that are the name of the king as well as the period when the king reigned. The most important thing is that the class is decorated with an attribute *Seralizable*. Specifically, note how to specify an attribute in square brackets. Also note that the class overrides methods from the class *object* and that two objects are the same if the value of all fields are the same. Finally, note that the class implements the interface *IComparable* so that objects can be compared. In this case, the comparison is made for the year of the start of the reign. There is a particular problem when not necessarily knowing the reign (kings who lived long ago, and kings who still reign), where the year *From* is set to 0 if unknown and the year *To* is set to 9999 if unknown. It causes little problems with *CompareTo()*, and compares two kings, where one does not know the beginning of the reign, is compared instead at the end of the period. It may lead to (in the absence of information) the comparison giving unintended results.

Below is a class which represents Danish kings as an ordered list:

```
[Serializable]
public class Kings : IEnumerable<King>
{
  private List<King> list;
  public DateTime Created { get; private set; }

  public Kings()
  {
    DeSerialize();
  }

  public void Save()
  {
    Serialize();
  }

  public bool Add(string name, int from, int to)
  {
    King king = new King(name, from, to);
    if (list.Contains(king)) return false;
    for (int i = 0; i < list.Count; ++i)
      if (king.CompareTo(list[i]) <= 0)
      {
        list.Insert(i, king);
        return true;
      }
    list.Add(king);
    return true;
  }
```

```
public bool Remove(King king)
{
  return list.Remove(king);
}

public IEnumerator<King> GetEnumerator()
{
  return list.GetEnumerator();
}

System.Collections.IEnumerator System.Collections.IEnumerable.
GetEnumerator()
{
  return GetEnumerator();
}

private void Serialize()
{
  BinaryFormatter bf = new BinaryFormatter();
  FileStream stream = File.Create("Kings.dat");
  bf.Serialize(stream, this);
  stream.Close();
}

private void DeSerialize()
{
  try
  {
    BinaryFormatter bf = new BinaryFormatter();
    FileStream stream = File.OpenRead("Kings.dat");
    Kings kings = (Kings)bf.Deserialize(stream);
    stream.Close();
    list = kings.list;
    Created = kings.Created;
  }
  catch
  {
    CreateList();
  }
}
```

```
  private void CreateList()
  {
    Created = DateTime.Now;
    list = new List<King>();
    for (int i = 0; i < data.Length; ++i)
    {
      try
      {
        string[] elem = data[i].Split('-');
        int from = int.Parse(elem[0].Trim());
        int to = int.Parse(elem[1].Trim());
        string name = elem[2].Trim();
        list.Add(new King(name, from, to));
      }
      catch
      {
      }
    }
    list.Sort();
  }

  #region
  private static string[] data =
  {
    "0000 - 0958 - Gorm den Gamle",
    "0958 - 0986 - Harald 1. Blåtand",
    "0986 - 1014 - Svend 1. Tveskæg",
    ...
    "1972 - 0000 - Margrethe 2. "
  };
  #endregion
}
```

The kingdom itself is laid out as a static array with strings at the end of the class. This array is used in the method *CreateList()* to create a list of *King* objects, where each line in the array is parsed to an object. After the list is created, it is sorted so that the objects are arranged similar to *CompareTo()* in the King class. In addition, the property *Created* is initialized with the time when the list is created.

The class has a method *Save()*, which stores the list in a file by serializing the object. This is done in the method *Serialize()*, which serializes the current object. Here it is the crucial that the class *Kings* are decorated *Serializable* as a *BinaryFormatter* reads this information. When the object is serialized, all instance variables are serialized to the extent that their types are

decorated *Serializable*, and this applies to all the simple types, but also the type *DateTime* and the type *List*. In order for the list to be serialized properly, it must contain objects that are *Serializable* and this applies to the type *King*. A *King* object can be serialized because it is decorated *Serializable* and because it consists of *Serializable* variables in the form of *int* and *string*. The result is that the formatter serializes the entire object structure to a file. In particular, note that the array with the strings is not serialized as it is a static variable.

The class's constructor calls the method *DeSerialize()*, which attempts to deserialize the object from a file. If it is not possible (if the file does not exist or it contains data that cannot be deserialized to a *Kings* object), the method calls *CreateList()*, which is initialized from the static array.

The class also implements methods to add new kings to the list and delete a king. Also note that the class overrides the interface *IEnumerable* so that a *Kings* object can be traversed with a *foreach* statement.

The class *Kings* is an example of a class that represents a relatively stable collection of objects, but allows the content to be modified and new objects added, and at the same time, the changes can be made persistently, so changes are saved between two runs of the program. The object is stored in a file *Kings.dat*, which must be in the program directory (and must be copied with the program if copied to another location). Also note that if the file is deleted, it will be re-created from the static array. The class is thus an example of a persistent object that can be used in situations where the object is rarely changed, but it is not the solution for programs that have to modify data on a continuous basis (transaction-heavy programs). Here you must use databases, which are the theme of the next book.

Below is a program that uses the class library. There is not much to explain, but note that the program saves the object by calling the method *Save()*:

```
class Program
{
  static void Main(string[] args)
  {
    Kings kings = new Kings();
    Console.WriteLine(kings.Created.ToLongDateString() + " " +
                      kings.Created.ToLongTimeString());
    kings.Add("Hardeknud", 0, 910);
    foreach (King king in kings) Console.WriteLine(king);
    kings.Save();
  }
}
```

It is possible to define your own attributes and I will start with the above class library *DataLib*. Here I have added the following classes:

```
using System;

namespace DataLib
{
  [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method |
    AttributeTargets.Property, AllowMultiple = true, Inherited = false)]
  public class Changes : Attribute
  {
    public DateTime Modified { get; private set; }
    public string Change { get; private set; }
    public string Issue { get; set; }

    public Changes(string modified, string change, string issue = "")
    {
      Modified = DateTime.Parse(modified);
      Change = change;
      Issue = issue;
    }
  }

  [AttributeUsage(AttributeTargets.Assembly)]
  public class SupportsChanges : Attribute
  {
  }
}
```

The first class is called *Changes* and defines an attribute. In particular, you should note that the class inherits the class *Attribute*. The class is simple and consists of three properties alone. The class itself is decorated with an attribute and it indicates

- that the attribute *Changes* can be used to decorate the program elements classes, methods and properties
- that the attribute must occur several times in front of a program element
- that the attribute should not be inherited to derived classes

The goal of this attribute is to be able to enter metadata related to changes to a class.

The second class defines an attribute called *SupportsChanges*, and it is an attribute that can only be used for an assembly. The class has no variables or methods, so it is a simple marker and as such can only be used to classify an assembly.

As an example is shown how these attributes are used for the class *Kings* (note that I've included only the part of the code where I added attributes):

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

[assembly: DataLib.SupportsChanges]
namespace DataLib
{
  [Changes("22-02-2020", "The class is extended with a new method")]
  [Changes("01-03-2020", "The class is extended with a new property")]
  [Changes("01-03-2020", "The class is extended with an index operator")]
  [Serializable]
  public class Kings : IEnumerable<King>
  {
    private List<King> list;
    public DateTime Created { get; private set; }

    public Kings()
    {
      DeSerialize();
    }

    public void Save()
    {
      Serialize();
    }

    [Changes("22-02-2020",
      "Method that returns all kings where the name contains a value",
      Issue =
      "The purpose is to make it possible to search kings with a
      particular name")]
    public King[] Find(string name)
    {
      List<King> kings = new List<King>();
      foreach (King k in this) if (k.Name.Contains(name)) kings.Add(k);
      return kings.ToArray();
    }

    [Changes("01-03-2020", "Returns the number of objects")]
    public int Count
    {
```

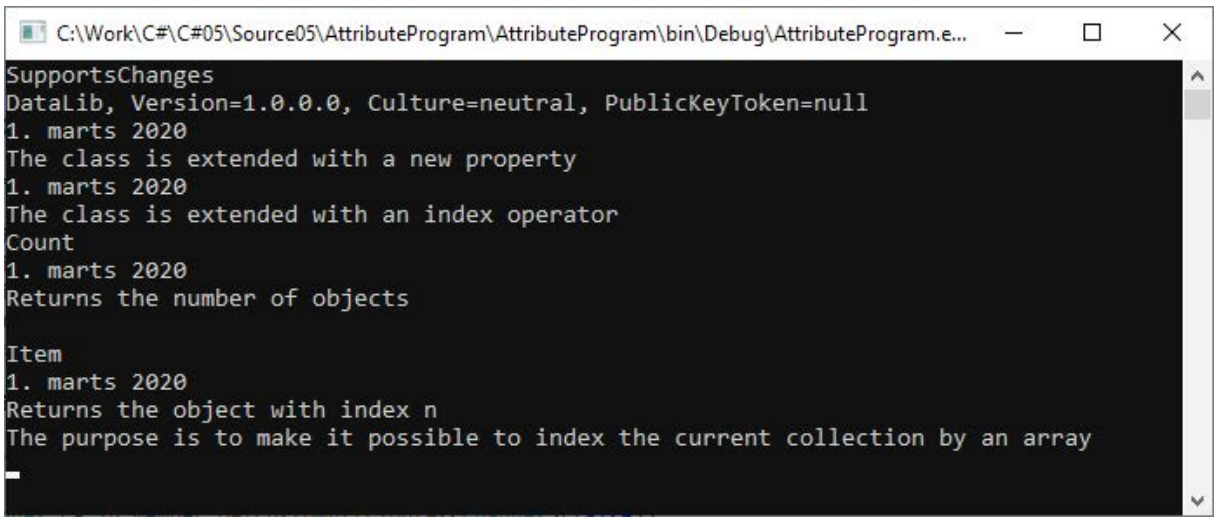
```
      get { return list.Count; }
    }

    [Changes("01-03-2020", "Returns the object with index n",
  "The purpose is to make it possible to index the current collection
  by an array")]
    public King this[int n]
    {
    get { return list[n]; }
    }

    ...
  }
}
```

First, note how to specify an attribute for the assembly. It requires a special syntax and it must precede the name of the assembly namespace. The natural interpretation of this attribute is that it is an assembly that supports (applies) *Changes* attributes, but it is an interpretation left to the program that uses the specific assembly.

The class *Kings* is decorated with three attributes of type *Changes*, which allows you to insert code into the metadata of the assembly, which tells what changes were made to the class. In this case, it is a history of what has happened. The class is extended with two properties and one method, and they are correspondingly decorated with an attribute that explains the purpose of the change. Again, it is important to emphasize that these attributes are not applied to anything, unless there is a program that pulls them out using reflection. As an example, below is the code for a program called *AttributeProgram*:

```
using System;
using System.Reflection;
using DataLib;

namespace Exam88
{
  class Program
  {
    static void Main(string[] args)
    {
      DateTime date = new DateTime(2020, 2, 25);
      Assembly asm = Assembly.Load("DataLib");
      Attribute attribute =
        Attribute.GetCustomAttribute(asm, typeof(SupportsChanges));
      Console.WriteLine(attribute.GetType().Name);
      Console.WriteLine(asm.FullName);
      Type type = asm.GetType("DataLib.Kings");
      foreach (Changes ca in type.GetCustomAttributes<Changes>(false))
      {
        if (ca.Modified > date)
          Console.WriteLine("{0}\n{1}", ca.Modified.ToLongDateString(),
          ca.Change);
      }
      foreach (MemberInfo member in type.GetMembers())
      {
        foreach (Changes ca in member.GetCustomAttributes<Changes>(fal
        se))
        {
          if (ca.Modified > date)
            Console.WriteLine("{0}\n{1}\n{2}\n{3}", member.Name,
                    ca.Modified.ToLongDateString(), ca.Change, ca.Issue);
        }
      }
    }
  }
}
```

The program has a reference to the assembly *DataLib* and the program loads this assembly. Next it prints that there is an assembly attribute *SupportsChanges* as well as the full name of the assembly. As a next step, a *Type* object is instantiated for the *Kings* class and the program prints all custom attributes of type *Changes*, which have a date after the value of the variable *date*.



4.4 USING REFLECTION

Above, I have shown how to use reflection to read an assembly's metadata, load an assembly at runtime, and even dynamically instantiate objects based on assembly types. I have also shown how to decorate types and other program elements with attributes, including how to write custom attributes. One thing is that it is possible, but another thing is what it can then be used for. Here, the main application is that it allows writing programs that can be dynamically expanded without having to change the existing program. This means that "others" can extend a program with new features if you meet certain standards. In this section, I will show how this is possible based on the class library *PaLib*.

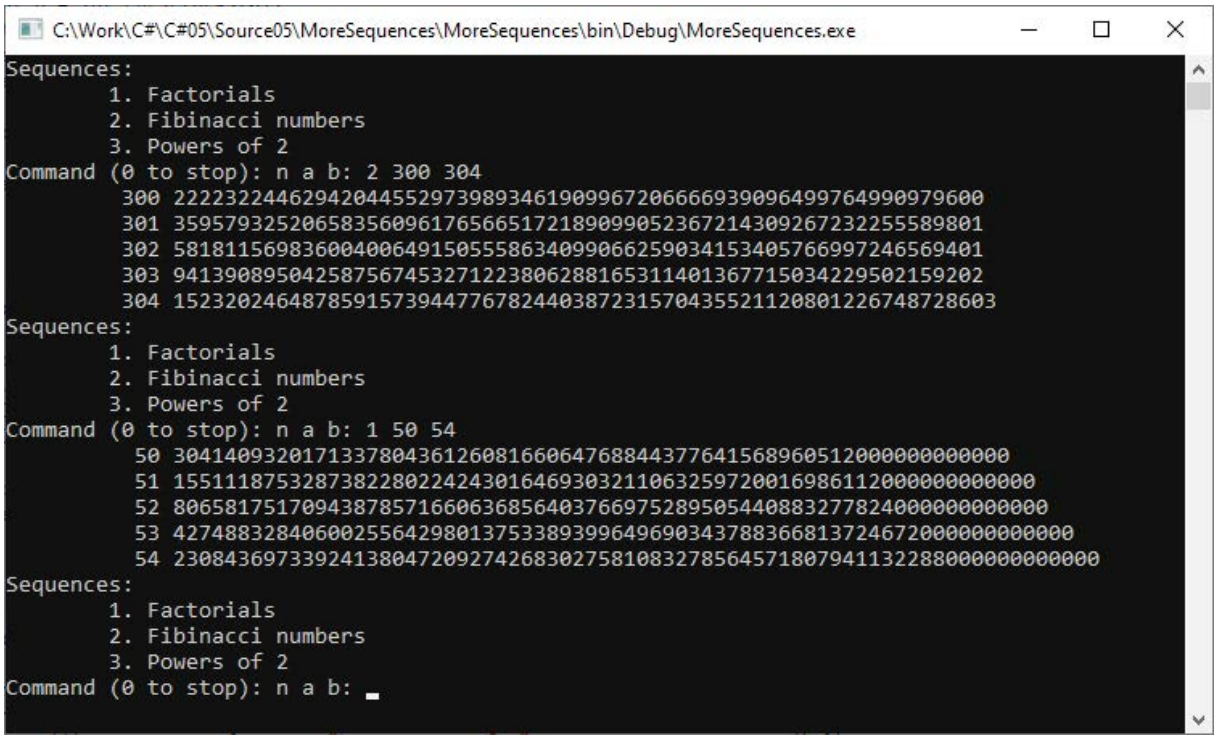
The class library *PaLib* implements four sequences:

- Factorial
- Fibonacci
- Power
- Power2

In the following program program I will only use the first and the last. The task is to write a console program where the user from a simple menu must select one of the three sequences as well as a range of values to print. As an example the following window shows a run of the program where the user has

1. printed the *Fibonacci* numbers with index between 300 and 304
2. printed the *Factorials* with index between 50 and 54

The program is called *MoreSequences* and is quite simple and can be written as follows:



```
class Program
{
  static List<MenuItem> items = new List<MenuItem>();

  static void Main(string[] args)
  {
    Build();
    Menu();
  }

  static void Menu()
  {
    while (true)
    {
      Console.WriteLine("Sequences:");
      foreach (MenuItem item in items) Console.WriteLine(item.Text);
      Console.Write("Command (0 to stop): n a b: ");
      string cmd = Console.ReadLine();
      if (!Parse(cmd)) Console.WriteLine("Illegal command...");
    }
  }

  static bool Parse(string line)
  {
    try
    {
      string[] args =
        line.Split(new char[] { ' ' }, StringSplitOptions.
        RemoveEmptyEntries);
      int n = int.Parse(args[0]);
      if (n == 0) Environment.Exit(0);
      Calculate(n, int.Parse(args[1]), int.Parse(args[2]));
      return true;
    }
    catch
    {
      return false;
    }
  }
```

```
  static void Calculate(int n, int a, int b)
  {
    ISequence seq = items[n - 1].Sequence;
    seq.Reset();
    while (seq.Index < a) seq.Next();
    while (seq.Index <= b)
    {
      Console.WriteLine(string.Format("{0, 12} {1}", seq.Index, seq.Value));
      seq.Next();
    }
  }

  static void Build()
  {
    items.Add(new MenuItem { Text = "\t1. Factorials",
      Sequence = new Factorial() });
    items.Add(new MenuItem { Text = "\t2. Fibinacci numbers",
      Sequence = new Fibonacci() });
    items.Add(new MenuItem { Text = "\t3. Powers of 2",
      Sequence = new Power2() });
  }

  class MenuItem
  {
    public string Text { get; set; }
    public ISequence Sequence { get; set; }
  }
}
```

There is not much to explain about the code, but note the project needs a reference to *System.Numerics* and *PaLib*. The class has an inner class *MenuItem* which represents a function for the menu in the form of a text and an *ISequence* object. The menu itself is represented by a list of items that are initialized in the method *Build()*. The user interaction is very simple where the user has to enter a line consisting of three numbers separated by spaces. The first number specifies which sequence to use, the next number is the index of the first element of the sequence, and the last number is the index of the last element of the sequence. Simply pressing 0 will stop the program. In case of a mistake, you get a simple error message. Also note the method *Calculate()*, which uses the selected sequence.

The above program does not contain anything new and it is just a program that uses a class library. The program does not use reflection, but I will now expand the program with an opportunity to add new sequences. Of course, this can be done by adding new classes to *PaLib* and then expanding the *Build()* method accordingly, but I will go another way.

I will start by expanding *PaLib* with a custom attribute:

