Poul Klausen

C# 3

Object Oriented Programming

Software Development

bookboon
LEARNING

POUL KLAUSEN

C# 3: OBJECT ORIENTED PROGRAMMING

SOFTWARE DEVELOPMENT

C# 3: Object oriented programming: Software Development
1st edition

ISBN 978-87-403-3464-7

CONTENTS

	Foreword	**6**
1	**Introduction**	**8**
1.1	The type object	8
2	**Struct**	**11**
	Exercise 1: A counter	16
2.1	Nullable struct	18
2.2	More on simple variables	20
	Exercise 2: Fibonacci numbers	28
3	**Enum**	**31**
4	**More on classes and interfaces**	**35**
4.1	Visibility	35
4.2	Design patterns	37
	Exercise 3: The Currency program	47
4.3	Interfaces	48
	Exercise 4: Programming to an interface	56
4.4	More students	57
	Exercise 5: A list of books	64
4.5	Factories	66
	Exercise 6: A factory	69
5	**Inheritance**	**70**
	Exercise 7: A loan calculator	84
	Problem 1: Number sequences	85
5.1	Considerations about inheritance	89
	Problem 2: Geometric objects	92
5.2	The composite pattern	98
6	**The class Object**	**100**
7	**Exception handling**	**105**
7.1	System Exception	114
7.2	The Exception class	115
	Exercise 8: Library exceptions	118
8	**Comments**	**120**

9	**More on C# syntax**	**126**
9.1	Automatic properties	126
9.2	The index operator	127
9.3	Object initializing	130
9.4	The var keyword	132
9.5	Expression bodied methods	133
9.6	More about parameters	134
9.7	Local methods	136
9.8	More on operators	137
9.9	Inner types	140
9.10	Partial classes and methods	142
9.11	Anonymous types	145
9.11	Anonymous methods	147
9.12	Tuples	149
9.13	Extension methods	151
10	**Recursion**	**153**
11	**A final example: A calculator**	**155**
11.1	Analysis	155
11.2	The prototype	159
11.3	Design	160
11.4	Programming, first iteration	171
11.5	Programming, second iteration	174
11.6	The last iteration	180

FOREWORD

This book is the third in a series of books on software development. In the first book I have generally mentioned classes and interfaces, and although the book C# 2 also intensive used classes and interfaces I have deferred the details to this book and also the next two, that are dealing with object-oriented programming. It deals with how a running program consists of cooperating objects and how these objects are defined and created on basis of the program's classes. Object-oriented programming is the knowledge of how to find and write good classes to a program, classes which helps to ensure that the result is a robust program that is easy to maintain. The subject of this book is object-oriented programming and here primarily about classes and how classes are used as the basic building blocks for developing a program. The book assumes a basic knowledge of C# corresponding to the book C# 1 in this series, but since some of the examples and exercises are relating to programs with a graphical user interface it is also assumed knowledge of the book C# 2 and how to write small GUI programs.

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#. You can learn that 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 treated in the books. All books in the series are 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 presented in the text, and furthermore it is relatively accurately described what to do. Problems in turn, are 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 is a program that you quickly become comfortable 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

C# has the following categories of types

1. struct
2. enum
3. delegate
4. class
5. interface

where the first two are value types while the last three are reference types. This book deals with these types, where the last two has been introduced in the first book, but there is more to say.

Here you should remember that a variable of a value type allocates space on the stack to its data, and one can say that the variable has its own copy of or contains the data stored in it. The variable is automatically destroyed (removed from the stack) when the program finishes the block where the variable is declared. A variable of a value type uses thus the number of bytes of the stack, which the type indicates. A variable of a reference type does not contain data directly, but contains instead a reference to data that is allocated on the heap. This means that one can have several variables that reference to the same data. This means also that a reference variable always has the same size on the stack that is the size of a reference (4 bytes). A reference can specifically have the value *null*, which means it does not refer to anything.

1.1 THE TYPE OBJECT

In the book C# 1 I mentioned briefly inheritance and classes can inherit each other, and I will tell more about that later in this book, but C# has a class *object* which all other types directly or indirectly inherits. When a type only can inherit one other type it means, that all types are linked together in a hierarchy with *object* as root. The type *object* is a class, and has the name *System.Object*, and it is then a class in the namespace *System*. When you in a program can write *object* of *Object* and it means the same it is because name *object* is a reserved work in the compiler, and the compiler knows that it means *System.Object*. Look as an example on the following program which is a simple console program:

```
using System;

namespace ObjectArray
{
  class Program
  {
    static void Main(string[] args)
    {
      object[] arr = { 123, DateTime.Now, Math.PI, "Gorm den Gamle",
      new Program() };
      foreach (Object obj in arr) Console.WriteLine(obj);
    }
  }
}
```

The program creates an array of the type *object* and the array is initialized with 5 objects. The types of these objects are

- *int*, which is a simple type and is as so a value type
- *DateTime*, which is a *struct* type (see the next chapter) and then a value type
- *double*, which is also a simple type, but a type that fills the double of an *int*
- *string*, which is a class type and then a reference type
- *ObjectArray.Program*, which is also a class type that is the type of the current program (a program is a class and you can then create an object of this type)

and it is legal as all types inherits the class *Object*. If you run the program you get the following result:

I think the first four lines are as you would expect, but the last line is probably not as you would expect, but you can see that the program runs without errors and also that the last *WriteLine()* prints something which is the full name of the object's class. But if you add the following method to the class

```
public override string ToString()
{
  return "Hello World";
}
```

and run the program again you will see that the program prints the text *Hello World* for the last object. The reason is that the class *Program* inherits the class *Object* which defines a method *ToString()*. This method is implemented that it prints the name of the class which defines the current object. A class can then override this method and give it its own meaningful meaning.

The class *object* defines other important methods that will be explained later in this book, but for now is it sufficient to know that all types in C# inherits the class *Object*.

2 STRUCT

In the book C# 1 I introduced class types, and in this chapter I will explains *struct* types which in many ways look like classes, but a class is a reference type while a struct is a value type, and objects of a struct type are then created on the stack. The reason for struct types are performance, as it is more effective to allocate objects on a stack than on the managed heap, but on the other hand, it is necessary for the compiler to calculate how much an object fills in memory.