```
using System;

namespace PaLib.Math.Sequences
{
  [AttributeUsage(AttributeTargets.Class)]
  public sealed class ExternSequence : Attribute
  {
    public string AssemblyName { get; private set; }
    public string ClassName { get; private set; }
    public string Description { get; private set; }

    public ExternSequence(string assemblyName, string className,
                          string description)
    {
      AssemblyName = assemblyName;
      ClassName = className;
      Description = description;
    }
  }
}
```

It is an attribute that can be used to decorate a class with three names:

- the name of the class's assembly
- the full name of the class
- a description

Note that the change requires the GAC to be updated.

As the next step, I will write three new sequences. The three sequences are placed in a new class library, which I have called *SequenceLib*. There must be a reference to *PaLib*. The first sequence represents the even numbers:

```
using System.Numerics;
using PaLib.Math.Sequences;

namespace SequenceLib
{
  public class Even : Sequence
  {
    public Even() : base("Even", BigInteger.Zero)
    {
    }

    public override void Reset()
    {
      index = 0;
      value = BigInteger.Zero;
    }

    public override void Next()
    {
      value += 2;
      ++index;
    }

    public override void Prev()
    {
      if (index > 0)
      {
        value -= 2;
        --index;
      }
    }
  }
}
```

Note that the class is written in exactly the same way as the other sequences and as a class that inherits the abstract class *Sequence*. The next sequence represents square of the index:

```
using System.Numerics;
using PaLib.Math.Sequences;

namespace SequenceLib
{
  [ExternSequence("SequenceLib", "SequenceLib.Squares", "Square numbers")]
  public class Squares : Sequence
  {
    public Squares() : base("Squares", BigInteger.One)
    {
    }

    public override void Reset()
    {
      index = 0;
      value = BigInteger.One;
    }

    public override void Next()
    {
      BigInteger val = new BigInteger(++index);
      value = val *val;
    }

    public override void Prev()
    {
      if (index > 0)
      {
        if (index == 1) Reset();
        else
        {
          BigInteger val = new BigInteger(--index);
          value = val * val;
        }
      }
    }
  }
}
```

Also this class is written in the same way as the other sequences, but you must note the attribute, which is the attribute from the class library *PaLib*. The last sequence is for the power of 3:

```
using PaLib.Math.Sequences;

namespace SequenceLib
{
  [ExternSequence("SequenceLib", "SequenceLib.Power3", "The powers of 3")]
  public class Power3 : Power
  {
    public Power3() : base(3, "Power of 3")
    {
    }
  }
}
```

and also this class is decorated with an attribute.

To use the new sequences I have created a copy of the project *MoreSequences* and called the copy *MoreSequences1*. To the application folder (the folder for the exe file) I then create a subdirectory called *Extensions* and copy the class library *SequenceLib.dll* to this subdirectory. Note that it is now a private assembly that the program can find (probe) if the config file is updated. As a next step, I have therefore modified *App.config* (the new project *MoreSequences1*):

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.5" />
  </startup>
  <runtime>
    <assemblyBinding xmlns="urn:schemas-microsoft-com:asm.v1">
      <probing privatePath="Extensions"/>
    </assemblyBinding>
  </runtime>
  <appSettings>
    <add key="Sequences" value="SequenceLib"/>
  </appSettings>
</configuration>
```

Here I first specified that the program should probe the folder *Extensions* for private assemblies. Next, in an *appSettings*, I have defined a value in the form of a key / value pair. It is a value that the program can load and thus it shows how it is possible to define names in a config file that the program can read. In this case, it is the name of the assembly in the folder *Extensions*.

As a final point, the program is extended by the following method, which is called as the last statement in the method *Build()*:

```
static void AddModules(int n)
{
  System.Configuration.Configuration config =
    ConfigurationManager.OpenExeConfiguration(ConfigurationUserLevel.None);
  foreach (KeyValueConfigurationElement kv in config.AppSettings.Settings)
  {
    try
    {
      Assembly asm = Assembly.Load(kv.Value);
      Type[] types = asm.GetTypes();
      foreach (Type type in types)
        if (type.IsClass)
        {
          try
          {
            ExternSequence attr = type.GetCustomAttribute<ExternSequen
            ce>();
            items.Add(new MenuItem {
              Text = string.Format("\t{0}. {1}", n, attr.Description),
              Sequence = (AbstractSequence)Activator.CreateInstance(type)
              });
            ++n;
          }
          catch
          {
          }
        }
    }
    catch
    {
    }
  }
}
```

It is a relatively complex method, but note first that a reference to the assembly *System.Configuration* is required. The method starts by creating a *Configuration* object, which is an object that represents the *App.config* configuration file. Next, all entries in the *appSettings* are looped (in this case, there is only one), and for each value the specified assembly is loaded. For that assembly, all types are looped, and if a type is a class type, it is examined whether there is an *ExternSequence* attribute (if not, you get an exception that the program simply ignores). If so, the sequence is added to the menu. The result is that two new menu items have now been added, as the sequence *Even* is not included as it is not decorated with an *ExternSequence* attribute.

The example *MoveSequences1* is an example of a program that can be expanded by simply copying an assembly to the program library and then updating the program's config file. This really means that other than those who have created the program or the assembly *PaLib* can add new functionality to the program.

EXERCISE 6: THE SEQUENCES PROGRAM

In the book C# 1 (Problem 1) you have written a program where the user can select a sequence and then determines values for the sequence. Make a copy of the project. Open the project in Visual Studio and runs the program to remember what the program is doing.

The project contains the 6 sequence types. Remove these types from the project. You should then add a reference to the class library *PaLib* in the GAC (and set *Copy Local* to *false*). In code behind for *MainWindow* you must write a *using* statement for the class library and then the program should be able to run again.

You must then copy the folder *Extensions* with the class library *SequenceLib.dll* from the previous project to the application folder for the current project. You must also update *App.config* as above to probe the class library.

You must then change the program such it no longer use the sequense *Power*, but instead uses the two decorated sequences in *SequenceLib* and when you must expand the program with the two new sequences in the same way as above using reflection.

5 IO

In the book C# 1 I have processed files and in the above I have used files several times. However, there are more about files that are the subject of this chapter, and here are the most important binary files and more about serialization. However, I will not mention more about text files as there is no more to add, but you can for example start with the following exercise.

EXERCISE 7: A CSV FILE

A common use of text files is to transfer data between applications. An application can store information in a text file that can be sent to a recipient who then read the content of the file. Often that kind of files are called CSV files. CSV stands for *comma separated values*, noting that the file contains values that are separated by a comma. The separation character do not necessarily needs to be a comma and can be anything. The only requirement is that it is a character that cannot occur in the individual data elements (values). Besides comma are often used spaces, semicolons, tabulator, or equivalent, but whatever character you use the file is still called a CSV file. In this exercise you have to write a program that creates a CSV file and read it again.

Create a new project, you can call *CsvFiles*. Add a method that creates a text file which consists of lines that start with a date and is followed by one or more decimal numbers separated by semicolons. The start of the file could, for example be:

```
19.09.2020;218,85;125,41;300,33;411,16;647,90;233,54;292,96;523,60;635
,01;311,16
13.03.2015;123,49;776,16;833,05;734,20;973,97;298,97;584,56;188,92
08.06.2019;41,93;757,28;608,09;930,42;618,31;447,46;480,18;239,52;462,
23;683,84
03.07.2019;200,23;774,34
12.04.2015;636,14
```

A line could, for example be interpreted as a number of amounts (for example product sales) as concerning a certain date. You should note, that there can be several lines with the same date and in any order. The difficult thing is not to create the file, but to create the lines with random values to be printed to the file, so you should create some helper methods.

You must then write a method that reads the content of the file, and prints a list as shown below:

```
02.01.2015: 1848,44
04.01.2016: 3303,82
05.01.2016:  756,04
08.01.2017: 9773,91
```

The table must show the total sales for each date, and the lines must be sorted by date.

5.1 BINARY FILES

Just as there are classes for working with text files, there are classes for processing binary files that is files containing numbers or other binary data. The program *BinaryFiles* shows how to use binary files. The method *Test1()* is an example:

```
static void Test1()
{
  FileStream stream = new FileStream("C:\\Temp\\Numbers.dat",
    FileMode.OpenOrCreate, FileAccess.Write);
  BinaryWriter writer = new BinaryWriter(stream);
  for (int i = 0; i < 100; ++i) writer.Write(rand.NextDouble() * 10000);
  writer.Close();
  stream.Close();
}
```

This time the file is represented by a *FileStream*. In addition to the file name, the constructor has two additional parameters. The file's mode specifies how to open the file. Here it is stated that it must be opened, if it exists, and otherwise it should be created. As an alternative you can specify

- *CreateNew*, that creates a new file, but if it already exists, you get an exception
- *Create*, that creates a new file and, if it already exists, overwrites it
- *Open*, that opens a file and if it does not exist, you get an exception
- *Truncate*, that opens a file and deletes the previous content
- *Append*, that opens a file and places the file pointer at the end of the file - if the file does not exist, it is created

The last parameter specifies access to the file and opens the file for writing. Alternatively, the file can be opened for reading or both reading and writing.

After the file is opened, a *BinaryWriter* is created for the file, which is used to enter numbers into it. Note the method *Write()*, which is found in 18 overrides and thus one override for each of the built-in data types.

The next method reads the contents of the file created with the above method and prints the number of numbers read, the sum of the numbers, the average, and the smallest and largest numbers:

```
static void Test2()
{
  double sum = 0;
  double min = double.MaxValue;
  double max = double.MinValue;
  int count = 0;
  FileStream stream = new FileStream("C:\\Temp\\Numbers.dat",
    FileMode.Open, FileAccess.Read);
  BinaryReader reader = new BinaryReader(stream);
  try
  {
    while (true)
    {
      double tal = reader.ReadDouble();
      sum += tal;
      ++count;
      if (min > tal) min = tal;
      if (max < tal) max = tal;
    }
  }
  catch (EndOfStreamException)
  {
  }
  reader.Close();
  stream.Close();
  if (count > 0) Result(sum, min, max, count);
}

private static void Result(double sum, double min, double max, int count)
{
  Console.WriteLine("It has been read {0} numbers", count);
  Console.WriteLine("The sum of the numbers are {0}", sum);
  Console.WriteLine("The average is {0}", sum / count);
  Console.WriteLine("The minimum number is {0}", min);
  Console.WriteLine("The maximum number is {0}", max);
}
```

Basically, there is not much new, but there are two things to be aware of. One is how to handle the end of the file, and thus when there are no more numbers in the file. To read the file a *BinaryReader* is used and if you try to read beyond the end of the file you get an exception. The other thing to note is how to read the file using the method *ReadDouble()*. This method reads the next 8 bytes and converts them into a *double* whatever content. Thus, it is the responsibility of the user (the user must know) that these 8 bytes actually represents a *double*. This information is not stored in the file. The *BinaryReader* class has similar methods for reading other built-in types.

A *FileStream* is always a sequence of bytes, and it is up to the program that reads the file to interpret the content correctly. Attached to a *FileStream* is a so-called file pointer, which is simply an integer that specifies the position in the file where the next read or write operation takes place. The program can manipulate this file pointer with a method *Seek()* that can position the file pointer anywhere - again it is up to the program to make sure it is a sensible place. The next example shows how to move the file pointer.

```
static void Test3()
{
  FileStream stream = new FileStream("Data.dat", FileMode.Create,
    FileAccess.ReadWrite);
  BinaryWriter writer = new BinaryWriter(stream);
  BinaryReader reader = new BinaryReader(stream);
  for (int i = 0; i < 10; ++i) writer.Write(Math.Sqrt(i));
  stream.Seek(5 * sizeof(double), SeekOrigin.Begin);
  writer.Write(Math.PI);
  Console.WriteLine(reader.ReadDouble());
  stream.Seek(5 * sizeof(double), SeekOrigin.End);
  writer.Write(Math.E);
  stream.Seek(0, SeekOrigin.Begin);
  try
  {
    while (true) Console.WriteLine(reader.ReadDouble());
  }
  catch (EndOfStreamException)
  {
  }
  reader.Close();
  writer.Close();
  stream.Close();
}
```

First, a *FileStream* is opened, but this time with read-write access. Also note that its mode is *Create*, so that the file is created each time the method is executed. As a next step, both a writer and a reader are associated, and 10 numbers are written to the file. Next, the number *pi* is written in the file, but first the file pointer is placed, then the number is written as the 5th number (the first number has the position 0). Since a double occupies 8 bytes, the number must start with the byte at position 40. After *pi* is written, a number is read, but as a write moves the file pointer 8 bytes, it becomes the next number to read. Now the file pointer is moved again 40 bytes, but this time from the end of the file. That is the file pointer is placed 40 bytes after the file and a number is written. The question is what happens to the intermediate spaces that are basically empty. The answer is that they are set to 0 - all bytes are 0. Finally, the entire contents of the file are read. Note how the file pointer is first set to the start of the file.

If the method is performed, the result is as follows:

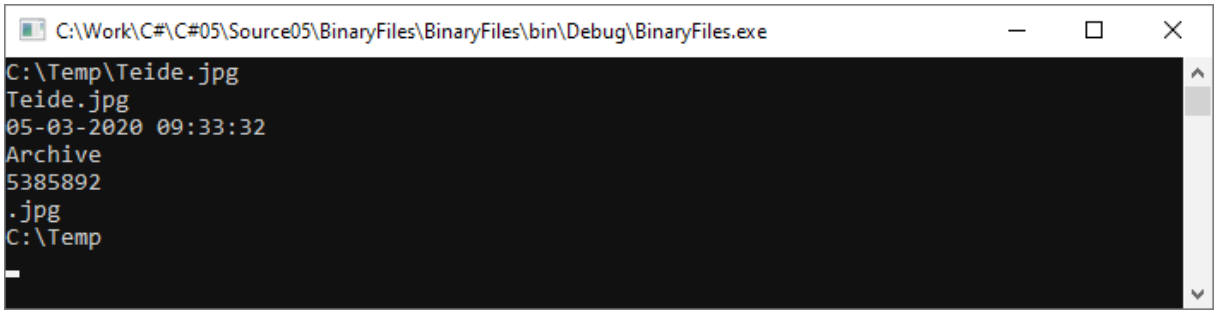
5.2 INFO ABOUT DIRECTORIES AND FILES

System.IO contains two classes that are useful for determining information about files and directories:

- *FileInfo*
- *DirectoryInfo*

The first represents a file, while the second represents a directory. There is not much to say about them, but they both have a number of properties that provide information about a specific file or directory. The following method (also a method in the program *BinaryFiles*) give some examples:

```
static void Test4()
{
  string filename = "C:\\Temp\\Teide.jpg";
  FileInfo info = new FileInfo(filename);
  Console.WriteLine(info.FullName);
  Console.WriteLine(info.Name);
  Console.WriteLine(info.CreationTime);
  Console.WriteLine(info.Attributes.ToString());
  Console.WriteLine(info.Length);
  Console.WriteLine(info.Extension);
  Console.WriteLine(info.DirectoryName);
}
```



There is nothing to explain, but you should note that the image *Teide.jpg* must be found in the folder *C:\Temp* and if you like you can test the method with another image.

The next example is basically the same as above, simply printing information about a directory instead:

```
static void Test5()
{
  string dirname = "C:\\Temp";
  DirectoryInfo info = new DirectoryInfo(dirname);
  Console.WriteLine(info.FullName);
  Console.WriteLine(info.Name);
  Console.WriteLine(info.CreationTime);
  Console.WriteLine(info.Attributes.ToString());
  Console.WriteLine(info.Root);
  Console.WriteLine("Subdirectories:");
  foreach (DirectoryInfo dir in info.GetDirectories()) Console.
  WriteLine(dir.Name);
  Console.WriteLine("Files:");
  foreach (FileInfo file in info.GetFiles()) Console.WriteLine(file.Name);
}
```

Note that the two classes *FileInfo* and *DirectoryInfo* have many other methods, such as methods that can determine the contents of a directory. The classes also have methods for manipulating the file system such as creating files and directories, deleting files and so on.

There are two other classes

- *File*
- *Directory*

which basically offers the same services as the classes *FileInfo* and *DirectoryInfo*, and the main difference is that the classes *File* and *Directory* offer the file operations as static methods. Therefore, they can sometimes be easier to apply and I will apply them in the last example of this chapter.

5.3 SERIALIZATION

Serialization of objects is discussed in the book C# 1, but there is a little more to add, what is the topic of what follows. Object serialization denotes the process by which an object's state is converted to a sequence of bytes, and thus to data that can be sent to a stream such as a file or over a communication line. It is possible to serialize arbitrary objects to a stream, and in order to read a serialized object again, it is necessary, together with the serialization of the object's data, to store information about the type. When reading a serialized object, it is called deserialization of the object.