Below is a *struct* which represents a point in a coordinate system and is a type in the project *Points*:

```
struct Point
{
  public double x;
  public double y;

  public Point(double x, double y)
  {
    this.x = x;
    this.y = y;
  }

  public double Length
  {
    get { return Math.Sqrt(x * x + y * y); }
  }

  public override string ToString()
  {
    return string.Format("({0},{1})", x, y);
  }
}
```

The type is called *Point*, and the type look like a class definition besides the word *struct* instead of *class*. There are two instance variables of the type *double*. This means that every time you create a *Point* there is allocated 16 bytes on the stack. The type have a constructor, which initializes the two coordinates, and there is a single property, which returns the distance from (0, 0) to the point. In addition there is a *ToString()* method. Actually, there is also an implicit default constructor that sets both variables to 0, and this constructor can't be overridden. Below is shown the *Main()* method that uses the type *Point*:

```
static void Main()
{
  Point p1;
  Point p2 = new Point(4, 5);
  p1.x = 2;
  p1.y = 3;
  Point p3 = p1;
  p3.y = 8;
  Show(p1);
  Show(p2);
  Show(p3);
  Console.WriteLine(p3.Length);
}

private static void Show(Point p)
{
  Console.WriteLine("({0},{1})", p.x, p.y);
}
```

If you run the program you get the result:

Main() creates a *Point*

```
Point p1;
```

That means creating a variable *p1* on the stack and the default constructor is executed. Note that it is not a reference, but a variable on the stack that takes up 16 bytes. Then *Main()* creates another variable *p2*, but this time using the *new* operator:

```
Point p2 = new Point(4, 5);
```

and even if *p2* is created with *new*, it is still created on the stack. The *new* operator is used here to get the constructor executed and therefore has a different meaning than it has for a class. As the next step values are assigned to the coordinates of *p1*. Note that it is possible, since both variables are *public*. Then the program create a third variable:

```
Point p3 = p1;
```

This means that the program create another variable on the stack, which is a copy of *p1*. Note that *p3* is a new variable, which has the same value as *p1*, but this is not a reference to *p1*. It can be seen by changing the value of *p3*, and that this change does not affect *p1*.

The syntax for a struct which is as for a class other than that the word *class* is replaced by the word *struct*. A *struct* can have constructors just as a class, properties and methods, and the only difference here is that a struct always have a default constructor, which can't be overwritten. A struct can also implement an interface, but a struct can't inherit. Aside from that there is no difference.

The difference between a struct and a class is thus the application. Typically, a struct is used to encapsulate a few simple data types to achieve a better performance equivalent to that it is a value type. Therefore you will often skip properties, making variables *public*, as is the case in the type *Point*. When an object is allocated on the stack, there can be only one reference to it, and the need for data encapsulation is not the same as for heap-allocated objects. You should be aware that a struct can very well have properties for instance variables with get and set methods and there can also be justifications for it if a change needs to be made before the value of a variable is updated. Finally, someone always chooses to make instance variables in a struct private and then define the required properties, for example, because they want the variables to be read-only or to be in accordance with good object oriented principles.

When a struct type is a value type, it means that if you assign a variable to another variable, it is a really copy

```
Point p1 = new Point(2, 3);
Point p2 = p1;
```

where *p1* is copied to *p2*. You must be special aware of that if a struct has instance variables of reference type, since it is only the references there are copied and not the objects which they refer. If you are not aware of that, it can sometimes produce unexpected results. As an example, consider the class *Cube* as I have used before:

```
class Cube
{
  private static Random rand = new Random();
  private int eyes;

  public Cube()
  {
    Roll();
  }

  public int Eyes
  {
    get { return eyes; }
  }

  public void Roll()
  {
    eyes = rand.Next(1, 7);
  }
}
```

There is nothing to note about it, besides it is a reference type. The following type defines a pair of cubes, a cup with two *Cube* objects:

```
struct Pair
{
  private Cube c1;
  private Cube c2;

  public void Toss()
  {
    if (c1 == null)
    {
      c1 = new Cube();
      c2 = new Cube();
    }
```

```
    else
    {
      c1.Roll();
      c2.Roll();
    }
  }

  public Cube First
  {
    get { return c1; }
  }

  public Cube Last
  {
    get { return c2; }
  }

  public override string ToString()
  {
    return c1.Eyes + " " + c2.Eyes;
  }
}
```

It is a value type with two instance variables which both are reference types. These means that the default constructor will assign the default value for these variables and it is *null*. When the type has no user defined constructor a new *Pair* will always have the value *null* for these variables. You must note, that in a definition of a struct you can not initialize variables in the declaration, and if you have to, it must happens in a user defined constructor.

When the variables are private I must define properties for the variables (not needed in this program), but I want the variables to be read-only. I gives a problem as a variable of the type *Pair* then can't create objects for *Cube* variables. The problem is solved when the method *Toss()* tests where the objects are created, and if not the method creates the objects. Note that a better solution would be to create a constructor which as parameters has the two *Cube* objects.