Seen from the programmer, serialization and deserialization of objects is simple, and you don't have to write very much, but if you think about it a little, it is a relatively complex process. An object is defined on the basis of a class and can contain both simple variables, arrays and references to other objects, and not only that, the class can also inherit variables from a base class. When serializing an object, all of these elements must be stored along with all the information needed to rebuild the entire structure during deserialization. In conclusion, although the following examples show that it is simple to serialize objects, there is a great deal going on, which is taken care of by the classes that underlie serialization. The program *Seralization* shows what to do. The program has a class *Person*:

```
namespace Serialization
{
  [Serializable]
  public class Person
  {
    public string Name { get; set; }
    public string Position { get; set; }

    public Person()
    {
      Name = "";
      Position = "";
    }

    public Person(string name, string position)
    {
      Name = name;
      Position = position;
    }

    public override string ToString()
    {
      return Name + ", " + Position;
    }
  }
}
```

The class is defined *Serializable* and when it contains two serializable fields an object of the type *Person* can then as shown in C# 1 be serialized to a file (the program *Serialization*):

```
static void Test1()
{
  Person pers = new Person("Valborg Kristensen", "Witch");
  Serialize1(pers);
}

static void Serialize1(Person pers)
{
  BinaryFormatter bf = new BinaryFormatter();
  FileStream stream = File.Create("C:\\Temp\\Person.dat");
  bf.Serialize(stream, pers, null);
  stream.Close();
}

static void Test2()
{
  Person pers = DeSerialize1();
  if (pers != null) Console.WriteLine(pers);
}

static Person DeSerialize1()
{
  Person pers = null;
  try
  {
    BinaryFormatter bf = new BinaryFormatter();
    FileStream stream = File.OpenRead("C:\\Temp\\Person.dat");
    pers = (Person)bf.Deserialize(stream);
    stream.Close();
  }
  catch
  {
  }
  return pers;
}
```

In this example is used a *BinaryFormatter* where the object is stored in a binary format. You can also use an *XmlSerializer* and the only difference is the use of this formatter instead of a *BinaryFormatter*:

```
static void Test3()
{
  Person pers = new Person("Knud Andersen", "Executioner");
  Serialize2(pers);
}

private static void Serialize2(Person pers)
{
  XmlSerializer xs = new XmlSerializer(typeof(Person));
  FileStream stream = File.Create("C:\\Temp\\Person.xml");
  xs.Serialize(stream, pers);
  stream.Close();
}

static void Test4()
{
  Person pers = DeSerialize2();
  if (pers != null) Console.WriteLine(pers);
}

static Person DeSerialize2()
{
  Person pers = null;
  try
  {
    XmlSerializer xs = new XmlSerializer(typeof(Person));
    FileStream stream = File.OpenRead("C:\\Temp\\Person.xml");
    pers = (Person)xs.Deserialize(stream);
    stream.Close();
  }
  catch
  {
  }
  return pers;
}
```

The advantage of serializing an object with an *XmlSerializer* is that the file can be opened with any program that can open a text file.

To serialize a *Person* object it is necessary to store the two member variables *name* and *position*, and they themselves are objects of the type *string* which is also *Serializable*. One thing is to save the object, but since you must also be able to read the object again, as mentioned above, it is also necessary to store information about the object type. What is stored, is raw bytes and in order for it to be re-loaded as a *Person* object consisting of two other objects

of the type *string*, this information must be stored with the object's data bytes. This process is called serialization, and for this to be possible, the object's type must be *Serializable* and all the object instance variables must be *Serializable*. In this case, it is all met as *Person* is defined *Serializable* and the member variables have the type *string* which is *Serializable*. The same goes for all the simple types and many of the types in the .NET framework.

To serialize an object, you need to use a so-called *Formatter*. There are several options, and the different *Formatter* types are defined in the namespace *System.Runtime.Formatters* or a nested namespace, but common to them is that they serialize an object with the information needed for the object to be deserialized. In contrast, they differ in the format in which the object is stored. In this case, I first use a binary formatter that stores an object in a binary format, and next I used an XML formatter which store the object as text.

5.4 USER DEFINED SERIALIZATION

The serialization process as described is generally straightforward from the programmer, and there is rarely reason to do more than described above. However, it is possible to intervene in the process if there are special needs.

As an example, it is possible that a class has variables that for some reason you do not want to be serialized. Below is a class *Employee*, which inherits *Person* and expands with two variables, the first being the date of employment while the second being the current year. The class has a method *Seniority()* which - a little simplified - returns the seniority of an employee as the number of years between the year of employment and the current year. The class is serializable, but it makes no sense to serialize the variable *year*, since its value depends on when the program is run. However, one can label a variable with the attribute *NonSerialized* that tells it not to be serialized.

```
[Serializable]
public class Employee : Person
{
  private DateTime date;

  [NonSerialized]
  protected int year = DateTime.Now.Year;

  [OnDeserialized]
  public void InitYear(StreamingContext context)
  {
    year = DateTime.Now.Year;
  }

  public Employee(string name, string position, DateTime date)
    : base(name, position)
  {
    this.date = date;
  }

  public int Seniority()
  {
    return year - date.Year;
  }
}
```

There are other similar attributes that can be used to decorate methods with a similar signature:

- *OnDeserializing*, to indicate a method performed before the deserialization process
- *OnSerialized*, indicates a method performed immediately after the serialization process
- *OnSerializing*, which specifies a method performed before the serialization process

In addition, as illustrated above, methods can be defined that are performed before and after respectively serialization and deserialization, it is also possible to directly control the process of how data is serialized and deserialized. The principle is illustrated by the following class, which represents a member who is a specialization of the class *Person*:

```
[Serializable]
public class Member : Person, ISerializable
{
  private string ssn;
  private string phone;
  private string email;

  public Member(string ssn, string name, string position, string phone,
    string email) : base(name, position)
  {
    this.ssn = ssn;
    this.phone = phone;
    this.email = email;
  }

  public Member(SerializationInfo info, StreamingContext context)
  {
    ssn = info.GetString("ssn");
    Name = info.GetString("name");
    Position = info.GetString("position");
    phone = info.GetString("phone");
    email = info.GetString("email").ToLower();
  }

  public string Ssn
  {
    get { return ssn; }
    set { ssn = value; }
  }

  public string Phone
  {
    get { return phone; }
    set { phone = value; }
  }

  public string Email
  {
    get { return email; }
    set { email = value; }
  }
```

```
  public void GetObjectData(SerializationInfo info, StreamingContext context)
  {
    info.AddValue("ssn", FormatSsn());
    info.AddValue("name", Name);
    info.AddValue("position", Position);
    info.AddValue("phone", FormatPhone());
    info.AddValue("email", email.ToUpper());
  }

  private string FormatSsn()
  {
    StringBuilder builder = new StringBuilder();
    for (int i = 0; i < ssn.Length; ++i)
      if (ssn[i] >= '0' && ssn[i] <= '9')
      {
        builder.Append(ssn[i]);
        if (builder.Length == 6) builder.Append('-');
      }
    return builder.ToString();
  }

  private string FormatPhone()
  {
    StringBuilder builder = new StringBuilder();
    for (int i = 0, j = 0; i < phone.Length; ++i)
      if (phone[i] >= '0' && phone[i] <= '9')
      {
        builder.Append(phone[i]);
        ++j;
        if (j == 2)
        {
          builder.Append(' ');
          j = 0;
        }
      }
    return builder.ToString();
  }

  public override string ToString()
  {
    return string.Format("{0}\n{1}\n{2}\n{3}", ssn, base.ToString(), phone, email);
  }
}
```

The important thing to note is that the class implements the interface *ISerializable*, which is an interface that defines a single method: *GetObjectData()*. If a serializable class implements

this interface, a formatter will call the method *GetObjectData()*, and here you can then define in the form of key / value pairs how the individual elements should be serialized. A key is a string and often the name of a variable will be used, but it is not a requirement. In this case - just to do something - the method should ensure that a ssn number is stored as 6 digits, a hyphen and 4 digits, that a phone number is stored as pairs of digits separated by spaces, and that the email address is only in uppercase letters. Please note that the method does not check if the ssn number and phone number are legal. In practice, it should have been checked somewhere, but it's not here not to make the code more complex than necessary.

The class also has a new constructor that among other things has a *SerializationInfo* parameter. This constructor is needed to deserialize an *ISerializable* object, and here a number of get methods can obtain the values serialized in the *GetObjectData()* method. The use of the custom serialization happens in exactly the same way as the other serialization examples.

PROBLEM 3: DENMARK

The folder for this book contains three text files called

1. *regions*, that contains a line for each Danish region where a line consists of a region number and the name of the region separated by comma
2. *municipalities*, that contains a line for each Danish municipality, where the line consists of the municipality's number, the name of the municipality and the number of the region that the municipality belongs, followed by one or more zip codes indicating the zip codes used by this municipality and where all fields are separated by semicolons
3. *zipcodes*, that contains a line for each zip code, where the line consists of the code and the city name separated by semicolons

You must now create a new WPF project in Visual Studio, as you for example can call for *Denmark*. You must then add the following three interfaces to your project and write classes that implements these interfaces:

```
namespace Denmark
{
  /// <summary>
  /// Interface, that defines a region. Two regions are equal if
  /// they have the same region number. Regions are ordered ascending
  /// after name. The iterator pattern must be implemented to iterates
  /// this region's municipalities.
  /// </summary>
  public interface IRegion : IComparable<IRegion>,
  IEnumerable<IMunicipality>
  {
    /// <summary>
    /// Returns the region number for this region.
    /// </summary>
    int Rnr { get; }

    /// <summary>
    /// The name of this region.
    /// </summary>
    string Name { get; set; }

    /// <summary>
    /// The number of municipalities in the region.
    /// </summary>
    int Count { get; }

    /// <summary>
    /// Indexer for municipalities.
    /// </summary>
    /// <param name="n">The municipality's index</param>
    /// <returns>The municipality with index n</returns>
    IMunicipality this[int n] { get; }

    /// <summary>
    /// Returns the municipality with number (id) mnr.
    /// </summary>
    IMunicipality GetMunicipality(int mnr);

    /// <summary>
    /// Add a municipality to this region.
    /// </summary>
    void Add(IMunicipality municipality);
```

```
    /// <summary>
    /// Remove the municiplaity with number mnr.
    /// </summary>
    /// <param name="mnr">The municipality number</param>
    /// <returns>True if the municipality was removed</returns>
    bool Remove(int mnr);
  }
}

namespace Denmark
{
  /// <summary>
  /// Interface, that defines a municipality. Two municipalities are
      considered
  /// equal if they have the same municipality number. Municipalities
      are ordered
  /// ascending by municipality name. The iterator pattern must be
      implemented to
  /// iterates the zip code's used by this municipality.
  /// </summary>
  public interface IMunicipality : IComparable<IMunicipality>,
    IEnumerable<IZipcode>
  {
    /// <summary>
    /// The municipality number for this municipality.
    /// </summary>
    int Mnr { get; }

    /// <summary>
    /// The name for this municipality.
    /// </summary>
    string Name { get; set; }

    /// <summary>
    /// The region to which this municipality belongs.
    /// </summary>
    IRegion Region { get; set; }

    /// <summary>
    /// The number of zip codes used by this municipality.
    /// </summary>
```

```
    int Count { get; }

    /// <summary>
    /// Indexer for zip codes used by this municipality.
    /// </summary>
    /// <param name="n">Index for zip code</param>
    /// <returns>The zip code with index n</returns>
    IZipcode this[int n] { get; }

    /// <summary>
    /// Returns the zip code with the number code if this municipality
    /// uses this zip code. Else the method returns null.
    IZipcode GetZipcode(string code);

    /// <summary>
    /// Adds a zip code to this municipality, thus indicating a zip code
    /// that this municipality uses.
    /// </summary>
    void Add(IZipcode zipcode);

    /// <summary>
    /// Remove a zip code from this municipality, thus indicating that
this
    /// municipality does not use the code.
    /// </summary>
    /// <param name="code">The zip code to be removed</param>
    /// <returns>True if the zip code was removed</returns>
    bool Remove(string code);
  }
}

namespace Denmark
{
  /// <summary>
  /// Interface, that defines a zip code. Two zip codes are considered
equal
  /// if they have the same number. Zip codes are ordered ascending
      by the code.
  /// The iterator pattern defines that you can iterates the municipalities,
  /// that use this zip code.
  /// </summary>
  public interface IZipcode : IComparable<IZipcode>,
```

```
    /// <summary>
    /// The code for this zip code.
    /// </summary>
    string Code { get; }

    /// <summary>
    /// The city name for this zip code.
    /// </summary>
    string City { get; set; }

    /// <summary>
    /// Number of municipalities that uses this zip code.
    /// </summary>
    int Count { get; }

    /// <summary>
    /// Indexer for municipalities that uses this zip code.
    /// </summary>
    /// <param name="n">Index for municipality</param>
    /// <returns>The municipality with index n</returns>
    IMunicipality this[int n] { get; }

    /// <summary>
    /// Returns the municipality with number mnr if the municipality
    /// uses this zip code. Else the method retursn null.
    /// </summary>
    IMunicipality GetMunicipality(int mnr);

    /// <summary>
    /// Adds a municipality to this Zipcode object and thus indicates
that
    /// this zip code is used by the municipality.
    /// </summary>
    void Add(IMunicipality municipality);

    /// <summary>
    /// Removes the municipality with the number mnr from this zip code
and
    /// thus indicates that the municipality does not use this zip code.
    /// </summary>
    /// <param name="mnr">The municipality number</param>
    /// <returns>True if the municipality was removed</returns>
    bool Remove(int mnr);
```

You must then write the following class, when the class should be written as a singleton:

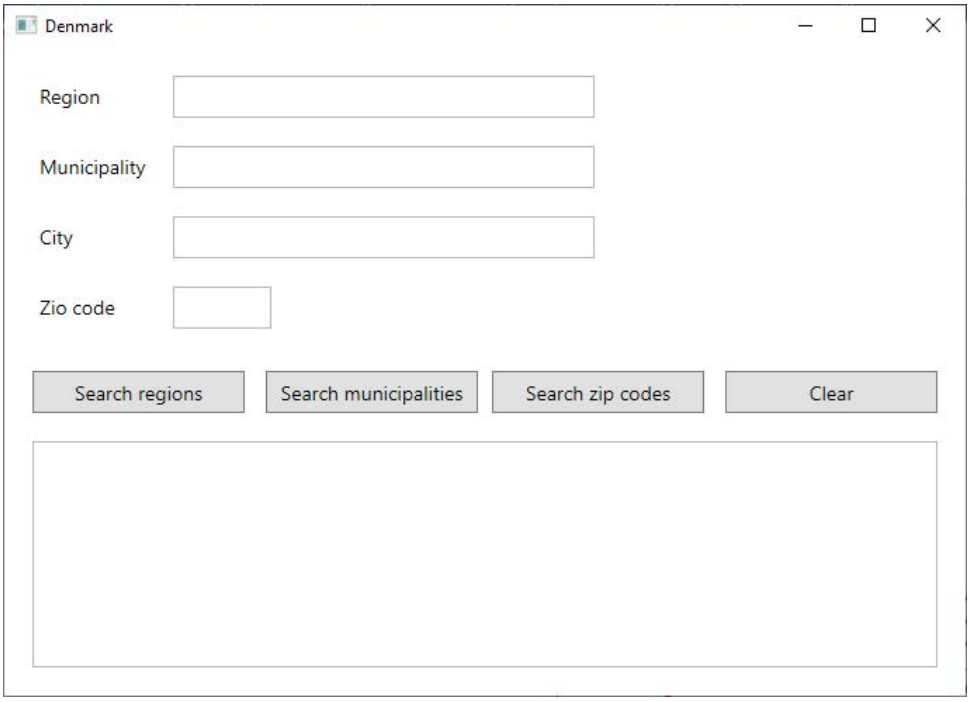
```
namespace Denmark
{
  [Serializable]
  public class Repository
  {
    private static String filename = "denmark.dat";
    private static readonly Repository instance = new Repository();
    private List<IRegion> regions;
    private List<IMunicipality> municipalities;
    private List<IZipcode> zipcodes;

    public static Repository Instance
    {
      get { return instance; }
    }

    /// <summary>
    /// Returns the zip code with zip code number code.
    /// If the code is not found the method should return null.
    /// </summary>
    public IZipcode GetZipcode(string code) { ... }

    /// <summary>
    /// Returns the region with region number rnr.
    /// If the region is not found the method should return null.
    /// </summary>
    public IRegion GetRegion(int rnr) { ... }

    /// <summary>
    /// Returns the municipality with municipality number mnr.
    /// If the municipality is not found the method should return null.
    /// </summary>
    public IMunicipality GetMunicipality(int mnr) { ... }

    /// <summary>
    /// Returns an iterator for zip codes
    /// </summary>
    public IEnumerator<IZipcode> GetZipcodes() { ... }

    /// <summary>
    /// Returns an iterator for regions
    /// </summary>
```

```
    public IEnumerator<IRegion> GetRegions() { ... }

    /// <summary>
    /// Returns an iterator for municipalities
    /// </summary>
    public IEnumerator<IMunicipality> GetMunicipalities() { ... }

    // Private constructor used to create a static instance for this
    // singleton.
    // The constructor try to deserialize an instance of a Repository
    // object. If it is not possibile the constructor reads the content
    // of the three text files
    // regions
    // zipcodes
    // municipalities
    // and creates the three lists with objects from these files. When
    // the objects are created and the lists initialized, the constructor
    // serialize the instance.
    private Repository() { ... }
  }
}
```

The program should open the window below, where it should be possible to search for regions, municipalities and zip codes. You can enter the following search texts:

1. *Region* that matches all regions where the name contains the search text
2. *Municipality*, that matches all municipalities where the name contains the search text
3. *City* that matches all zip codes where the city starts with the search text
4. *Zip code* that matches all zip codes where the code starts with the search text

It is a part of the task to decide what it will mean if a search is combined of several criteria, but if, for example you searches municipalities and typed something for municipality and city, it might for example mean that you should see the names of all the municipalities where the municipality's name contains the search text for municipality, and the municipality uses the zip code that matches the search text for city name.



5.5 A FILE FOR OBJECTS

As the last example in this chapter about files I will show a class representing a file to store objects. The example is a bit technical and although the class can be used it has hardly as much interest. The purpose of the example is primarily to show topics related to files not discussed above. The class is called *ObjectFile* and is implemented in a class library called *FileLib*.

A class to represent a file must have the following operations:

1. Open the file so it is ready for use
2. Close the file when it should no longer be used
3. Write an object to the file
4. Read a particular object from the file
5. Delete a particular object
6. Find objects with certain properties

To write such a class more decisions are needed, and the following is just one of several options. First, I want an object to have a key field and then a read-only property which can identify an object. This means that only objects of a type of this kind can be stored in the file and I will defined a file class as:

```
public class ObjectFile<K, T> : IEnumerable<T>
  where K : IComparable<K> where T : ObjectKey<K>
```

that is a generic class parameterized with two parameters where the first represents the key and must be a comparable type, while the other defines the objects to be stored in the file and must be a type that inherits the class

```
namespace FileLib.OF
{
  [Serializable]
  public class ObjectKey<K> where K : IComparable<K>
  {
    public K Key { get; private set; }

    public ObjectKey(K key)
    {
      Key = key;
    }

    public override bool Equals(object obj)
    {
      return Key.Equals(obj);
    }

    public override int GetHashCode()
    {
      return Key.GetHashCode();
    }
  }
}
```

It is a very simple class, and the purpose is alone to ensure that objects to be stored in the file has a read-only unique key which overrides *Equals()* with value semantics.

An object that is an object of the type *T*, can have any size as there are no other requirements, and the problem is where to store the object in the file so it can be found again based on the key, and without having to search the entire file. Here you should note that two objects of the same type can have very different sizes. To solve this problems I will use a file divided into blocks where a block has a fixed size. An object should then be stored in one or more blocks. When an object is deleted the blocks used for the object are added

to a free list which is a linked list of blocks that are unused and can be used to store new objects. When an object is added to the file, the file must allocate blocks for the object, and here the algorithm first uses blocks in the free list, and when it is empty blocks are allocated from the end of the file.

It solves the problem for allocated room for the object in the file, but to solve the other problem and find an object from the key the file has assigned a dictionary with keys and the object's position in the file as value. This dictionary can be interpreted as an index for the objects in the file.

This solution ensure that records of any size can be stored, and they do not use more blocks than needed and also that the blocks can be reused when an object is deleted. The file has no extern fragmentation, there are no unused areas between the blocks. In contrast, the file has internal fragmentation, since the last block for a file does not necessarily use all the space. Therefore, when using the file, you have a choice, since a small block size can mean that objects must have allocated many blocks that can result in bad performance, while a large block size can mean large internal fragmentation and then unused disk space.

Another problem is the dictionary when it also must be saved somewhere. Here I have chosen to save the dictionary using serialization as it is simple when the class *Dictionary* is *Serializable*.

A data block for an object can be defined as:

```
namespace FileLib.OF
{
  class Block
  {
    private byte[] bytes;

    public long Next
    {
      get { return BitConverter.ToInt64(bytes, 0); }
      set
      {
        byte[] arr = BitConverter.GetBytes(value);
        Array.Copy(arr, 0, bytes, 0, 8);
      }
    }
```

```
    public int First
    {
      get { return BitConverter.ToInt32(bytes, 8); }
      set
      {
        byte[] arr = BitConverter.GetBytes(value);
        Array.Copy(arr, 0, bytes, 8, 4);
      }
    }

    public byte[] Bytes
    {
      get { return bytes; }
    }

    public Block(int size)
    {
      bytes = new byte[size];
    }

    public void Fill(byte[] buff, int n)
    {
      for (int i = 12; i < bytes.Length && n < buff.Length; ++i, ++n)
        bytes[i] = buff[n];
    }

  public void Retrieve(byte[] buff, int n)
    {
      for (int i = 12; i < bytes.Length && n < buff.Length; ++i, ++n)
        buff[n] = bytes[i];
    }
  }
}
```

A block is defined as an array of bytes. An object is as mentioned stored as one or more blocks, and the first 8 bytes is used for the location of the next block in the chain of blocks for the current object. In the last block is value 0 indicating that it is the last block in the chain. The next 4 bytes are used for an *int* indicating the size of the object in bytes. Only the first block has a value in this field, while the field in the other blocks is 0. The object size could instead be saved in the dictionary and the field would be unnecessary, but when the field is included, it is due to a desire to implement an option to restore a file that is corrupted, but about that later. The rest of the block is used for data, and that means that the first 12 bytes of a block are used to chain the blocks. You should note how to convert from and to a byte array and numbers and here specifically the class *BitConverter*. Also note the two last methods that are used to initialize the data part of this block from a byte buffer and how to retrieve the data bytes in this block to a byte buffer.

The data file consists of blocks for the objects stored in the file, but the file starts with 20 bytes used for file info:

1. Start of the free list and then the location of the first block in the free list that is the first unused block.
2. The end of the file and then the first location after the files.
3. The block size used to check the size when an *ObjectFile* object is created.

```
namespace FileLib.OF
{
  class Header
  {
    public long Free { get; set; }
    public long End { get; set; }
    public int Size { get; set; }
  }
}
```

The class *ObjectFile* has the following variables:

```
private SortedDictionary<K, long> entries = null;
private string filename1 = null;
private string filename2 = null;
private FileStream data = null;
private BinaryWriter writer;
private BinaryReader reader;
private Header header;
private bool flush;
private bool copy;

public string Filename { get; private set; }
public int BlockSize { get; private set; }
public bool IsOpen { get { return entries != null; } }
```

The first is for the dictionary. The two next for the filenames, where the first denotes the index and then the file where to serialize the dictionary while the other is for the data file. Next follows a *FileStream* for the data file as well as a *writer* and a *reader* for this file. Then here is a variable to the file header. The last two variables is used to define where to flush the file after each write operation and where to make a copy of the file before it is opened. The three properties should be self-explanatory.

The code fills some and I do not want to show it all but mainly focus on where there is something new.

The class has one constructor:

```
public ObjectFile(string filename, int size = 1024)
```

The first parameter is the name of the directory that should contain the two files, while the other is the block size. If the file already exists the constructor test the block size, and if the size is illegal the constructor throws an exception. If the file does not exists the constructor creates the directory.

The method

```
public void Open(bool flush = true, bool copy = false)
```

must open the file and it happens different depending on whether the file exists or not. If the file exists the method must deserialize the dictionary, open the data file and read the header. Else it must create the dictionary, create a header object and initialize it and create the data file. In both cases the method must create a reader and a writer. If the file exists and the parameter *copy* is *true* the method should make a backup, which means the method must copy the two files to a subdirectory called *backup*. The backup is used if one of the methods *Close()*, *Write()* or *Delete()* fails. If so there could be inconsistency between the data file and the dictionary in the index file, and if there is a backup it is restored. You must note that this solution is somewhat destructive as it means that all changed performed since the file was opened are lost.

The method to take a backup is:

```
private void Backup()
{
  try
  {
    string name = Filename + "\\backup";
    if (!Directory.Exists(name)) Directory.CreateDirectory(name);
    string name1 = Filename + "\\backup\\OF.idx";
    string name2 = Filename + "\\backup\\OF.dat";
    if (File.Exists(name1)) File.Delete(name1);
    if (File.Exists(name2)) File.Delete(name2);
    File.Copy(filename1, name1);
    File.Copy(filename2, name2);
  }
  catch
  {
    copy = false;
    throw new OFErrorException(1);
  }
}
```

When you see the code it is easy enough to understand what happens but you should note the classes *Directory* and *File* which are classes with static methods to manipulate the filesystem, that is methods to create directories and files, delete directories and files and so on.

The method *Close()* flushes the file which means that the dictionary must be serialized, the header must be written to the data file and the data file must be closed. Also all variables are initialized to their default values and the dictionary is set to *null* which indicates that the file is not open. If the file cannot be closed properly and the variable *copy* is *true* the file is restored from the backup.

The method

```
public void Write(T obj)
```

writes an object to the file. This operation requires two things

1. converting the object to a byte array
2. allocates blocks to the object

To convert an object to a byte array the following method is used:

```
private static byte[] ToBytes(T obj)
{
  byte[] arr = null;
  using (MemoryStream ms = new MemoryStream())
  {
    BinaryFormatter br = new BinaryFormatter();
    br.Serialize(ms, obj);
    arr = ms.ToArray();
  }
  return arr;
}
```

The method serialize an object, not to file but instead to a *MemoryStream* that is a representation of a stream in memory. A *MemoryStream* has a method *ToArray()* which returns the contents as a *byte* array and the result is the object *obj* serialized to a *byte* array. This *byte* array must be written to the data file. If the object is already in the file it is first deleted and then it is written to the file, which means to allocate blocks ether from the free list or from the end of the file. After the object is stored in the file also the dictionary must be updated.

If the variable *flush* is *true* which is default the file is flushed. The implementation of a file as in this example is a bit vulnerable as in cases where the program fails you risk that the data file and the dictionaries are no longer synchronized and if not the file cannot be used. You can help with the problem by flushing the file after each update, and in the case of a few updates it is also not a problem, but for many updates it is as it takes a long time to serialize the index. Therefore, to flush after each write is an option that you can specify when the file is opened. If there is many updates it is better to use the other option for the method *Open()* and creates a backup.

The next method is *Read()* but I will not show the code here. The method has a key as parameter and can in the dictionary find the location of the first block in the data file. The method must then read all blocks to at byte buffer and deserialize the content of the buffer to an object:

```
private static T ToObject(byte[] param)
{
  T obj = null;
  using (MemoryStream ms = new MemoryStream(param))
  {
    BinaryFormatter br = new BinaryFormatter();
    obj = (T)br.Deserialize(ms);
  }
  return obj;
}
```

The class has also a method

```
public List<T> Find(params ObjProperty[] fields)
{
  if (!IsOpen) throw new OFWarningException(5);
  List<T> list = new List<T>();
  foreach (K key in entries.Keys)
    try
    {
      T obj = ReadObject(key);
      bool ok = true;
      for (int i = 0; i < fields.Length && ok; ++i)
      {
        object value = GetPropertyValue(obj, fields[i].Name);
        ok = value != null && value.Equals(fields[i].Value);
      }
      if (ok) list.Add(obj);
    }
    catch
    {
    }
  return list;
}
```

used to return a list with all objects which satisfies some search criteria. The search criteria are defined as key / value pairs where a key is the name of a property and value is a value for that property:

```
namespace FileLib.OF
{
  public class ObjProperty
  {
    public string Name { get; private set; }

    public object Value { get; private set; }

    public ObjProperty(string name, object value)
    {
      Name = name;
      Value = value;
    }
  }
}
```

You should note how the method *Find()* use reflection to search the file, and it is only possibility as the method does know the type *T*.

The class has many details and it is a good practice to study the finished code. The class can certainly be used in practice if many object updates are needed, but if it is the case, you will usually use a database, what is the topic of the next book. Finally, the class has the problem that it is vulnerable if an error occurs and for that the class having built in a backup / restore strategy or by flushing the file after updating.

Either way, the class is vulnerable and if there is a lack of integrity between the data file and index, the file cannot be used. Therefore, the class has a static method *Repair()* which, based on the data file alone, tries as a backup to create a copy of the data file with a new index. It is intended as a method that can be used if something goes wrong. I will not show the method here.

As the last thing I will mentioned the implementation of the iterator pattern:

```
public IEnumerator<T> GetEnumerator()
{
  if (!IsOpen) throw new OFWarningException(5);
  foreach (K key in entries.Keys) yield return ReadObject(key);
}
```

and here the statement *yield return* which has nothing to do with files or this example. When you use *yield* in a statement, you indicate that the method, operator, or `get` accessor in which it appears is an iterator. This means that you do not need for an explicit extra class (the class that holds the state for an enumeration) when you implement the iterator pattern for a custom type. The compiler will automatic generate the required code.

EXERCISE 7: TEST OBJECTFILE

In this exercise you must test the class *ObjectFile* from the above section, and that is you should try to use the class. Create a new console application project that you can call *FileProgram*. Add a reference for the class library *FileLib*.

Add the following class to the project:

```
using System;
using FileLib.OF;

namespace FileProgram
{
  [Serializable]
  public class Person : ObjectKey<string>
  {
    public string Firstname { get; set; }
    public string Lastname { get; set; }
    public string Address { get; set; }
    public int Zipcode { get; set; }
    public string Email { get; set; }
    public string Title { get; set; }

    public Person(string phone, string firstname, string lastname, string
    address,
      int zipcode, string email, string title) : base(phone.ToUpper())
    {
      Firstname = firstname;
      Lastname = lastname;
      Address = address;
      Zipcode = zipcode;
      Email = email;
      Title = title;
    }

    public override string ToString()
    {
    return Firstname + " " + Lastname;
    }
  }
}
```

The class is simple and represents a person as 7 properties, but you must note the class inherits the class *ObjectKey<string>*, is serializable and then is a class defining objects that can be stored in a file of the type *ObjectFile*. The keys has the type *string* and is interpreted as the phone number which is used as key.

You must then write a method *Test1()* which create a file and store an object in the file. The method must create an *ObjectFile<string, Person>*:

```
ObjectFile<string, Person> file =
  new ObjectFile<string, Person>("C:\\Temp\\Persons", 256);
```

You can use another filename as you like, but you should use the block size on 256 as it means that all *Person* objects needs two data blocks. You must then

1. Create a *Person* object, you determines the values for the 7 fields.
2. Open the file with the default parameters (the file will be created).
3. Write your *Person* object in the file.
4. Close the file.
5. Open the file again.
6. Read the stored *Person* object, remember to use the phone number as key.
7. Close the file.
8. Print the object read in the file.

If it all seems to work, and it should do, use Explorer to test that the files are created and check where the size of the data file has the expected size.

You must then write a method *Test2()*, that defines an *ObjectFile<string, Person>* object for the same file as above. The method must then

1. Open the file with the default parameters
2. Write 9 new *Person* objects to the file (with values as you decides but with different phone numbers)
3. Use the method *Find()* without parameters to select a list with all persons in the file
4. Use the method *FileSize()* to print the number of objects in the file
5. Close the file
6. Prints all persons in the list

When you have performed the method without errors use Explorer to test the size of the data file. Is it as expected?

In the next test method *Test3()* you should use the same file and perform the following operations:

1. Open the file, but without to flush after each operation and such the method creates a backup
2. Delete an object
3. Update an existing object, that is write an object to the file with an existing phone number
4. Delete two objects
5. Write a new object, a *Person* with a new phone number
6. Print all objects in the file but using the iterator and a *foreach* statement
7. Close the file

When it all seems to work use Explorer to test that the backup is created and check the size of the data file. Is it as expected?

The source code for this book contains a file called *Persons.txt*. It is a text file with 10000 lines where each lines is a semicolon separated line with data for a *Person* object. The data is not for real persons and is Danish names, but the values do not matter. You must write a method *Test4()* which

1. Read the text file *persons.txt* and create a list with 10000 *Person* object from the lines in the file
2. Creates an *ObjectFile* for the same file as above
3. Open the file as *Open(false)* and without let the *Write* operations flush the file
4. Read the clock
5. Write the 10000 *Person* objects to the file
6. Read the clock
7. Close the file
8. Print the number of milliseconds it has taken to write the 10000 objects to the file
9. Open the file again
10. Use the method *FileSize()* to print the number of objects in the file
11. Close the file

When you performs the method you should observe that the file is quite effective. On my machine it took approx. 300 milliseconds to write the 10000 persons to the file.

You should then write a method *Test5()* which do exactly the same as *Test4()*, but with one difference that the file is opened as *Open()* and without parameters. This time the method *Write()* does not add new objects to the file but updates the existing objects (with the same values). That doesn't mean much, but you will, in turn, observe the operation now taking a long time, and on my machine it took 3 minutes and some more. The difference is because this time a flush is performed as part of each write. This means opening and closing a file that is relatively slow operations, but also that it takes a long time to serialize a large object like a dictionary. In round numbers, each writing has taken 20 milliseconds. It may not sound like much, but if there are 10000 of those kinds of operations it does matter. You should then note that for a task with many write operations you should not open the file with flush.

As a last test method write a method *Test6()*:

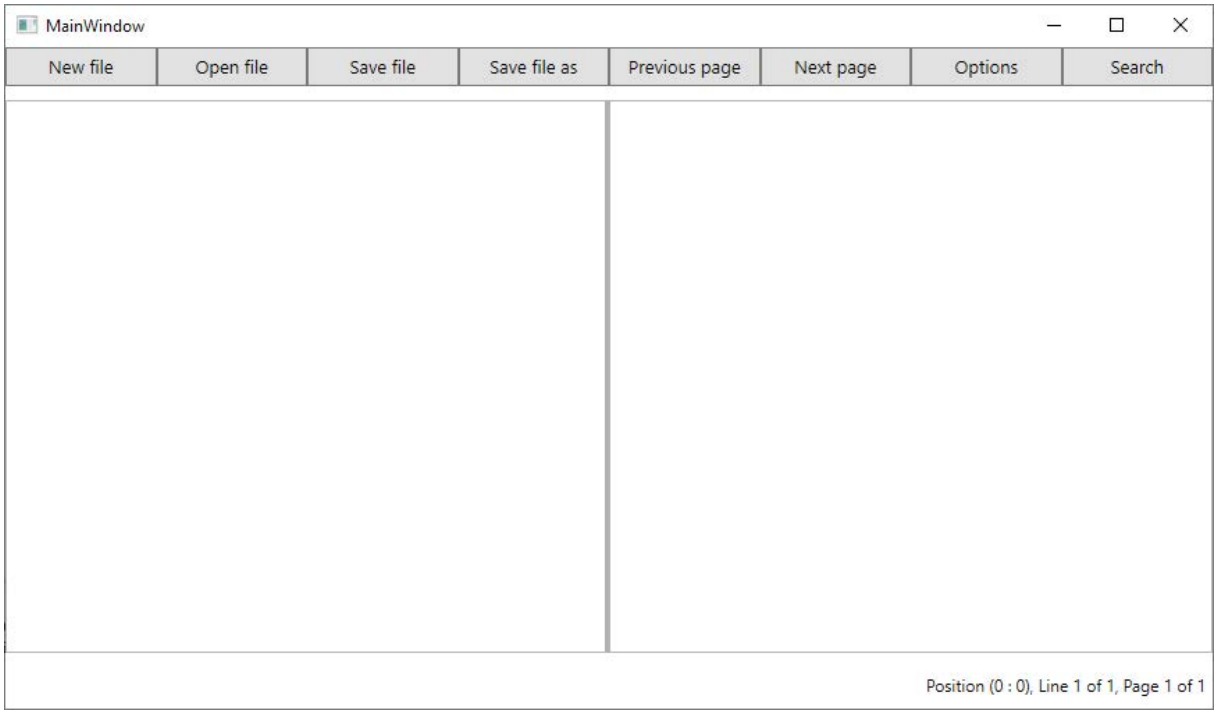
1. Read the above text file with the 10000 persons again and create a *List<string>* which contains all the phone numbers
2. Create an *ObjectFile* for the file again
3. Define a variable *sum* of the type *long*
4. Open the file
5. Read the clock
6. Iterate a loop 1000000 times where you in each iteration selects a random phone number from the above list, read the object in the file that has this phone number as key, add the zip code to the variable *sum* (just to do something)
7. Read the clock
8. Close the file
9. Print the number of milliseconds it has taken to read the 1000000 objects in the file

On my machine the operation took 23 seconds.