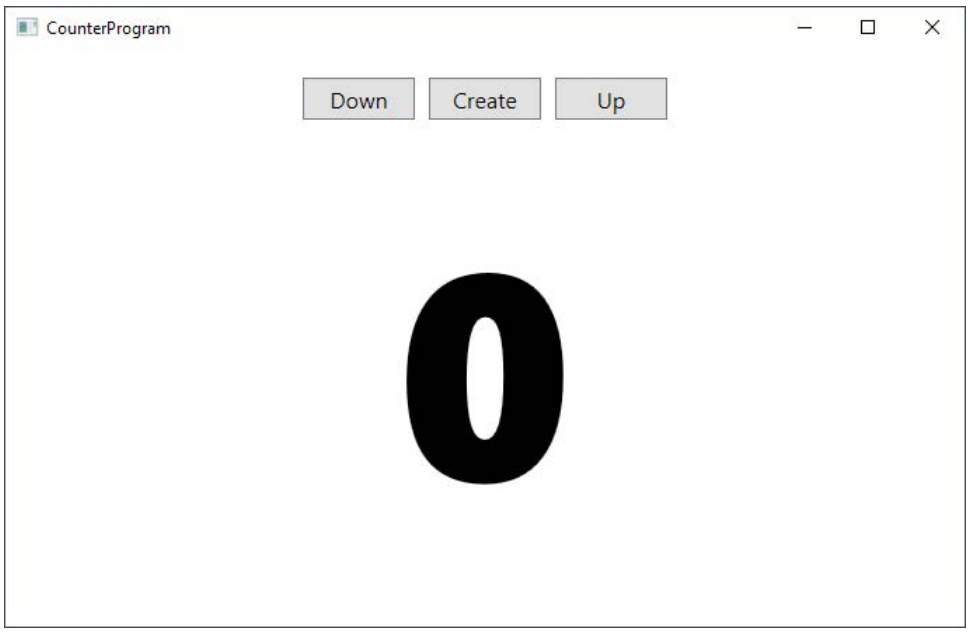
The program has the following *Main()* method:

```
static void Main(string[] args)
{
  Pair p1 = new Pair();
  p1.Toss();
  Pair p2 = p1;
  Console.WriteLine(p2);
  p1.Toss();
  Console.WriteLine(p1);
  Console.WriteLine(p2);
}
```

The first statement creates a *Pair* object. Note the use of the *new* operator. It should not be required, as the variable is created on the stack, but if not the compiler will report an error that the variable is not initialized when it reaches the third statement. Also note the second statement which is necessary to create the two *Cube* objects. In the third line another *Pair* object is created, which is set equal to *p1*. Note that this means that the object *p1* is copied to *p2*, but what is copied, is the two instance variables, and they references to two *Cube* objects. The result is that the two *Pair* objects, each has their own reference variables, but they refer to the same *Cube* objects. It becomes clear if you *Toos()* with one *Pair* (*p1*):

EXERCISE 1: A COUNTER

You must write a program which opens the window:



The window shows the value of a counter, and the window has buttons to move the counter up and down, and a button to create a new counter.

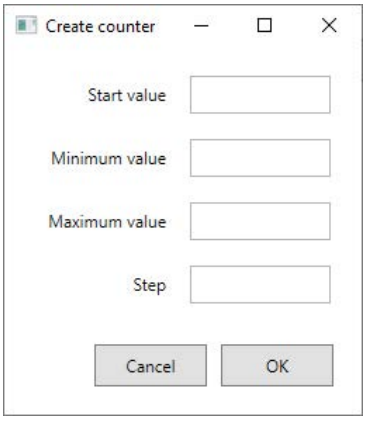
A counter is an object with four values:

- *value*, which is the current value of the counter
- *max*, which is the maximum value
- *min*, which is the minimum value
- *step*, which indicates how much the counter must be moved when it is counting up or down

If the value of the counter becomes larger than maximum it must swap around and start at minimum. Corresponding, if the value becomes less than minimum it must swap around and start at maximum.

You must write the counter as a struct with four private instance variables and read-only properties. You must add the necessary constructors to the struct when *value* as default should be the value of *min*, *min* as default should be 0 and *step* as default should be 1. It is a requirement that *step* is greater or equal 1 and *min* is less or equals *max* and *value* has a legal value. Else the constructor(s) must raise an exception. The struct must have methods to count the counter up and down.

When you have written the struct and tested the program for a hard coded *Counter* you should add a dialog box to create a new counter:



When you click OK the event handler must create a new *Counter*. If it is possible (the values are legal) the dialog box must fire an event with the new *Counter* as argument, and *MainWindow* should be listener for this event.

You can think of a *Counter* (and whatever else such one could be used for) as a good candidate for a struct, as it consists of four simple variables and therefore data that can be immediately allocated on the stack. Note that a *Counter* object occupies 16 bytes.

2.1 NULLABLE STRUCT

A variable of a simple type cannot have the value *null*, but sometimes it is useful to also operate with variables of value types that do not have a value, and therefore there are some special value types that can be null. The syntax is simple and you just have to write a question mark after the type as for example

```
long? n;
double? x;
```

Such a type is said to be *nullable*. It is simply a matter of extending the type with a possibility that it may be null.

A *struct* is a value type, and therefore can't be *null*, but it can be defined *nullable*:

```
using System;

namespace Nullables
{
  class Program
  {
    static void Main(string[] args)
    {
      Nullable<Point> p = null;
      Console.WriteLine(": " + p);
      p = new Point(3.14, 1.41);
      Console.WriteLine(": " + p);
      Console.WriteLine(p.Value.Length);
      Point v = p.Value;
      v.x = 11;
      v.y = 13;
      Console.WriteLine(": " + p);
    }
  }
}
```

Here, *p* is not a *Point* but a *Nullable<Point>*, a *Point*, which may be *null*. What happens is that the value type is encapsulated in a reference type. Above, *p* is initially *null*. Next, *p* is set to a new *Point* object, but you now have hidden the properties of the type *Point*. Instead, we can refer to the encapsulated object with the property *Value*. Note especially that when you write

```
Point v = p.Value;
```

you get a copy of the encapsulated object.

One should therefore not create nullable variables, unless you have special needs. Note especially that instead of writing

```
Nullable<Point> p = null;
```

you can write

```
Point? p = null;
```

which means the same.

2.2 MORE ON SIMPLE VARIABLES

The simple or built-in types are, for example *int*, *char*, *double*, etc. Each type has a name that is a reserved word, but these names are really just an alias (a name as the compiler knows) for a similar *struct* in the *System* namespace. This means that there are also associated methods to the simple types. For example is *int* an alias for the *System.Int32*, which represents a 4 byte integer and is a struct. The following table that shows the simple types in the namespace *System* and there name as a struct:

C# alias	Type in System	Meaning
sbyte	SByte	8 bit signed integer
byte	Byte	8 bit unsigned integer
short	Int16	16 bit signed integer
ushort	UInt16	16 bit unsigned integer
int	Int32	32 bit signed integer
uint	UInt32	32 bit unsigned integer
long	Int64	64 bit signed integer
ulong	UInt64	64 bit unsigned integer
char	Char	16 bit Unicode character

C# alias	Type in System	Meaning
float	Single	32 floating-point number
double	Double	64 floating-point number
bool	Boolean	8 bit that is *true* or *false*
decimal	Decimal	96 bit decimal number with 28 significant digits

You should note that *string* is not mentioned in the table as it is not a struct but a class.