6 FINAL EXAMPLE: THE FILEBROWSER

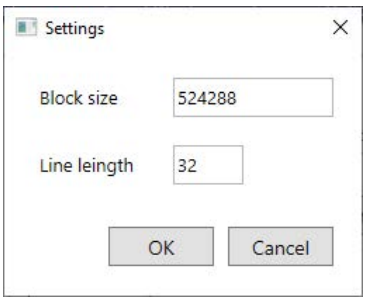
As a final example I will show a program that can open any file and edit the content - whether you have the right to do so. The program should display the content of a file in hexadecimal, and you can edit the file by modifying the individual bytes. The program can edit all files without exception (text files, images, translated programs, etc.), but another question is, whether you can get something out of it, and whether you can interpret the hexadecimal codes.

Contrary to what has been the case with the final examples in the previous books, which have primarily been focused on the process, I will in this example primarily look at the result and hence the program code. I will start with a presentation of the program to explain what the program is doing. When you open the program, you get the following window:

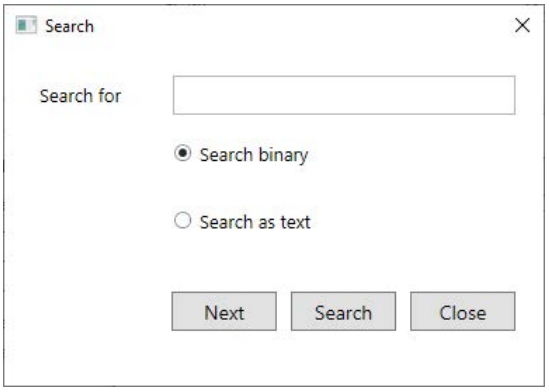


The window contains two *TextBox* components in a *GridSplitter*. The left component is the one that displays the content of the file in hexadecimal and where you can edit the file. The right component shows the content of the file interpreted as a text - to the extent that it is possible.

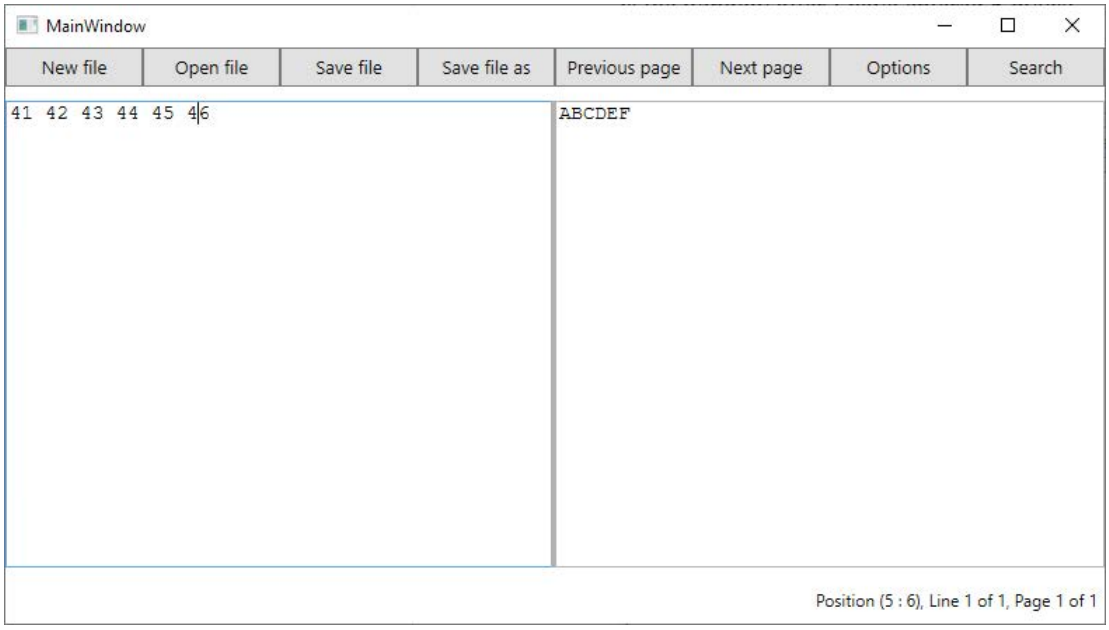
The function of the top eight buttons appears largely of the text where the first button clear the window and thus creates an empty file, while the three next are used to respectively open a file, save a file, and save a file as. Because files can be very large, the content of a *TextBox* can be so large that it is not manageable (the program will be slow), and therefore large files are split into blocks, and the program works with a single block at a time. The fifth and sixth button is used to scroll back and forth in these blocks. Finally opens the next last button, a dialog box where you can set the size of the blocks and how many bytes to show per line:



and the last button opens a search dialog:



After the program is opened there is not yet loaded any file, but you may well start entering data. Below is the window after I have entered 6 bytes:



Bytes are entered as their hexadecimal codes. Now you can not only enter codes, but you must insert a byte by pressing the *Insert* key, which inserts a byte at the cursor position width the value 00. Then you can change the value by typing

```
0, 1, 2, ..., 9, A, B, ..., F
```

The program inserts a space between the individual bytes by itself, and it is only for the sake of readability. If you save the above result in a file and examine the file, you will find that the file contains the text

ABCDEF

If you look at the program it is in principle a very simple program that alone contains a main window and two simple dialog boxes. There is also a model, which, incidentally, is not very complex, and the program consists then of three classes. The most complex is the class *MainWindow*, which contains many details.

When studying the code you should especially note

1. how the left component is a subclass of *TextBox*
2. how the program handles events related to the keyboard
3. how to scroll the two *TextBox* components as they are synchronized
4. how to read the content of any file and split into blocks

6.1 THE MODEL

I'll start with the model. The model must represent a data file, and in order to handle large files, that have been defined a simple class that represents a data block in a file:

```
public class Block
{
  private List<Byte> buffer = new List<byte>();
  private bool changed;

  public Block(byte[] buff, int size)
  {
    for (int i = 0; i < size; ++i) buffer.Add(buff[i]);
  }
```

The class is consisting of a byte buffer and a *bool* where *changed* defines where the content of the buffer is changed. The class is quite simple, and you should primarily note that the *set* part of the indexer sets the variable *changed* to *true*, and the class has a search method. The class is in the same file as the class *Model*.

The model class is the following:

```
public class Model
{
  private int lineLength = 32;      // number of bytes in a line
  private int blockSize = 524288;  // size og af block in bytes
  private string filename = null;   // reference to the file to be edit

  private List<Block> buffers = new List<Block>();
  private int current = 0;
```

By default the program displays 32 bytes in a line, and the block size is as default 524288 bytes (½ megabyte). This is a bit of a trade-off, where a small block size means effectiveness in the editing of the file's content, but in return many blocks to be edited one at a time. Increasing the block size also increases the chances that the whole file can be in a single block, but on the other hand, it could be slow to edit the file. These are the two values that can be changed in the *Options* dialog box. Then there is the variable *filename* that refers to the file been opened. Is the variable *null*, it means that there is not loaded a file.

The list is for blocks, and for small files (by default less than a ½ megabyte) there is only one block. Along with the list is the variable *current*, which indicates the index of the block that is currently been displayed in the editor.

The model has several properties and including properties for navigating the variable *current*, and there is especially more properties that return information about the model's state. They are generally simple, and I will not show these methods here.

The class also has methods to read and save a file. Below is shown the method, which opens a file:

```
public bool Open(string filename)
{
  this.filename = filename;
  buffers.Clear();
  FileStream file = null;
  try
  {
    file = new FileStream(filename, FileMode.Open, FileAccess.Read);
    while (true)
    {
      byte[] bytes = new byte[blockSize];
      int size = file.Read(bytes, 0, blockSize);
      buffers.Add(new Block(bytes, size));
      if (size < blockSize) break;
    }
    foreach (Block block in buffers) block.IsChanged = false;
    return true;
  }
  catch
  {
    Clear();
    return false;
  }
  finally
  {
    if (file != null) file.Close();
  }
}
```

The method is in principle simple, and the file is read as a *FileStream*. Input is performed in a loop, which is read a block at a time, and the result is that a large file is divided into several blocks.

Similarly the class has a method that save data in the file, and the following method save the blocks in a file:

```
public bool Save()
{
  if (filename == null) return false;
  FileStream file = null;
  try
  {
    file = new FileStream(filename, FileMode.Create, FileAccess.ReadWrite);
    long length = 0;
    foreach (Block block in buffers)
    {
      file.Write(block.GetBytes(), 0, block.Size);
      length += block.Size;
    }
    foreach (Block block in buffers) block.IsChanged = false;
    return true;
  }
  catch
  {
    return false;
  }
  finally
  {
    if (file != null) file.Close();
  }
}
```

and the method is easy enough to understand and in many ways is the opposite of the method *Open()*.

6.2 THE USER INTERFACE

This is essentially the class *MainWindow*, which is a relatively complex class. Since I do not want to allow the user to edit the content arbitrarily, but only must enter the hexadecimal values of the individual bytes, it is necessary to program the event handling for the keyboard and also the mouse, and it is in fact why the class is a bit complicated. You are encouraged to study the finished code thoroughly, but below I will briefly mention the most important.

Here are the two *TextBox* components that are for the views of the current file. The first *txtEdit* is used to edit the content as hexadecimal values, while the other *txtText* is used to display the content interpreted as text. At the bottom are two *Label* components used to show respectively, the file name (and size), and where in the file the cursor is.

The first *TextBox* component *txtEdit* has the type *HexEditor*, that it is a class that inherits *JTextArea*:

```
class HexEditor : TextBox
{
  private MainWindow view;
  private Model model;

  public HexEditor(MainWindow view, Model model)
  {
    this.view = view;
    this.model = model;
    FontFamily = new FontFamily("Courier New");
    FontSize = 14;
    VerticalScrollBarVisibility = ScrollBarVisibility.Auto;
    HorizontalScrollBarVisibility = ScrollBarVisibility.Auto;
    AcceptsReturn = true;
    PreviewKeyDown += KeyPressed;
    PreviewMouseDown += MouseHandler;
  }
```

The class's constructor adds handlers for events from the keyboard and for the mouse. The last does nothing but must place the cursor, if you clicks on the component. The handler must solve the problem that you must not place the cursor in front of a space or the end of a line. As for the keyboard, the program beyond the 16 keys to hexadecimal digits, the program supports the following keys:

- insert
- delete
- the 4 arrow keys
- home and ctrl home
- end and ctrl end
- pagedown and ctrl pagedoen
- pageup and ctrl pageup

wherein the two first modifies the content, while the other navigates the cursor.

The event handler for the keyboard is called

```
public void KeyPressed(object sender, KeyEventArgs e)
{
```

and mainly consists of a *switch*, where is switched on the keyboard codes to be treated. Whether it is a key to be treated or not the following statement is executed

```
e.Handled = true;
```

which means that the event is not passed on and thus not to the usual event handling in the base class *TextBox*. Most entries in the switch statement calls a method that performs the desired action. As an example is shown below the method called, if the insert key is entered:

```
private void InsertChar(int p)
{
  if (model.Block.GetBytes().Length == model.Block.Size + 1) return;
  int n = p / 3;
  if (p % 3 > 0) ++n;
  model.Block.Insert(n);
  view.ShowFile(p);
  view.ShowPosition();
}
```

The parameter is the cursor position. The first thing that happens is that the cursor position must be converted to the byte position in the model, where to insert a new byte, and here you must note that a byte because of the space fills three characters in the component. Here are adopted, if the cursor is in front of a byte, a new byte must be inserted in front, and the cursor is inside the byte, a new byte must be inserted after. After the new byte (with value 0) is inserted the method *ShowFile()* (a method also used in other contexts) is called which is the method that shows the content of the file (actual the current block). Finally the method *ShowPosition()* is called, which updates the status bar. These methods are, in principle, simple and not shown here. I will instead show the method that is called when a byte has to be changed:

```
private void ChangeChar(char ch)
{
  try
  {
    int p = CaretIndex;
    string text = Text;
    Text = text.Substring(0, p) + ch + text.Substring(p + 1);
    byte b2 = (byte)(ch >= '0' && ch <= '9' ? ch - '0' : ch - 'A' + 10);
    int n = p / 3;
    model.Block[n] = p % 3 == 0 ?
      (byte)(((b2 << 4) | (model.Block[n] & 0x0f)) & 0x000000ff) :
      (byte)(((model.Block[n] & 0xf0) | b2) & 0x000000ff);
    CaretIndex = p;
    byte b = model.Block[n];
    char c = b >= 32 && b < 127 ? (char)b : '.';
    text = view.GetText().Text;
    n = n + n / model.LineLength;
    view.GetText().Text = text.Substring(0, n) + c + text.Substring(n + 1);
    view.GetText().CaretIndex = n;
  }
  catch (Exception ex)
  {
  }
}
```

The method's parameter is the character entered, and the first thing that happens is the component is updated with the character in the right place. Then also the model must be updated. First is determined the character's value as a value between 0 and 15 inclusive. Next, the cursor position again must be converted into the right byte position in the model, and then it is determined whether it is the first or the last of the two digits to be changed. You should note that this method does not call the *ShowFile(),* what it really should. The reason is performance and the result is that the right component that displays the content as text is not updated, and therefore do not necessarily show the right content. Maybe it should have been an option.

Back there is the dialog box, which is used to change the parameters of the block size and the number of bytes per line. There is not much to explain, but you should note, that if the dialog box is used, the result is a blank file, and there is not loaded any file. It's a somewhat easy solution, but the reason is to change the block size, it is necessary to create new blocks.

The program also has another dialog that can be used to search for a particular pattern in a file. It is a modeless dialog so you can continue the search for the next match. The dialog box has an option where you can choose whether the search text should be interpreted as text or binary. If you choose binary, the search text is interpreted as a hexadecimal pattern, and this is the pattern that searches for. Otherwise, the search text is converted to an array of binary codes.

APPENDIX A: BINARY NUMBERS

If you technically have to deal with computers, you cannot ignore the binary numbers, as there are a lot of details that you cannot explain without having knowledge of the numbers and their binary representation. The following is a brief introduction to the binary numbers and including the hexadecimal system in view of applications in computer technology, but it is by no means an accurate treatment of number systems in general.

The numbers presented to us at the school are decimal numbers in which numbers are represented by 10 symbols, for example

```
18207
```

When we meet the number, we know exactly what it means, because we know the 10 symbols and their interpretation in relation to where a symbol appears in the number. The decimal number system is a positional number system as the value of a digit is determined by its position:

18207 = 1 tens of thousands plus 8 thousands plus 2 hundreds plus 0 tens plus 7 ones

When we taught to work with these numbers, we find them simple and easy to understand, but the reason is, that people very easy and very safe can survey and learn the 10 symbols.

The decimal system or 10-number system is not the only number system that people have historically used. For example, was previously used a 60-number system, which you can see the remains of in our division of the time: 1 hour divided into 60 minutes, 1 minute divided into 60 seconds. The 60-number system has also been used in commercially trade calculation. The system has several advantages. 60 has many divisors, which means that you can formulate many rules of arithmetic, and even quite large numbers does not take up so much, but there is however also a very big disadvantage, namely that the system requires 60 symbols (there are actually historical writings that shows the 60 symbols). That the 10-number system that has become the preferred, it is believed that it is because we have 10 fingers. Actually, there are also examples of how people have used a 20-number system (we also have 10 toes).

If you look at integers in the 10-number system, they can be written as $t = \sum_{i=0}^{n} a_i \, 10^i$ where the number has $n + 1$ digits, and $a_0, a_1, \ldots, a_n$ all is one of the symbols 0, 1, 2, ..., 9. As an example is

$$18207 = 7 * 10^0 + 0 * 10^1 + 2 * 10^2 + 8 * 10^3 + 1 * 10^4$$

That is $n = 4$ and $a_0 = 7, a_1 = 0, 2_2 = 2, a_3 = 8$ and $a_4 = 1$.

We say therefore that the decimal system is a positional number system with base 10. This presentation of the numbers can immediately be generalized to other bases, and actually there is a number system for any natural number greater than 1. As an example the 60-number system mentioned above is a system where the base number is 60.

THE BINARY NUMBER SYSTEM

The binary number system is simply a positional number system with base 2. Thus, it is only necessary with two symbols, and here are used 0 and 1. A number could, for instance be

$$t = \sum_{i=0}^{n} a_i 2^i$$

where $a_0, a_1, \ldots, a_n$ all are 0 or 1. If $n = 7$ and

$$a_0 = 1, a_1 = 0, a_2 = 0, a_3 = 0, a_4 = 1, a_5 = 1, a_6 = 0, a_7 = 1$$

is

$$t = 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0$$

This number is also written as $t = 10110001_2$ where the last number 2 tells that it is a number from the 2-number system. As you can see, the 2-number system is in principle complete equivalent to the 10-number system, where a number is presented as a sequence of symbols - 10 symbols in the 10-number system and 2 symbols in the 2-number system - and the numbers value is determined by the symbols position. The difference is that a symbol is interpreted as a power of 2 instead of a power of 10, and as such is the binary system simpler, since a given power of 2 either is included in the number (the symbol is 1) or it is not (the symbol is 0). If you look at the above construction, it is easy to determine the value of a number in the 2-number system by a simple calculation of powers of 2:

$$\begin{aligned} t &= 1 * 2^7 + 0 * 2^6 + 1 * 2^5 + 1 * 2^4 + 0 * 2^3 + 0 * 2^2 + 0 * 2^1 + 1 * 2^0 \\ &= 128 + 32 + 16 + 1 = 177 \end{aligned}$$

To compare the two systems, please note the following:

$0_2 = 0_{10}$	$100_2 = 4_{10}$	$1000_2 = 8_{10}$	$1100_2 = 12_{10}$
$1_2 = 1_{10}$	$101_2 = 5_{10}$	$1001_2 = 9_{10}$	$1101_2 = 13_{10}$
$10_2 = 2_{10}$	$110_2 = 6_{10}$	$1010_2 = 10_{10}$	$1110_2 = 14_{10}$
$11_2 = 3_{10}$	$111_2 = 7_{10}$	$1011_2 = 11_{10}$	$1111_2 = 15_{10}$

You can thereof immediately see that the numbers in the binary system takes up more space than numbers in the decimal system.

Below are some examples of positive integers written in the 2-number system, and how to convert those numbers to the 10-number system:

$1110000111110 = 2^{12} + 2^{11} + 2^{10} + 2^5 + 2^4 + 2^3 + 2^3 + 2 = 4096 + 2048 + 1024 + 32 + 16 + 8 + 4 + 2 = 7230$

$$1111111111 = 2^9 + 2^8 + 2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 1 = 1023$$

$$1110000001 = 512 + 256 + 128 + 1 = 897$$

$$10000000000 = 2^{10} = 1024$$

To determine the value of a binary number - or in other words to convert it to the decimal system - is simple and is just a question to remember the powers of 2. The other way to convert a number in the 10-number system to a binary number requires a little more.

Given a number, for example 2423, one can determine the largest power of 2 that is less than or equal to the number. It is $2^{11} = 2048$ and

$$2423 = 2^{11} + 375$$

The largest power of 2 which is less than or equal to 375 is $2^8 = 256$. That is

$$2423 = 2^{11} + 375 = 2^{11} + 2^8 + 119$$

The largest power of 2 which is less than or equal to 119 is $2^6 = 64$ and

$$2423 = 2^{11} + 375 = 2^{11} + 2^8 + 119 = 2^{11} + 2^8 + 2^6 + 55$$

The largest power of 2 which is less than or equal to 55 is $2^5 = 32$ and

$$2423 = 2^{11} + 375 = 2^{11} + 2^8 + 119 = 2^{11} + 2^8 + 2^6 + 55 = 2^{11} + 2^8 + 2^6 + 2^5 + 23$$

To continue that way, and you will find:

$$2423 = 2^{11} + 2^8 + 2^6 + 2^5 + 2^4 + 2^2 + 2 + 1 = 100101110111_2$$

That is, to convert a decimal number to a binary number you all the time subtracts the maximum power of 2 and continue until you get 0 or 1.

Below is another example:

$$265 = 2^8 + 9 = 2^8 + 2^3 + 1 = 100001001_2$$

The method is basically simple, but in practice and especially for large numbers, it can be difficult to determine the powers of 2 that you needs. Below is a little more direct way, where you constantly divide by 2.

You divide by 2 (equivalent to halving the number), and you continues with that until the result is 0. In each division there is a residue that is either 0 or 1. This residue is written after the colon. When finished - the quotient is 0 - the bits after the colon is the binary representation in reverse order - that is the most significant bit at the end.

```
265 : 100100001
132
  66
  33
  16
  8
  4
  2
  1
  0
```

$$265_{10} = 100001001_2$$

This method is really not so much simpler than the first, the division may be long, but the operations are simple and consists of determine the half of a number and where the remainder is 0 or 1.

```
2423 : 111011101001
1211
 605
 302
 151
  75
  37
  18
   9
   4
   2
   1
   0
```

$$2423_{10} = 100101110111_2$$

Note that it is easy to calculate whether the result is correct.

THE HEXADECIMAL SYSTEM

With reference to the above it is easy to define the hexadecimal system or 16-number system because it is a number system where the base or radix is 16, for example

$$327_{16} = 3 * 16^2 + 2 * 16 + 7 = 807_{10}$$

Again, it is just a question of the individual symbols positions means something different than in the decimal system.

However, there is one problem, since the 16-number system requires 16 symbols, and for this applies the 10 digits and the first 6 letters: A, B, C, D, E and F, and the symbols can then be interpreted as in the following table:

HEX	DEC	BIN	HEX	DEC	BIN	HEX	DEC	BIN	HEX	DEC	BIN
0	0	0000	4	4	0100	8	8	1000	C	12	1100
1	1	0001	5	5	0101	9	9	1001	D	13	1101
2	2	0010	6	6	0110	A	10	1010	E	14	1110
3	3	0011	7	7	0111	B	11	1011	F	15	1111

It is easy to convert a hexadecimal number to the decimal system, and is just a case of simple arithmetic. Below are some examples:

$$1A3E_{16} = 16^3 + 10 * 16^2 + 3 * 16 + 14 = 6718_{10}$$

$$B0032_{16} = 11 * 16^4 + 3 * 16 + 2 = 720946_{10}$$

$$2E_{16} = 2 * 16 + 14 = 46_{10}$$

$$ABCDEF_{16} = 10 * 16^5 + 11 * 16^4 + 12 * 16^3 + 13 * 16^2 + 14 * 16 + 15 = 11292531_{10}$$

The interesting thing about the 16-number system viewed from a computer is that 16 is a power of 2:

$$16 = 2^4$$

There are 16 symbols, and each of these symbols can be precisely expressed binary with 4 bits (see table above). This means that it is extremely simple to convert a hexadecimal number to a binary number, when one simply replaces each hexadecimal symbol with its corresponding binary representation as 4 bits:

$$A3B2_{16} = 1010001110110010_2$$

$$1234_{16} = 0001001000110100_2$$

when preceded by 0 bits has no effect on the value of the number.

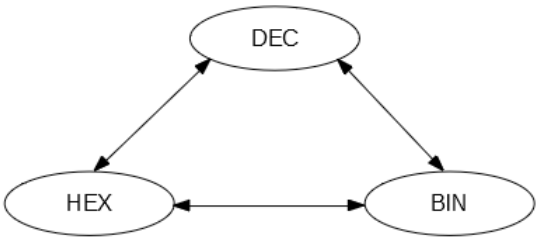
Converting the other way - from binary to hexadecimal - is similarly simple. The binary digits are arranged into groups of 4 bits, and each 4 bit group are converted to the corresponding hexadecimal symbol:

$$1001110101011110_2 = 1001\ 1101\ 0101\ 1110 = 9D5E_{16}$$

$$1110000101011_2 = 1\ 1100\ 0010\ 1011 = 1C2B_{16}$$

One can thus conclude that conversion between the binary number system and the hexadecimal system is extremely simple, and in fact it is the same, because the hexadecimal system is merely a shorter representation of the binary numbers. The binary number system is simple - at least seen from a machine, but for us humans, it is difficult to grasp and remember long sequences of 0 and 1. This is where the hexadecimal system comes into play, as it may represent sequences of 0 and 1 in a short way that is much easier for us humans both to read and remember.

Above, I mentioned three number systems and the conversion rules between these systems can be illustrated as follows:



Here you should note that I have not shown any method to convert from decimal to hexadecimal. However, one can immediately use the division method, which has been used when converting from decimal to binary, simply you must divide by 16 instead of 2. In practice it may be a bit difficult to divide by 16 - if it is done manually - and the method has only limited interest. Should you convert a decimal number to hexadecimal, it is easiest to convert the number to binary and from there to a hexadecimal number.

You often hears that a computer work binary, and what you mean by this is, that everything in a computer is represented as binary digits - or perhaps more accurately as sequences of 0 and 1. The reason for that is that it technically is simple to represent two states in a stable and simple manner, for example as two voltage levels, 0 volt may represents 1, while the 5 volts could mean 0. The important thing is that, technically it is easily separate the two voltage levels and in a safe way to determine whether a unit represents 0 or 1.

Internally in a computer the smallest unit that directly can be accessed is a byte, which is a bit pattern consisting of 8 bits. Exactly is the machine's RAM memory a large table of 8 bit patterns. If you have to specify the exact content of such a memory cell in the machine's memory, you specify 8 bits, for example

```
01000001
```

How it is interpreted depends on the program that refers the storage cell, but interpreted it as a binary number, it is the number 65. This number could also be used as a code for the letter A, and a program could thus choose instead of interpreting the above bit pattern as an A.

A computer will always internally represent data as a bit pattern - not necessarily 8 bits, but often 16 or 32, and perhaps even more bits. These bit patterns are difficult to handle for us humans: They are hard to write, and they are hard to remember. This is where the hexadecimal numbers come in, that as the hexadecimal numbers in reality is just a short representation of binary numbers, so they are often chosen to show the binary values for us people in hexadecimal form. The above bit pattern may also be written as hexadecimal

41

As a further example. Consider the number

28319

If you converts the number to the binary you get:

```
28319 : 111110010111011
14159
 7079
 3539
 1769
  884
  442
  221
  110
   55
   27
   13
    6
    3
    1
    0
```

$$28319_{10} = 110111010011111_2$$

Such a binary number are difficult to remember and hard to convey to others while its hexadecimal representation is much simpler to work with:

$$110111010011111_2 = 110\ 1110\ 1001\ 1111 = 6E9F_{16}$$

THE INTEGERS

When introduced to the whole numbers in mathematics, you learn that the whole numbers is an infinite set. A computer is a machine and on a machine there is nothing that is infinite, and specifically it means that a computer can represent only a subset of the whole numbers. As mentioned above, the smallest directly addressable unit is a byte consisting of 8 bits. Since a bit can be either 0 or 1, a byte can represent $2^8 = 256$ different bit patterns, and if you only have 1 byte available, it offers 256 different numbers. It is too little, and therefore a machine groups several bytes in a word. If you group 2 bytes, we say that the unit has a word length of 2 bytes or 16 bits. Similarly, if you group the 4 bytes, you get a word length of 4 bytes, or 32 bits, and the machine is working with 32-bit integers.

Previously, it was common to use 16-bit integer words, in order to have 16 bits available to an integer. As long as you talk about non-negative integers, the last 15 bits are used for the number, while the first is always 0. The number is then represented as a binary number, and as an example is the number 219 represented as:

```
0000000011011011
```

(you can easily calculate that it is the number 219). As another example the number 2890 is represented as:

```
00000101101001010
```

This representation allows to represents the following non-negative integers:

```
0000000000000000      0
0000000000000001      1
0000000000000010      2
0000000000000011      3
0000000000000100      4
....
0000000011011011    219
....
0111111111111111    32767
```

That is that the largest positive integer is 32767. Note especially that it is the number $2^{15} - 1$, and generally is the largest number that can be represented by n bits the number $2^n - 1$. The representation of the non-negative integers on a machine is straightforward and the only thing that matters is the word length, which determines how large numbers to work with. Most machines today using a word length of 32 bits for integer, and it means that the largest positive integer is $2^{31} - 1 = 2147483647$ and thus well over two billions. As an example is the number 219 represented as

```
00000000000000000000000011011011
```

on a machine with 32 bit words. You can therefore see the advantages and disadvantages of a large word length. A large word length means the machine can work with big numbers, but should you most of the time only work on small numbers, a large word length result in wasted space, since a large part of the space are just used to 0 bits.

Sometimes you also uses double-word integers, where you has 64 bits available. The largest positive integer is then 9223372036854775807 whose binary representation is:

```
0111111111111111111111111111111111111111111111111111111111111111
```

And then to the negative numbers, as is represented by their 2 complement. For not having to write too long bit patterns, I will start with a word length of 8 bits, that is a byte. Note that the largest positive integer that can be represented in one byte is

```
01111111 = 127
```

Consider as an example the number 97, and in one byte it is represented as the binary number:

```
01100001
```

By a number's complement means the number obtained by inverting all bits, that is,

```
10011110
```

This operation is seen from a machine extremely simple and efficient to implement in hardware and in principle similar to change the voltage level.

A numbers 2 complement is 1 added to its complement and the 2 complement of 97 (= 01100001) is then:

```
10011110
       1
10011111
```

That is that the number -97 in an 8-bit unit is represented by the bit pattern

```
10011111
```

It may be noted that adding 1 to a binary number can also be implemented very efficiently in hardware, so the entire operation to determine a number's 2 complement is highly effective.

As another example, consider the number 28, which in an 8-bit unit is represented as

```
00011100
```

Then the representation of -28 is:

```
11100011
       1
11100100
```

This calculation may require a little comment about how to add two binary numbers. When you adds two decimal numbers, you typical writes the numbers above each other so that they are aligned with the rear digit. Then you adds that the digits from back:

1. if the sum of the two digits is less than the base 10, it is the result
2. else you subtract the base 10 from the sum, and the difference is then the result, while you get a carry on 1, to be included in the sum of the next two digits

This method can be applied directly to binary numbers (and in principle on any other number system), and the difference is only that the base number is now 2: The sum at a particular position is either 0 or 1, and is the sum greater than 1, there will be a carry to the next position. All options can be illustrated in the following table:

Bit 1	Bit 2	Carry before	Sum	Carry after
0	0	0	0	0
1	0	0	1	0
0	1	0	1	0
1	1	0	0	1
0	0	1	1	0
1	0	1	0	1
0	1	1	0	1
1	1	1	1	1

It is thus a simple matter manually adding binary numbers.

If you look at the number representation from a computer, it will look at the first bit, and from the value of this bit know if it is a negative or a non-negative number, and is the first bit 1, the current software instantly convert the number by taking the 2 complement. Actually it is based on the following very important observation that if one determines a number's 2 complement and then again its 2 complement (thus taking the 2 complement twice), then you come back to the original number. For example is the 2 complement of 28

```
11100100
```

and taking the 2 complement to it again, you get:

```
11100100
00011011
       1
00011100
```

that is the number 28.

Consider the number -1. As other negative numbers it is represented by its 2 complement:

```
00000001
11111110
       1
11111111
```

That is where all bits is 1.

Consider the bit pattern

```
10000000
```

that, if interpreted as a number of a computer will be interpreted as a negative number:

```
10000000
01111111
       1
10000000
```

and thus the number -128. It means that with a word length of 8 bits, the computer can work with numbers in the interval [-128; 127], which has exactly 256 different integers.

Note that the interval is asymmetric such there is a negative number more than there are positive numbers. The reason is that there must also be room for the number 0.

I will conclude the examination of the negative numbers with a few examples. Consider again number 219, which in a binary 16-bit device is represented as

```
0000000011011011
```

The representation of -219 can be determined as

```
1111111100100100
               1
1111111100100101
```

In a 16-bit device, -1 is represented as

```
0000000000000001
1111111111111110
               1
1111111111111111
```

again only by 1 bits. As the above does not depend on the word length, you may notice that -1 independent of the word length will always be represented as nothing but 1 bits.

Also note what happens if you calculates the 2 complement of 0:

```
0000000000000000
1111111111111111
                1
________________
0000000000000000
```

as just again results in 0, and as it also should be.

Consider a 16-bit device that contains the number 1000000000000000, that is where the first bit is 1, but the rest is 0. The machine will interpret this number as a negative number and determine its 2 complement:

```
1000000000000000
0111111111111111
                1
________________
1000000000000000
```

and thus interpret it as -32768. With a word length of 16 bits you can represents the integers in the range [-32768; 32767].

If the word length is 32 bits, the smallest (negative) number is

```
10000000000000000000000000000000
```

that is the number -2147483648, and then you get the interval [-2147483648; 2147483647]. With a 64-bit word length the interval is [-9223372036854775808; 9223372036854775807].

If you look at how a modern computer represents negative integers, it's not the only way to do it, and there have also been used other methods. We uses as mentioned the 2 complement because it is an operation that can be effectively implemented in hardware, but previously has been used only the complement which is even more effective since it simply just turns the bits. This provides another problem, however. Consider the number 0 in an 8-bit device:

```
00000000
```

If you calculates the number's complement you get

```
11111111
```

that on the machine would be interpreted as -0, a result, that according to the basic math is 0. This means that when you have two representations of the number 0, there is a problem, which has to be solved in the software, and then complicates the calculations. It is the reason that computers today anywhere using the 2 complement rather than only the complement.

COMPLEMENT ARITHMETIC

The goal of this section is to show how the computer uses the above number representation for calculations. All calculations are performed by the electronics in your computer's processor, and from manufacturers of processors it is about developing processors that are as effective as possible, but also want a competitive in price.

Addition

A processor must be able to perform multiple operations, but the main thing is to add two integers. This section will be based on a processor that can perform addition of two binary numbers, and let me begin by are few examples to illustrate what can be done by addition of two binary numbers. Basically, I will count on 8 bits. It is something of a simplification, but since it does not change anything for the basics, it means that I can illustrate it all with fewer bits.