The simple types can basically be divided as

- *char*
- *bool*
- types to integers (*sbyte*, *byte*, *short*, *ushort*, *int*, *uint*, *long*, *ulong*)
- types to decimal numbers (*float*, *double*, *decimal*)

The type char is simple, since it represents the individual characters as a 16-bit numeric code in the *unicode* system. There are tables that show which encodes each character has. Below is shown a method that prints the characters with codes from 32 up to and including 255:

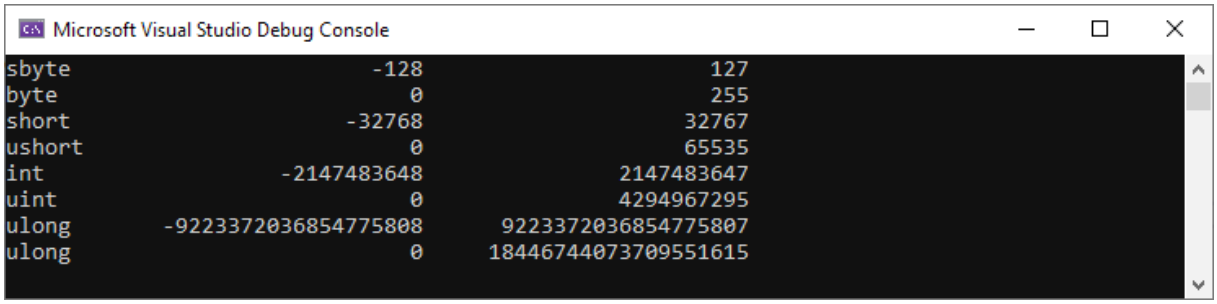
```
static void Test1()
{
  for (char c = ' '; c <= 255; ++c) Console.WriteLine("{0}{1, 4:D}",
  c, (int)c);
}
```

Note especially the comparison in the *for* statement where a *char* is compared with an *int*, and that the ++ operator also makes sense for a *char*. By comparison, the characters code number is compared, and ++ works because it is the code that counted.

To the type *bool* can't be made more comments, but the number types are worth a closer look. A *sbyte* fill 1 byte, and thus 8 bits. It can therefore represent any integer that binary can be written with 8 bits, which are the numbers from -128 to 127. The slightly oblique interval has to do with the internal representation where a negative integer is represented by its 2-complement. If instead you have a *byte*, its values range from 0 to 255. The difference between a *sbyte* and a *byte* is thus simply a parallel shift of the range which is representative of the type in which the one is a symmetrical range of 0, while the other is the non-negative

integers starting with 0. The same is true for the types *short*, *int* and *long* the difference is only the size of the range of integers, which are available. The following method prints these intervals for all 8 integer types:

```
static void Test2()
{
  Console.WriteLine("sbyte {0, 25:D}{1, 25:D}", sbyte.MinValue, sbyte.
  MaxValue);
  Console.WriteLine("byte {0, 25:D}{1, 25:D}", byte.MinValue, byte.
  MaxValue);
  Console.WriteLine("short {0, 25:D}{1, 25:D}", short.MinValue, short.
  MaxValue);
  Console.WriteLine("ushort {0, 25:D}{1, 25:D}", ushort.MinValue,
  ushort.MaxValue);
  Console.WriteLine("int {0, 25:D}{1, 25:D}", int.MinValue, int.
  MaxValue);
  Console.WriteLine("uint {0, 25:D}{1, 25:D}", uint.MinValue, uint.
  MaxValue);
  Console.WriteLine("ulong {0, 25:D}{1, 25:D}", long.MinValue, long.
  MaxValue);
  Console.WriteLine("ulong {0, 25:D}{1, 25:D}", ulong.MinValue, ulong.
  MaxValue);
}
```



You can't just copy an integer of one type into another type without making a type cast, but the general rule is that you can copy a smaller type into a larger type. If you try to translate the following code

```
static void Test3()
{
  sbyte b1 = 2;
  byte b2 = b1;
  int n = 3;
  short s = n;
  long t = n;
}
```

you will get two translation errors. The statement

```
byte b2 = b1;
```

gives an error because you try to copy an integer which may be negative in a variable that can only contain no negative numbers. Similarly, the statement

```
short s = n;
```

cause an error when trying to copy 4 bytes into 2 bytes. In both cases the problem is solved with an explicit type cast:

```
static void Test3()
{
  sbyte b1 = 2;
  byte b2 = (byte)b1;
  int n = 3;
  short s = (short)n;
  long t = n;
}
```

As a last remark concerning integers, I will mention the possibility of statements that works with integers as hexadecimal digits:

```
int a = 0x1abc23;
```

where 0x tells the compiler that the value should be interpreted as a hexadecimal number.

Back there are the decimal numbers that are represented by the types *float*, *double* and *decimal*. The first two is so-called floating-point numbers, while the latter is a decimal number. The difference is that the *float* and *double* meets a very large interval with fewer significant digits, while the latter supports many significant digits and return a more limited range. Common to the three types is that they represent only a portion of the numbers within the intervals that they span, simply because that every interval contains infinitely many real numbers, and an infinite amount can't be represented with a finite number of bits. It is therefore important to realize that when working with decimal numbers, that many of the results are rounded off and approximates the values, a relationship which is also known from an ordinary calculator. Especially in the context of comparisons it is important to be aware of this fact.