First, recall that with 8 bits, only the interval [-128; 127] are provided.

Calculate 73 + 46

The processor must then add the two binary numbers 01001001 and 00101110:

```
00010000
01001001
00101110
01110111
```

where the top row of bits is the carry. It is easy to figure out that the result is the number 119, which is also the correct result.

Calculate 1 + 2

This is a simple calculation and the result must be 3:

```
00000000
00000001
00000010
00000011
```

Calculate 112 + (-43)

```
112 = 01110000
43 = 00101011
```

The 2 complement of 43:

```
00101011
11010100
       1
11010101
```

-43 = 11010101 and the processor must therefore add the numbers 01110000 and 11010101:

```
 11100000
 01110000
 11010101
 01000101
```

This means that you get a result that can quickly be converted to 69, which is the correct result.

Perhaps it is not entirely obvious, for if you manual do the above addition you will see that there is a carry the last time, that apparently just disappears - and actually it do, because there is no room for it. You can say that you drops a sign bit away because the result was positive.

So we can note that a processor may well add a positive and a negative number.

Calculate 43 + (-112)

```
112 = 01110000
43 = 00101011
```

The 2 complement to 112:

```
 01110000
 10001111
        1
 10010000
```

-112 = 1001000 and the processor must therefore add the numbers 1001000 and 00101011:

```
 00000000
 10010000
 00101011
 10111011
```

When the first bit is 1, the result is negative and you have to determine the 2 complement:

```
10111011
00000100
        1
--------
01000101
```

It is the number 69, and then the result is -69, which is also correct.

Calculate 73 + 60

```
73 = 01001001
60 = 00111100
```

The processor must calculate the sum of 01001001 and 00111100:

```
11110000
01001001
00111100
--------
10000101
```

The result is negative, then you have to determine the complement

```
10000101
01111010
        1
--------
01111011
```

Converted to decimal it will be the result -123, which is obviously wrong. What you see here is an example of an overflow: The sum of the two numbers cannot be in an 8-bit device and fall outside the range that can be represented with 8 bits. Note that, when the addition of the last two digits there was carry, which was discarded.

Usually makes a processor nothing in such a situation beyond that it delivers a wrong result. If it is desired to handle this situation, it is up to the software, testing for overflow. If the processor had to do something, it would be more complex, and it would cost in efficiency and it is also not so easy to determine what the processor where appropriate should do. It is not entirely true that the processor is not doing anything. After each addition sets the processor more status bits, and one of them tells, if there has been an overflow. There is also a status bit, which is the last carry and software can thus test whether there is a carry which is "thrown away".

It is worth emphasizing that the problem of overflow is not specifically associated with a word length of 8, but it can occur regardless of the word length used. It is only a question of adding two numbers that are sufficiently large.

32 bits addition

As a final example, I look at addition of two numbers, but this time with a word length of 32 bits, which better reflects what happens in a real processor. I will look at the calculation

2819316 + 32512819

Actually there is nothing new in relation to the above, except that I should write more bits.

First a conversion:

```
  2819316 = 2B04F4 = 00000000001010110000010011110100
32512819 = 1F01B33 = 00000001111100000001101100110011
```

Then the calculation:

```
00000011110000000011111111100000
00000000001010110000010011110100
00000001111100000001101100110011
00000010000110110010000000100111
```

And so converting the result:

```
00000010000110110010000000100111 = 0000|0010|0001|1011|0010|0000|0010|
0111
                                  = 021B2027
                                  = 35332135
```

and it is easy to calculate that the results is correct.

Subtraction

If we now have a machine (a processor) which adds, you also has a machine that can subtract. Should you subtract two numbers, simply adding the second number's 2 complement to the first, and really it's nothing more than to exploit one of the rules in mathematics:

$$a - b = a + (-b)$$

If we again have a word length of 8 bits, and if you have to calculate 83 - 23 the following happens when you have to calculate the sum 83 + (-23)

```
83 = 01010011
23 = 00010111
```

The complement of 23:

```
00010111
11101000
_______1
11101001
```

-23 = 11101001

```
10000110
01010011
11101001
00111100
```

The result is 00111100 = 60.

As another example: Calculate 2819316 - 32512819, when the word length is 32 bits:

```
  2819316 = 2B04F4 = 00000000001010110000010011110100
32512819 = 1F01B33 = 00000001111100000001101100110011
```

The complement to 32512819:

```
00000001111100000001101100110011
11111110000011111110010011001100
                                1
11111110000011111110010011001101
```

That is

```
-32512819 = 11111110000011111110010011001101
```

Calculation:

```
00000000000111100000100111111000
00000000001010110000010011110100
11111110000011111110010011001101
11111110001110101110100111000001
```

The complement of the result:

```
11111110001110101110100111000001
00000001110001010001011000111110
                                 1
--------------------------------
00000001110001010001011000111111
```

and the result:

```
00000001110001010001011000111111 = 0000|0001|1100|0101|0001|0110|0011|
1111
                                  = 01C5163F
                                  = 29693503
```

The result is as follows, and it is easy to calculate that it is correct:

```
-29693503
```

The arithmetic as outlined in this section is called *complement arithmetic*, and as you can see, it is sufficient to construct a processor that can add. Then at the same time a processor also can subtract. This is one of the reasons to implements the representation of negative numbers by their 2 complement.

Multiplication

If you in the decimal number system multiply a number by 10 (with the base), you must move the number one position to the left and then add a 0. The same is valid for the binary number system: If you have to multiply a number by 2, you must move the bits one position to the left and adding a 0 and such an operation is called a left shift, and it is an operation which is simple to implement in hardware. Consider again the number 219:

```
0000000011011011
```

If you perform a left shift on this number you will get:

```
0000000110110110
```

and a simple calculation shows that it is the number 438 corresponding to a left shift is multiplying by 2.

If you look at how to perform multiplication in the 10-number system, for example, 432 * 1322, so this is done by multiplying the 1322 first with 2, then by 3 and finally with 4, but such that you each time shifts 1322 a position left and finally adds to it all:

```
432 * 1322
022100
  2644
 39660
528800
571104
```

The same method can be applied to the binary numbers, but just simpler, because every time either multiply by 0 or 1: To multiply by 0 gives 0, and multiplying by 1 gives the number again.

Suppose you want to multiply 219 by 38, which have the binary values 11011011 and 100110. If the multiplication is carried out with a word length of 16 bits, it may be performed as follows:

```
100110 * 0000000011011011
0000010010000000
0011110110111000
0000000110110110
0000001101101100
0001101101100000
0010000010000010
```

where I have only included the sub-results which do not give 0 - there are three such sub-results to calculate the sum, as there are three 1 bits in the number 100110. The result is the number 8322.

It is somewhat difficult to perform this multiplication manually, as by adding several binary numbers you need to be more careful to note what you have in carry.

As another example, I calculate the product 752 * 41 in a 16-bit device:

```
752 = 1011110000
41 = 101001

101001 * 0000001011110000
0000100000000000
0011101100000000
0000001011110000
0001011110000000
0101111000000000
0111100001110000
```

that is the number 30832.

In practice, it is not so important to be able to multiply binary, but the important thing is that the multiplication can be performed only by the two operations, left shift and addition and, therefore, can be carried out by a processor which can shift and add.

Division

Division is performed in the decimal number system by subtraction. The division algorithm can also be transferred to the binary number system, and division can then be performed by a processor, which can add.

It is not so easy to divide manually because you have to keep track of whether you have to borrow, but one example.

Consider the number 13556 and assume that it must be divided by 38, and the use of a word length of 16 bits. Note first that the result is an integer, and a simple calculation says that the result should be 356:

```
38 | 13556 | 356
     114..
      215.
      190.
       256
       228
        28
```

The quotient is 356, and you get a remainder of 28, corresponding to the division does not go up.

If you attempts to copy the usual division algorithm for decimal numbers to the binary system, the method is simpler, since each iteration always results in a 0 time or a 1 time.

```
13556 = 11010011110100
38 = 100110
```

The division can now be done binary, as shown below. However, it is not quite easy to do manually:

1. if the number you divide with is greater than the number above, it goes a 0 time, and the upper number goes unchanged on to the next step
2. otherwise it goes a 1 time, and the number you divide by, must then be subtracted from the number above - here it may be necessary to “borrow”, which you should keep track of
3. the next bit is used (as the right bit)
4. continuing until there are no more bits

```
100110 | 0011010011110100 | 00101100100
         100110.........
          011010.........
          100110.........
           110100........
           100110........
            011101.......
            100110.......
             111011......
             100110......
              101011.....
              100110.....
               001011....
               100110....
                010110...
                100110...
                 101101..
                 100110..
                  001110.
                  100110.
                   011100
                   100110
                    11110
```

That is, the result is

```
0000000101100100
```

that is the number 356.

It is not particularly interesting to be able to calculate binary, but the section shows that using the complement arithmetic and a processor that can add it can perform the four arithmetical operations.

BINARY OPERATIONS

Above I have discussed binary arithmetic and explained why the addition is a fundamental operation for a processor. There are other important operations as processors must be able to perform all of which can be implemented efficiently and simple in hardware. This section lists these basic binary operations.

Another reason for looking at these operations is that they are supported by many programming languages, so the languages has operators that correspond to the binary operations. In many contexts, it is important directly to be able to manipulate the individual bits, and to these the binary operators are important.

Left shift

This operation I have already mentioned, but given a device with a particular word length the operation shifts all bits one position to the left and insert a 0 at the right end. For example in a 16 bits unit

```
0011101111100001
```

and after a left shift the value is:

```
0111011111000010
```

The bit that was left is gone, which is a part of the operation, but in a processor this bit is transferred to a status bit.

If the bit pattern that is shifted to left represents a number it corresponds to, that number is multiplied by 2. Note that this means that if a number is shifted n positions, it corresponds to, that the number has been multiplied by 2^n.

Right shift

A right shift is an operation that shifts all the bits in a word one position to the right and inserts a 0 in the first position, for example

```
0011101111100001
```

and after a right shift the value is

```
0001110111110000
```

If the bit pattern that is shifted, represents a number, it means that the number is divided by 2. If the number is shifted n positions, it corresponds to, that the number is divided by 2^n.

There is a variant of a right shift, known as an *arithmetic right shift*. The difference is that the bit that is inserted in first position is the sign bit. That is, if the first bit is 0, a 0 bit is inserted, and if the first bit is 1, a 1 bit is inserted. If, for example you has the bit pattern

```
0011101111100001
```

is the result of a right arithmetic shift

```
0001110111110000
```

and such the same as a right shift. If, however, the pattern

```
1011101111100001
```

is the result of a right arithmetic shift

```
1101110111110000
```

AND

It is an operation that basically works on two bits, and where the result is a 1 bit if both of the arguments are 1 and otherwise 0. The operation may be expressed in the following table:

Arg 1	Arg 2	AND
0	0	0
0	1	0
1	0	0
1	1	1

The operation corresponds to a conjunction of two statements.

If you have two words an AND of the two words, is a bit-wise AND. If, for example you has two 8-bit words:

```
01100010
11110000
```

you determines their AND as follows:

```
    01100010
    11110000
AND 01100000
```

That is, you get 1, where there are two 1 bits, and otherwise the 0.

OR

It is an operation that basically works on two bits, and the result is a bit, which is 0 if both arguments are 0 and otherwise is 1.

The operation may be expressed in the following table:

Arg 1	Arg 2	OR
0	0	0
0	1	1
1	0	1
1	1	1

The operation corresponds to a disjunction of two statements.

If you have two words an OR the two words it is a bit-wise OR. If, for example you has two 8-bit words:

```
01100010
11110000
```

you determines their OR as follows:

```
   01100010
   11110000
OR 11110010
```

That is, you get 0, where there are two 0 bits, and otherwise 1.

XOR

It is an operation that basically works on two bits, and where the result is a bit which is 1 if the two arguments are different and 0 if they are equal. The operation may be expressed in the following table:

Arg 1	Arg 2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

The operation corresponds to the opposite of a bi-implication of two statements. If you have two words an XOR of the two words is a bit-wise XOR. If, for example you has two 8-bit words:

```
01100010
11110000
```

you determines their XOR in the following manner:

```
    01100010
    11110000
XOR 10010010
```

NOT

NOT is acting on a single bit by simply turning the bit. The operation can be described in a table as follows:

Arg	NOT
0	1
1	0

The operation is similar to a mathematical negation. If you have a word a NOT of the word is a bit-wise NOT. If, for example you have an 8-bit word:

```
01100010
```

you determines it's NOT as follows:

```
    01100010
NOT 10011101
```

That is, that a NOT is the same as the complement of the word.

Examples

I will conclude this section with a few examples of how using the binary operations to manipulate the individual bits in a word. Remember in this context that the smallest unit that directly can be addressed on a computer is a byte, and you cannot directly address the individual bits.

Given a 16-bit word:

```
0011101111100001
```

Usually are the bit positions numbered from behind starting with 0, and so that the last bit has index 0, while the first bit has index 15. In this case I would like to set the bit number 3 to 1 - no matter what value it may already have. This can be done with the following expression:

```
0011101111100001 OR 0000000000001000
```

If in a certain position you OR a 0 to the first bit pattern the result in that position will be unchanged and be the same as the value of the first bit pattern - to OR a 0 do not change anything. If in a certain position you OR a 1 to the first bit pattern the result in that position will certainly be 1 regardless of the value of the first bit pattern may have had. The result of the foregoing is, therefore,

```
0011101111100001 OR 0000000000001000 = 0011101111101001
```

The example can immediately be generalized to other word lengths and thus show how to set a particular bit of 1.

A related task concerns how to set a particular bit to 0, no matter what value it may have.

Given a 16-bit word:

```
0011101111100001
```

then set bit 8 to 0:

```
0011101111100001 AND 1111111011111111
```

If you in a certain position AND a 1 to the first bit pattern the result in that position will be the same as the value of the first bit pattern - that is AND a 1 does not change anything. If you in a certain position AND a 0 to the first bit pattern the result in that position certainly will be 0 regardless of the value of the first bit pattern may have had. The result of the foregoing is, therefore,

```
0011101111100001 AND 1111111011111111 = 0011101011100001
```

The example can immediately be generalized to other word lengths and thus show how to set a particular bit to 0.

As a final example I want to show how to test if a particular bit is 0 or 1. Given a 16-bit word:

```
0011101111100001
```

So, I would like to test the value of bit 4. Consider the following expression:

```
(0011101111100001 SHIFT RIGHT 4) AND 0000000000000001
```

If the value of this expression is 0, then the bit 4 is also 0. Otherwise, bit 4 is 1. You could also test bit 4 more directly by looking at the value of the following expression:

```
0011101111100001 AND 0000000000001000
```

ENCODING OF CHARACTERS

Above I have shown how to represent integers on a computer that works exclusively binary - that is where all data are represented as sequences of 0 and 1 bits. I have also demonstrated how the computer by the use of complement arithmetic can perform the four arithmetical operations. A computer works with other data types than integers and as such a computer must be able to work with text. In this section I will show how to represent text using binary numbers.

There are several ways, but the principle is simple, as you for each letter, digit and other characters associates a numeric code, and the only thing that is necessary is to agree on an encoding table that for each character determines which code to be used. That is where the differences occurs, as historically are used multiple tables.

ASCII

I'll start with a table called ASCII (*American Standard Code for Information Interchange*) that may not be used directly with modern computers, but it is yet an important table to know. The table encodes characters as 1 byte numeric codes, and the table then has room for 256 characters. The table is usually divided into three parts

1. the codes 0 - 31 that defines various control characters
2. the codes 32 - 127 that defines standard characters from the English alphabet
3. the codes 128 - 255 that defines among other country-specific characters

Note that the first two parts totally consists of 128 codes, and thus can be represented by 7 bits, and the first ASCII tables (the first standard) covered only the first two parts and was thus a 7-bit encoding.

Below is the first part of the table where I partly have shown the codes (both in decimal, binary and hexadecimal) and partly symbolic character's name and a brief descriptive text. Note that these codes do not directly correspond to the characters on the keyboard but are control codes, many of which are defined for the purpose of data communication in which text has to be transmitted over one or another communication line.