The difference between *float* and *double* are, how big a range, they span. The internal representation comprises of a sign, a mantissa with the digits and an exponent. If for example you calculates 3.14^{50}

```
Console.WriteLine(Math.Pow(3.14, 50));
```

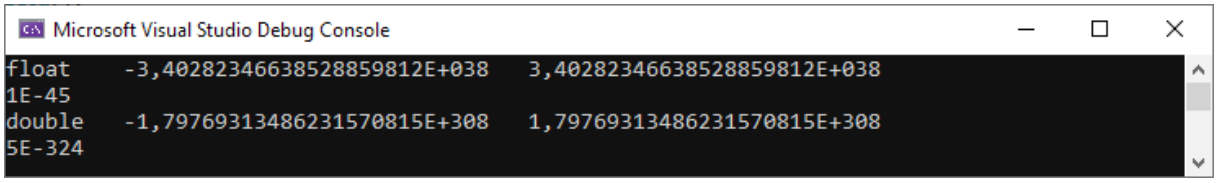
you get the result 7.02234890660215E+24 that means $7.02234890660125 * 10^{24}$.. This means that a *double* works with approximate 15 significant digits. The above is thus a rounded result.

Note that you can also use that notation for a constant, for example

```
double x = 1234.678E+20;
```

The types *float* and *double* defines several constants and some of them are used in the following method:

```
static void Test4()
{
  Console.WriteLine("float {0, 30:E20}{1, 30:E20}", float.MinValue,
  float.MaxValue);
  Console.WriteLine(float.Epsilon);
  Console.WriteLine("double {0, 30:E20}{1, 30:E20}", double.MinValue,
  double.MaxValue);
  Console.WriteLine(double.Epsilon);
}
```



Here you can see the intervals the two types of spans, but you should also notice the constant *Epsilon*, which indicates the smallest positive number. Values are also available which do not represent numbers:

- `double.PositiveInfinity` representing the plus infinity
- `double.NegativeInfinity` representing minus infinity
- `double.NaN` that means a value that does not represent a number

The type decimal spans a smaller range, but in return for up to 29 significant digits:

```
static void Test5()
{
  Console.WriteLine(decimal.MinValue);
  Console.WriteLine(decimal.MaxValue);
  Console.WriteLine(decimal.Zero);
  Console.WriteLine(decimal.One);
  Console.WriteLine(decimal.MinusOne);
  decimal x = 2;
  Console.WriteLine(Sqrt(x));
}
```

Note that the type defines several constants. Also note the last statement that prints the square root of 2. The square root function in the class *Math* is not implemented for variables of the type *decimal* then if you want to determine the square root of a *decimal* as a *decimal*, you have to write the function:

```
static decimal Sqrt(decimal x)
{
  decimal y = x;
  while (true)
  {
    decimal v = y + x / y;
    decimal z = v / 2;
    if (Math.Abs(y - z) <= decimal.Zero) break;
    y = z;
  }
  return y;
}
```

In general, the simple types are *unchecked*. That is, that they not tested for overflow if a value is too large. If, for example you performs the following method

```
static void Test6()
{
  int a = 1234567;
  int b = 1234567;
  Console.WriteLine(a * b);
}
```

you get the following result which is obviously not correct (the result can't be within the range as an *int* span):

When it is so, it is because it takes time to test for overflow, and if it should happen for all calculations in a program, it could reduce the program's effectiveness. Instead they have left it to the programmer to deal with situations that can cause overflow. This can be done as follows:

```
static void Test6()
{
  checked
  {
    int a = 1234567;
    int b = 1234567;
    Console.WriteLine(a * b);
  }
}
```

If you execute the method now, the program will stop with an error message, an exception. It is also possible to set an option to the compiler that all code should be checked. If this is done, there is also an *unchecked*, which can be used to select the blocks of which one does not wish to be *checked*.

When simple types are stucts they are as mentioned also nullable as the following method shows:

```
static void Test8()
{
  int a = 0;
  int? b = null;
  Console.WriteLine("a = " + a);
  Console.WriteLine("b = " + b);
  b = 3;
  Console.WriteLine("b = " + b);
  int?[] t = new int?[5];
  for (int i = 0; i < t.Length; ++i) t[i] = Number();
  Print(t);
  for (int i = 0; i < t.Length; ++i) t[i] = Number() ?? 0;
  Print(t);
}
```

```
static void Print(int?[] t)
{
  foreach (int? n in t) Console.Write("{0, -3}|", n);
  Console.WriteLine();
}

static int? Number()
{
  if (rand.Next(2) == 1) return rand.Next(1, 10);
  return null;
}
```

EXERCISE 2: FIBONACCI NUMBERS

The Fibonacci numbers are the integers

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

That is the two first Fibonacci numbers are 0 and 1, and from there you get the next number as the sum of the two previous numbers.

The first Fibonacci number (that is 0) has the index 0, the next index 1 and so on.

Create a console application which you can call *FibonacciProgram*.

Add a struct called *Fibonacci* which must represents the Fibonacci numbers:

```
public struct Fibonacci
{
  private ulong value;  // the value of the Fibonacci number
  private int index;    // the index of the Fibonacci number
  private ulong temp;  // the previous Fibonacci number if it exists

  // Creates the Fibonacci number with index n.
  // If n is negative it should be the first Fibonacci number.
  // If n is too large the constructor must raise an exception.
  public Fibonacci(uint n)
  {
  }

  // Creates the first Fibonacci number which is greater than or equal
     number.
  // If n is negative it should be the first Fibonacci number.
  // If not such Fibonacci number exsists the constructor must raise
     an exception.
  public Fibonacci(ulong number)
  {
  }

  public ulong Value
  {
    get { return value; }
  }

  public int Index
  {
    get { return index; }
  }

  // Calculates the next Fibonacci number.
  // If possible the method returns true and else the method returns
     false.
  public bool Next()
  {
  }

  // Calculates the previous Fibonacci number.
  // If possible the method returns true and else the method returns
     false.
```

```
  public bool Prev()
  {
  }

  // Returns where number is a Fibonacci number.
  public static bool IsFibonacci(ulong number)
  {
  }
}
```

When you have written the struct you must add a test method the *Program* class which prints all the Fibonacci numbers. Perform the test method by calling it from *Main()*.

Write also a test method which prints all Fibonacci numbers less than 1000000000 but in descending order.

AS the last write a test method where the user must enter a number, and the method must test if it is a Fibonacci number. This operation should be repeated until the user enter an empty string.

3 ENUM

In the previous book I used *enum* types to define values and colors for playing cards, and an *enum* is a value type which assigns names to constants of one of the types *byte*, *short*, *int* and *long*. As default are assigned constants values from 0 onwards, but you can also explicitly assign a constant a particular value. An enumeration is declared in the following manner:

```
public enum WeekDay : byte
{
  Monday,
  Tuesday,
  Wednesday,
  Thursday,
  Friday,
  Saturday,
  Sunday,
  Error = 10
}
```

which declared 8 constants. The type is here *byte* and the constant *Monday* has the value 0, Tuesday has the value 1, etc. The constant *Error* is initialized explicitly with the value 10.

In the following method, the user must enter a text: Mo, Tu, We, Th, Fr, Sa or Su. This text is then converted to an *enum* that is printed on the screen.

```
static void Test1()
  {
    WeekDay day;
    Console.Write("(Mo, Tu, We, Th, Fr, Sa, Su)? ");
    string text = Console.ReadLine();
    switch (text)
    {
      case "Mo":
        day = WeekDay.Monday; break;
      case "Tu":
        day = WeekDay.Tuesday; break;
      case "We":
        day = WeekDay.Wednesday; break;
```

```
      case "Th":
        day = WeekDay.Thursday; break;
      case "Fr":
        day = WeekDay.Friday; break;
      case "Sa":
        day = WeekDay.Saturday; break;
      case "Su":
        day = WeekDay.Sunday; break;
      default:
        day = WeekDay.Error; break;
    }
    Console.WriteLine(day);
}
```

Note the *switch* statement that switches on a string. It is allowed in C#. Note also that the program writes *Sunday*, if the user enter *Su*, that is the name of the constant and then the enum value.

An enum as above define 8 values, but that does not mean that a variable of that type takes up 8 bytes. It takes up only one byte, and it is only a question that there is assigned a name determined by the variable's value.

The default type is *int*, and you can simply write:

```
public enum CardColor
{
    Diamonds, Hearts, Spades, Clubs
}
```

and the four names has the values 0, 1, 2, and 3. If you instead writes:

```
public enum CardColor1
{
  Diamonds = 100, Hearts, Spades, Clubs
}
```

the values are 100, 101, 102 and 103. The values do not have to be continuous and they do not even have to be sequential and it is allowed to write

```
public enum CardColor2
{
  Diamonds = 300, Hearts = 100, Spades = 400, Clubs = 200
}
```

An enum gets its functionality from a type *System.Enum* where it inherits some methods. Most important is that you can get the values of an enum type as an array. The following method prints all values in an enum:

```
static void Test2()
{
  Array arr = Enum.GetValues(typeof(CardColor1));
  for (int i = 0; i < arr.Length; ++i) Console.WriteLine(arr.GetValue(i));
}
```

Here you should note that *Array* is a class which is an array with objects of some type. *typeof* is an operator which can be used to determine the type based on the type name, and the class *Enum* has a method *GetValues()* which returns the values for this type if it is an enum. With this array you can loop over all values in the enum:

The following method do the same, but for another enum type:

```
static void Test3()
{
  Array arr = Enum.GetValues(typeof(CardColor2));
  for (int i = 0; i < arr.Length; ++i) Console.WriteLine("{0} {1}",
  (int)arr.GetValue(i), arr.GetValue(i));
}
```

You should note that you can type cast the value of an *enum* to an *int*, and when you loop over an *enum* the sequence is ordered after the *int* values and not how the *enum* is initialized:

Enum types are effective and it is a better solution than to define a number of constants for a concept, and enum types can be used to increase readability.

4 MORE ON CLASSES AND INTERFACES

In the book C# 1 I have described classes and to some extent interfaces, but there is a lot more to add, as classes and interfaces are the very basic concepts of object oriented programming, and this chapter therefore focuses exclusively on classes and interfaces and including the concepts associated with them. Classes and interface types are as mentioned many times reference types, but in C# all types are part of the same class hierarchy.

4.1 VISIBILITY

I will start with visibility which is something that has significance for all types. If you look at all examples in this book and the previous books I have (almost) always defined variables as *private* and methods and properties as *public*. It defines from where members can be used of referenced (from where there is access to the member), where private members only can be used from the class where they are defined, while public members can be used from everywhere. In this section I will explain visibility in more details, as there are other possibilities than *private* and *public*, and as both types and type members can have a visibility.

Note first that an assembly is a file that contains compiled types and then an *exe* file or a *dll*. Assemblies are explained in detail later. In general there are the following possibilities called accessibility modifiers:

- *public*, the type or member can be referenced by any other code in the same assembly or any other assembly
- *private*, the type or member can be referenced only from code in the same class or struct
- *protected*, the type or member can be referenced only by code in the same class, or in a class that is derived from this class
- *internal*, the type or member can be referenced by any code in the same assembly, but not from another assembly
- *protected internal*, the type or member can be referenced by any code in the same assembly in which it is defined, or from within a derived class in another assembly
- *private protected*, the type or member can be referenced only within the defining assembly, by code in the same class or in a type derived from that class
- nothing, the same as default

Not all access modifiers can be used by all types or members and in all contexts. In some cases the accessibility of a member is constrained by the accessibility of its containing type. The following list summarizes the most important rules, but you should be aware that the some of the rules relate to concepts that will only be explained later

1. Classes and structs that are declared in a namespace and not nested can be either *public* or *internal*, where *internal* is default.
2. Struct members, including nested classes and structs can be *public*, *internal* or *private*, where *private* is default.
3. Class members, including nested classes and structs can be *public*, *protected internal*, *protected*, *internal, private protected* or *private*, where *private* is default.
4. Private nested types are not accessible from outside the containing type.
5. Derived classes can not have greater accessibility (that means give more access) than there base type.
6. Class members can have all 6 accessibility levels, but struct members have only 3 levels as a struct can not be inherited.
7. Usual, the accessibility of a member is not greater than the containing type, but a *public* member of an *internal* class might be accessible from another assembly if the member implements an interface method or overrides a virtual method defined in a *public* base class.
8. The type of a variable, a property or an event must at least have accessible as the member itself. Similarly, a return type, a parameter type, an indexer or a delegate must be at least as accessible as the member itself.
9. User defined operators must always be defined *public* and *static*.
10. *Finalizers* can not have accessibility modifiers.
11. The accessibility modifier *protected internal* means *protected* or *internal*.
12. Accessibility modifiers can not be used for interface members, and they are always *public*.
13. Accessibility modifiers can not be used for enum members, and they are always *public*.
14. Delegates behave like classes.