DEC	BIN	HEX	Symbol	Text
0	00000000	00	NUL	Null char
1	00000001	01	SOH	Start of heading
2	00000010	02	STX	Start of text
3	00000011	03	ETX	End of text
4	00000100	04	EOT	End of transmission
5	00000101	05	ENQ	Enquiry
6	00000110	06	ACK	Acknowledgment
7	00000111	07	BEL	Bell
8	00001000	08	BS	Back space
9	00001001	09	HT	Horizontal tab
10	00001010	0A	LF	Line feed
11	00001011	0B	VT	Vertical tab
12	00001100	0C	FF	Form feed
13	00001101	0D	CR	Carrige return
14	00001110	0E	SO	Shift out / X-on
15	00001111	0F	SI	Shift In / X-off
16	00010000	10	DLE	Data link escape
17	00010001	11	DC1	Device control 1 (oft. XON)
18	00010010	12	DC2	Device control 2
19	00010011	13	DC3	Device control 3 (oft. XOFF)
20	00010100	14	DC4	Device control 4
21	00010101	15	NAK	Negative acknowledgement
22	00010110	16	SYN	Synchronous idle
23	00010111	17	ETB	End of transmit block
24	00011000	18	CAN	Cancel
25	00011001	19	EM	End of medium

26	00011010	1A	SUB	Substitute
27	00011011	1B	ESC	Escape
28	00011100	1C	FS	File separator
29	00011101	1D	GS	Group separator
30	00011110	1E	RS	Record separator
31	00011111	1F	US	Unit separator

Below I have listed the codes used in connection with plain text:

- LF, used for line break
- HT, used for the tabulator character
- FF, used for page break
- ESC, the ESC key
- BS, the backspace key

Below is the second part of the table, which defines codes for characters in the English alphabet:

DEC	BIN	HEX		DEC	BIN	HEX		DEC	BIN	HEX	
32	00100000	20		64	01000000	40	@	96	01100000	60	´
33	00100001	21	!	65	01000001	41	A	97	01100001	61	a
34	00100010	22	"	66	01000010	42	B	98	01100010	62	b
35	00100011	23	#	67	01000011	43	C	99	01100011	63	c
36	00100100	24	$	68	01000100	44	D	100	01100100	64	d
37	00100101	25	%	69	01000101	45	E	101	01100101	65	e
38	00100110	26	&	70	01000110	46	F	102	01100110	66	f
39	00100111	27	'	71	01000111	47	G	103	01100111	67	g
40	00101000	28	(	72	01001000	48	H	104	01101000	68	h
41	00101001	29	)	73	01001001	49	I	105	01101001	69	i
42	00101010	2A	*	74	01001010	4A	J	106	01101010	6A	j
43	00101011	2B	+	75	01001011	4B	K	107	01101011	6B	k
44	00101100	2C	,	76	01001100	4C	L	108	01101100	6C	l
45	00101101	2D	-	77	01001101	4D	M	109	01101101	6D	m
46	00101110	2E	.	78	01001110	4E	N	110	01101110	6E	n
47	00101111	2F	/	79	01001111	4F	O	111	01101111	6F	o
48	00110000	30	0	80	01010000	50	P	112	01110000	70	p
49	00110001	31	1	81	01010001	51	Q	113	01110001	71	q
50	00110010	32	2	82	01010010	52	R	114	01110010	72	r
51	00110011	33	3	83	01010011	53	S	115	01110011	73	s
52	00110100	34	4	84	01010100	54	T	116	01110100	74	t
53	00110101	35	5	85	01010101	55	U	117	01110101	75	u
54	00110110	36	6	86	01010110	56	V	118	01110110	76	v
55	00110111	37	7	87	01010111	57	W	119	01110111	77	w
56	00111000	38	8	88	01011000	58	X	120	01111000	78	x
57	00111001	39	9	89	01011001	59	Y	121	01111001	79	y
58	00111010	3A	:	90	01011010	5A	Z	122	01111010	7A	z
59	00111011	3B	;	91	01011011	5B	[	123	01111011	7B	{
60	00111100	3C	<	92	01011100	5C	\	124	01111100	7C	|
61	00111101	3D	=	93	01011101	5D	]	125	01111101	7D	}
62	00111110	3E	>	94	01011110	5E	^	126	01111110	7E	~
63	00111111	3F	?	95	01011111	5F	_	127	01111111	7F	

Note particularly the code 32, which is a space.

This encoding is not sufficient, as there are a number of other letters that are national. For example the Danish letters æ, ø and å, but there are also other characters found on a keyboard, for example ½, and may also include other special characters such as the Greek alphabet.

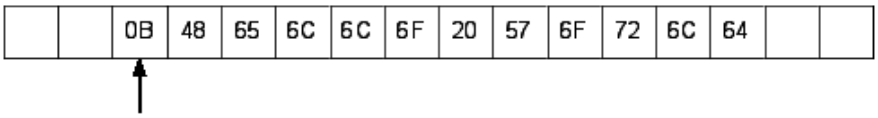
Encoding of these symbols have been solved by defining extended ASCII tables for codes 128-255 that defines additional 128 characters. A concrete machine can then load the table extension that you want to use. As an example is a code table, called ISO 8859-1, which includes the Danish letters, and below is shown the codes of the Danish letters in ISO 8859-1:

- æ 230
- ø 248
- å 229
- Æ 198
- Ø 216
- Å 197

Consider as an example the text

Hello World

where there are 11 characters. If this text is encoded as ASCII and stored in the machine's memory, it uses 11 bytes plus one, and the contents of the memory might be:



where the content is shown in hexadecimal (so it's easier for us humans to read), and where the arrow indicates the address where the text is stored. Note the first byte that indicates how long the text is. In one way or another there must be stored information about where the text ends, and it does not have to be a counter, as shown above, but it is one of the ways used. Note also that if you as illustrated above uses one byte to specify the length, then a text has a max length of 255 characters. The problem can of course easily be solved by using two bytes to or more to indicate the length.

It is important to note that there nowhere are said something about that the 12 bytes should be interpreted as text, and how those bytes are processed is completely determined of the program that reads the 12 bytes. As an example could a program choose to interpret the 12 bytes as three 32-bit integers, and the program will then read the following numbers:

```
0B48656C = 189293932
6C6F2057 = 1819222103
6F726C64 = 1869769828
```

which is something completely different than the text *Hello World*. A program can in principle read arbitrary bytes and interpret them as it will, but if it makes sense is a whole different matter.

Unicode

The encoding used today, is called *Unicode*, and the purpose is to replace the many different code pages known from ASCII, with a single code page, which contains all possible characters in all different languages. The price for it is obviously a very large table (just think of all non-European languages, chemical symbols, mathematical symbols, etc.), and it is also the system's disadvantage, both because it can be difficult to define a table, which all agree on, or that it could be "expensive" to use such a large table, if you really only need a small subset.

There is defined two different systems:

- UTF (Unicode Transfer Format)
- USC (Universal Character Set)

and each of these systems has multiple encoding, and I will mention the following:

- UTF-8, which is an 8-bit variable length encoding, which basically is a kind of extension of the ASCII system, and in particular is widely used on the Internet
- UTF-16 which is a 16 bit variable length encoding used by Windows
- UTF-32, which is a 32-bit fixed-length encoding, used to some extent in the Unix and Linux world (the same as the UCS-4)

Mentioned must also UCS-2, which is a fixed-length 16-bit encoding, that only has support a subset of all Unicodes.

That an encoding is of fixed length means that all characters take up the same. In UTF-32 all the characters fills 4 bytes. Is an encoding of variable length, the characters does not fill the same, and basically it is the principle that characters used often has codes with a smaller size, while characters used rarely take up more space. It is a choice. Fixed length encoding is the simplest but pay by memory consumption, while a variable length encoding minimizing the required to space, and pay with a more complex encoding, which places extra demands on the software.

UTF-16

I'll start with UTF-16 as the encoding used under Windows and also Linux. All codes are 2 or 4 bytes. The first 256 are ASCII codes that fully correspond to ISO 8859-1 and uses two bytes (16 bits) for each character. As an example is the encoding of a large A

```
0041
```

For all of the most frequently occurring characters 16 bits are used and they have thus hex codes from 0000 to FFFF, which gives 65,536 different codes. All other codes occupies 4 bytes or 32 bits. All the 16 bits codes is called *Basic Multilingual Plane* (BMP).

UTF-16 is an ISO standard that has the name ISO / IEC 10646 and include most of the world's characters. In principle, the 4 bytes to assign codes corresponds to 4,294,967,296 characters, but not all are used, and the encoding of 4 bytes code is relatively complex. Below is a general description.

For codes outside the BMP defines UTF-16 character codes in the range 10000 - 10FFFF that gives 1048576 codes. Each code is translated into two codes, called a surrogate pair. The procedure is as follows:

1. if the code, which according to the standard belongs to the interval [10,000; 10FFFF] subtracts 10000 from the code, and the result is certainly less than or equal to FFFFF, and thus fills a maximum of 20 bits
2. split the 20-bit code into 2 halves of 10 bits
3. adds D800 to the left half (as max is 3FF) that gives a value in the range [D800; DBFF]
4. adds DC00 to the right half (as max is 3FF) that gives a value in the range [DC00; DFFF]

It provides precise 1024 * 1024 = 1048576 pairs (10 bits provides 1024 different values).

The standard guarantees that there will never be defined surrogate pairs outside these ranges. Note that the two intervals are disjoint, and the reason for the somewhat complex encoding is that it must be simple to decode a four bytes Unicode.

UTF-8

UTF-8 is an encoding that is used much on the Internet as it directly is an expansion of ASCII coding, and as it strives to be sent as few bits as possible. It is a variable-length encoding, using from 1 to 6 bytes per character. In practice, however, a maximum of 4 bytes, as it is sufficient to encode the entire UTF-16 area.

The table below shows the principles for encoding the UTF-16 area, where a *b* indicates a data bit:

Unicode	UTF-8	Number of characters
000000 - 00007F	0bbbbbbb	128
000080 - 0007FF	110bbbbb 10bbbbbb	1920
000800 - 00FFFF	1110bbbb 10bbbbbb 10bbbbbb	63488
010000 - 1FFFFF	11110bbb 10bbbbbb 10bbbbbb 10bbbbbb	2031616

UTF-32

This encoding uses 32 bits for all the characters, and that means that text encoded by this standard takes up more memory. The system is used only to a limited extent in the Unix world. In principle, many text operations are simpler when all characters take up the same, but in practice the gains are modest.

REPRESENTATION OF DECIMAL NUMBERS

Above I have shown how to represent integers in a computer where positive integers are directly represented as binary numbers, and negative integers are represented by their 2 complement. A computer must also be able to work with decimal fractions, and it is once more difficult. There are a number of ways, but basically they can be divided into two categories

1. decimal numbers with fixed decimal point, which in many ways is an extension of the representation of integers, so that there is a number of the bits for integer part and a number of bits for the fractional part
2. decimal numbers with floating point, where you have a certain number of bits for the whole number, but the bits used for integer part and which are used for fractional part depends on the numbers current value

The two categories are quite different, and the first is by far the simplest, but is best suited if the numbers do not spread over too wide an interval. The second category enables a far greater range of numbers, but the representation is more complex.

I will only look at the last category. Partly because it is the most used form of representation of the decimal numbers, and partly because that's where most are to add in terms of what I have previously shown. One speaks generally about floating-point numbers, and here again there are several variants, but I would look at a standard called *IEEE Standard 754*, which is the most commonly used standard for representing floating-point numbers.

A bit more about binary numbers

In the introduction I mentioned that an integer in the 10-number system can be represented as a sum

$$\sum_{i=0}^{n} a_i \, 10^i$$

where the symbols $a_0, a_1, \ldots, a_n$ are digits in the decimal number system, and I also mentioned how this representation can be generalized to an arbitrary base and especially to the base 2 and the binary number system. Looking again at the 10-number system, the notation can be used to describe arbitrary decimal numbers that is numbers which include fractions:

$$\sum_{i=-m}^{n} a_i \, 10^i$$

If you looks at the number 1234.56, it can be written as

$$\sum_{i=-2}^{3} a_i \, 10^i$$

where $a_{-2} = 6, a_1 = 5, a_0 = 4, a_1 = 3, a_2 = 2, a_3 = 1$.

This notation in which you works with negative powers of the base number, may also be used for binary numbers. Consider the binary number

```
10110.10010111
```

which this time has a fraction. It is easy to determine the number's value as a decimal number, because it still just is a sum of powers of 2:

$$10110.10010111 = 2^4 + 2^2 + 2^1 + 2^{-1} + 2^{-4} + 2^{-6} + 2^{-7} + 2^{-8} = 22.58984375$$

However, you should note that if the result is calculated on a machine, the result will be rounded, if there are many bits after the decimal point, for example

$$1.00000000000011 = 1 + 2^{-13} + 2^{-14} = 1.0001831054688$$

You can also convert the decimal number to a binary fraction. This is done by converting the integer part and the fractional part separately. Consider the number 123.45

The integer part can be converted as shown previously:

```
123 =
64 + 59 =
64 + 32 + 27 =
64 + 32 + 16 + 11 =
64 + 32 + 16 + 8 + 3 =
64 + 32 + 16 + 8 + 2 + 1
```

or by division by 2. The result is that 123 = 1111011

The fraction part can be converted by multiplying by 2:

- multiply the fraction part by 2
- the integer part, that is 0 or 1, is the next bit
- repeat above until the fraction part is 0, or you have the wanted number of bits

This means that you finds the next bit of multiplying by two, and then continue with the fractional part to this result.

In this case, the method is:

```
  45
0 90
1 80
1 60
1 20
0 40
0 80
1 60
....
```

and the result is that 1111011.0111001 = 123.45 which is a rounded result.

As another example, consider the decimal number 0.0825. Here, it is only necessary to convert the fractional part, as the integer part is 0:

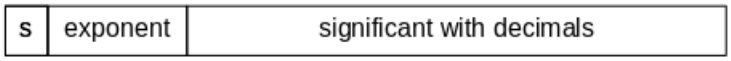
```
  0825
0 1650
0 3300
0 6600
1 3200
0 6400
1 2800
0 5600
1 1200
0 2400
0 4800
0 9600
1 9200
1 8400
1 6800
1 3600
```

The result: 0.0825 = 000101010001111

Note that if I converts the binary result to a decimal number, I get 0.0824890137 so you can see that there must be many bits to get an accurate result.

Floating point

The decimal numbers are a subset of the real numbers, and the problem is to represent real numbers using a finite number of bits, for example 32 or 64 which is the most common. In general, a real number is represented in the form



where the *s* field always fills 1 bit and represents the sign, the exponent field is an exponent and the significant field is used for number's digits.

The main problem is that the representation of a real number is not unique, but is always a rounded result. A representation of the integer defines a subset of the integers, such that all integers in a range is represented. Similarly it is, by the representation of the real numbers, that since there are only a limited number of bits available, we can represent only a subset of the real numbers, and thus the real numbers within a range, but unlike the integers includes any interval of real numbers infinite many numbers, and any representation of real numbers by using a specific number of bits can only represent a subset of the range. Another problem is that a decimal number such 123.45 cannot be converted to a binary number with a finite number of bits:

```
123.45 = 1111011.011100110011001100110011......
```

So there is no clear relation between decimal fractions and the binary numbers.

32 bits floating points

I'll start with a representation that takes up 4 bytes, that is 32 bits:

- 1 bit to the sign
- 8 bits for the exponent
- 23 bits for the significant

The sign bit is 0 if the number is non-negative and 1 if the number is negative. If you look at the significant field it contains the 23 bits:

```
bbbbbbbbbbbbbbbbbbbbbbb
```

and it is interpreted as the binary number

$$1.bbbbbbbbbbbbbbbbbbbbbbb * 2^{n-127}$$

where n is the value of the exponent field. 127 is called the *bias value*, which is binary 01111111. It is used to ensure that the content of the exponent field is not negative. I would as an example look at the number 123.45. Since it is a positive number is the sign bit 0. The number's binary representation is

$$123.45 = 1.11101101110011001100110011001100110011001100110011001\ldots * 2^6$$

Therefore, the content of the significant field is 11101101110011001100110 (the first 1 digit before the decimal point is implicit). Since 133-127 = 6 is the content of the exponent field 133 or binary 10000101. The number 123.45 is represented as the bit pattern

```
01000010111101101110011001100110
```

In the case of floating points are not used complement arithmetic, and a negative number is handled only by changing the sign bit. As an example is -123.45 is represented as

```
11000010111101101110011001100110
```

that is the same bit pattern as 123.45 except the sign bit.

To highlight the principle I will look at a small calculation. Suppose a floating point contains the following bits:

```
11000111001100111100111001010100
```

When the first bit is 1, it represents a negative number. The exponent part is 10001110, which is the number 142, and subtract the bias you gets 15.

The significant part is 01100111100111001010100 and the number is then

$$1.01100111100111001010100 * 2^{15} = 1011001111001110.01010100 = 46030.32813$$

and the value of the number is -46030.32813.

There is little variation in terms of how the boundary conditions are treated, but it can basically be assumed as follows. The exponent field represents values in the range of 0 to 255, but 255 which has the value 11111111, is illegal. The power of 2 exponent thus lies between -127 and 127. The largest positive number is:

```
01111111011111111111111111111111
```

The exponent is 11111110 - 127 = 254-127 = 127, and the greatest positive number is therefore

$$1.11111111111111111111111 * 2^{127} = 11111111111111111111111 * 2^{104}$$
$$= 3.402823466 * 10^{38}$$

Similarly, the smallest positive number:

```
00000000000000000000000000000001
```

The exponent is 00000000 - 127 = -127 and the number is

$$1.00000000000000000000001 * 2^{-127} = 10000000000000000000001 * 2^{-150}$$
$$= 5.87747245 * 10^{-39}$$

(it is one of the numbers that vary slightly). The result is that you can assume that the standard may represents numbers within the following range:

$$[-10^{38} ; -10^{-38}] \cup [10^{-38} ; 10^{38}]$$

and that when $2^{24} = 16777216$, there are about 7 significant digits.

Individual bit patterns are assigned a special meaning:

Bit pattern	*Value*
`00000000000000000000000000000000`	0
`01111111100000000000000000000000`	Infinity
`11111111100000000000000000000000`	Minus infinity
`01111111110000000000000000000000`	Not a number

but there are others.

64 bits floating points

The principle is the same as above, but using 64 bits which are interpreted as follows:

- 1 bit to the sign
- 11 bits for the exponent
- 52 bits for the significand

The bias value is 1023, that binary is 01111111111.

Consider as an example the number 12345.6789. Binary it is

```
11000000111001.10101101110011000110001111110001010000001.....
```

or

```
1.10000001110011010110111001100011000111111000101000001.....*2^13
```

Since 13 = 1036 - 1023 and 1036 = 10000001100 is the representation of 12345.6789:

```
0100000011001000000111001101011011100110001100011111100010100001
```

Slightly rounded it provides the opportunity to represent numbers within the following range:

$$[-10^{308}\,;\,-10^{-308}] \cup [10^{-308}\,;\,10^{308}]$$

and it means approximately 15 significant digits.

As a final calculation. Assume that a real number is represented as the following 64-bit pattern

```
0100000100010111000101101100100111001111110001111010001110011000
```

Sign:

0, the number is positive

Exponent:

```
10000010001 - 1023 = 1041 - 1023 = 18
```

Significant:

```
0111000101101100100111001111110001111010001110011000

1.0111000101101100100111001111110001111010001110011000 * 218 =
101110001011011001.0111001111110001111010001110011000 =
378290.45291
```