There are many rules, and this is especially true for members of a type and here the following table may provide an overview:

Members of	Default accessibility	Allowed accessibility modifiers
enum	public	none
interface	public	none
class	private	public
		protected
		internal
		private
		protected internal
		private protected
struct	private	public
		internal
		private

4.2 DESIGN PATTERNS

I will start this chapter on classes with a program that can convert an amount in one currency to an amount in another currency. The program consists of several classes, but basically there is nothing new regarding classes, but the program introduces the concept of design patterns as two simple patterns. A design pattern is a certain way to solve certain problems that are general in nature and are appearing in many different situations. It is natural to seek a standard for how to use proven methods for solving such a problem, and that's what a design pattern is. In the example I presents the patterns

1. a *singleton*
2. an *iterator pattern*

The program is called *CurrencyProgram* and has a simple model class, which is called *Currency* that represents a currency with three properties:

- currency code (that is a code on 3 characters)
- currency name (that must not be blank)
- currency rate that should be a non-negative number

The code of the class is shown below and do not require further explanation:

```
namespace CurrencyProgram
{
  public class Currency
  {
    private string code; // currency code
    private string name; // currency name
    private double rate; // currency rate

    public Currency(string code, string name) : this(code, name, 0)
    {
    }

    public Currency(string code, string name, double rate)
    {
      if (!IsValuta(code, name, rate)) throw new Exception("Illegal
      currency");
      this.code = code;
      this.name = name;
      this.rate = rate;
    }

    public string Code
    {
      get { return code; }
    }

    public string Name
    {
      get { return name; }
      set
      {
        if (value == null || value.Length == 0)throw new
        Exception("Illegal currency name");
        name = value;
      }
    }

    public double Rate
    {
      get { return rate; }
      set
```

```
      {
        if (value < 0) throw new Exception("Illegal currency rate");
        rate = value;
      }
    }

    public override string ToString()
    {
      return string.Format("{0}: {1} ({2:F4})", code, name, rate);
    }

    public static bool IsValuta(string code, string name, double rate)
    {
      return code != null && code.Length == 3 && name != null && name.
      Length > 0 && rate >= 0;
    }
  }
}
```

You should note that the above class is a typical model class that describes objects in the program's problem area. Also note that the class has two constructors and the use what is called constructor chaining where the one constructor calls the other using the work *this*. In this way a constructor can transfer parameters to another constructor and you can save some code and if you at some point has to modify the code you should only change the code in one place.

The singleton pattern

The next class is called *CurrencyTable*, and is a class that defines the *Currency* objects that the program knows and has to work with. The program must have an object of the type *CurrencyTable*, and it is an object, which in principle should always be there and be available no matter where in the program you are. Since it is very often that situation arises that a program must use an object of a particular type, and you want to ensure

1. the object is always there, without explicitly being created
2. the object is available to all other objects in the program
3. there certainly exists only one object of that type

there has been defined a particular design pattern for how the class for such an object should be written. This design pattern is called a *singleton*. The class can be written as follows:

```
public class CurrencyTable : IEnumerable<Currency>
{
  private static readonly CurrencyTable instance = new CurrencyTable();
  private List<Currency> table = new List<Currency>();

  static CurrencyTable()
  {
  }

  private CurrencyTable()
  {
    Init();
  }

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

  public Currency this[string code]
  {
    get
    {
      foreach (Currency c in table) if (c.Code.Equals(code)) return c;
      throw new Exception("Illegal currency");
    }
  }

  public IEnumerator<Currency> GetEnumerator()
  {
    return table.GetEnumerator();
  }

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

```
public bool Update(Currency currency)
{
  foreach (Currency c in this)
    if (c.Code.Equals(currency.Code))
    {
      try
      {
        c.Name = currency.Name;
        c.Rate = currency.Rate;
        return true;
      }
      catch
      {
        return false;
      }
    }
  table.Add(currency);
  return true;
}

private double Parse(string text)
{
  text = text.Replace(".",
    CultureInfo.CurrentCulture.NumberFormat.NumberDecimalSeparator);
  return double.Parse(text);
}

private void Init()
{
  try
  {
    foreach (string line in rates)
    {
      string[] elems = line.Split(';');
      if (elems.Length == 3) table.Add(new Currency(elems[0],
      elems[1], double.Parse(elems[2])));
      else throw new Exception("Error: " + line);
    }
  }
```

```
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
      Environment.Exit(1);
    }
  }

  private string[] rates = {
    "Danske kroner;DKK;100.00",
    "Euro;EUR;746.00",
    ...
  };
}
```

This time there are more details to note. The class has an instance variable, which is called *table* and is a *List<Currency>* and should be used for the *Currency* objects. The class also has a *private static* variable named *instance* that has the type *CurrencyTable* (the type of the class itself) and is initialized to and object of the current class. This variable is an important part of the singleton pattern. The class has a constructor, but you should note that the constructor is defined *private*, and when the class has no other constructors, it means that other objects cannot instantiate objects of this class. It is another important part of the singleton pattern. The only object which can be instantiated is the static object referenced with the static variable *instance* and it will automatically be there.

The *List<Currency>* must be initialized with *Currency* objects, and it requires currency data in the form of a rate list. It can, for example be found on

http://www.nationalbanken.dk/da/statistik/valutakurs/Sider/Default.aspx

and an example is laid out in the bottom of the class as an array of strings (only a few examples of currencies are shown). You should note that these are exchange rates in relation to Danish currency. The method *Init()* uses this the array to create *Currency* objects, and the method *Init()* is called from the *private* constructor. It is of course a little pseudo, since it is not the current exchange rates, but if you wants you can update the table.

The class has a static property called *Instance*, which returns the static variable *instance*. It is the last and perhaps most important part of the singleton pattern. If you want to execute a method on the only existing *CurrencyTable* object you need to use that property.

The iterator pattern

In terms of the other methods in the class, there is nothing to explain except the method *GetEnumerator()*. The class is an example of a collection and, in this case a collection of *Currency* objects. That kind of collections with objects of a certain kind occurs very often in practice. One of the operations that almost always is needed is being able to iterate over the collection with a loop. This requires access to the objects and it is often resolved by means of a pattern which we call the *iterator pattern*. The class *CurrencyTable* implements this pattern as follows.

The class implements an interface, which here is called *IEnumerable<Currency>*. This means that the class must implements two methods with the following signatures:

```
public IEnumerator<Currency> GetEnumerator()
System.Collections.IEnumerator System.Collections.IEnumerable.
GetEnumerator()
```

where *IEnumerator<Currency>* is an interface that defines methods, so you can iterate over the collection of *Currency* objects. In this case it is particularly simple, since a *List* has such an iterator, and the method can therefore be written as shown in the class.

The need for two versions of the method is something you should simply accept and the last version always needs to be written as shown above, but in short it is because C# was at one point expanded to support generic methods.

How to write your own iterator, I will return to later, but the result of the iterator pattern is that you can use the *foreach* construction:

```
foreach (Currency c in CurrencyTable.Instance) { ... }
```

In reality it is nothing but a short way of writing

```
for (IEnumerator<Currency> itr = CurrencyTable.Instance.GetEnumerator();
  itr.MoveNext(); ) { ... }
```

The class *CurrencyTable* also has an *indexer*, which I have used before and is a property where the value is referenced with an index or parameter which here is a *string*:

```
public Currency this[string code]
{
  get { .... }
}
```

An indexer is a form for operator overriding, but has nothing with the iterator pattern to do. I means that you in a program can write a statement as

```
Currency c = CurrencyTable.Instance["EUR"];
```

and then (in this case) us a *CurrencyTable* object with the same syntax as you use an array or another collection.

The two classes *Currency* and *CurrencyTable* are the program's model.

The Calculator

The program has a class *Calculator* that by using the above model classes must perform the currency conversion, including validation of parameters from the user interface:

```
public class Controller
{
  public double Calculate(String amount, Currency from, Currency to)
  {
    try
    {
      return Calculate(double.Parse(amount), from, to);
    }
```

```
    catch (Exception ex)
    {
      throw new Exception("Illegal amount");
    }
  }

  public double Calculate(double amount, Currency from, Currency to)
  {
    if (from == null || to == null) throw new Exception("No currency");
    return amount * from.Rate / to.Rate;
  }

  public List<string> Update(string filename)
  {
    List<string> errors = new List<string>();
    try
    {
      StreamReader reader = new StreamReader(filename);
      for (String line = reader.ReadLine(); line != null; line =
      reader.ReadLine())
      {
        string[] elems = line.Split(';');
        try
        {
          if (elems.Length == 3) CurrencyTable.Instance.Update(new
          Currency(elems[1], elems[0], Parse(elems[2])));
          else throw new Exception();
        }
        catch
        {
          errors.Add(line);
        }
      }
      reader.Close();
    }
    catch
    {
    }
    return errors;
  }
}
```

There are two overloads of the method *Calculate()*, which is the method that will perform the conversion. The difference is only the type of the first parameter, where the first converts this from *string* to a *double*. Furthermore, there is a method *Update()* which parameter is a

file name. The file name will represent a text file that contains a list of currencies when the file must have the same format as the table been laid out in the class *CurrencyTable*. The method updates the model corresponding to the content of the file. The method returns a *List<string>* which contains the lines from the file that could not be successfully parsed into a *Currency*.

MainConsole

In this case then code for the *Main()* class is moved to its own class called *MainConsole*. There is no special justification for it besides pointing out that it is an opportunity. The program should works as follows:

The program starts to print an overview of the current currencies. In turn, there is a loop in which the user for each iteration must

1. Enter the currency code for the currency to be converted from
2. Enter the currency code for the currency to be converted to
3. Enter the amount to be converted

and then the program prints the result of the conversion. This dialog is repeated until the user just types *Enter* for the first currency code. The program is represented by the class *MainConsole*. I will not show the code here, but when you study the code, notice how I use, that the class *CurrencyTable* is a singleton, and that it implements the iterator pattern.

With this class ready the *Program* class can be written as:

```
namespace CurrencyProgram
{
  class Program
  {
    static void Main(string[] args)
    {
      new MainConsole();
    }
  }
}
```

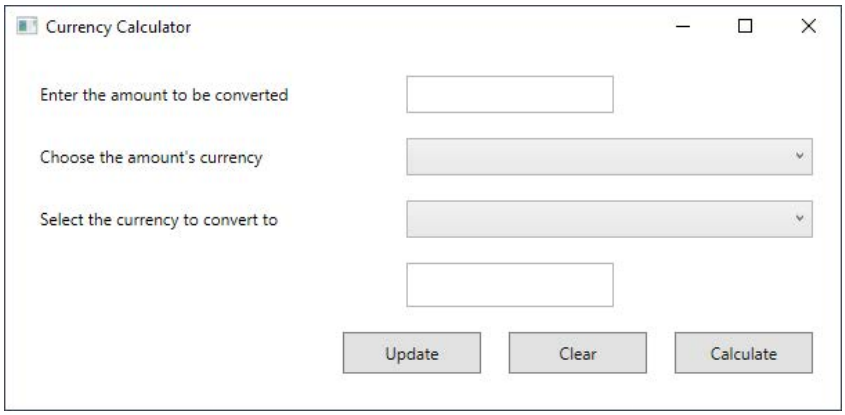
EXERCISE 3: THE CURRENCY PROGRAM

In this exercise you must write a version of the above program that works in exactly the same way, but the program must this time have a graphic user interface. Create a new WPF project which you can call *CurrenceProgram*. Then add the three classes from the previous version of the program:

1. *Currency*
2. *CurrencyTable*
3. *Calculator*

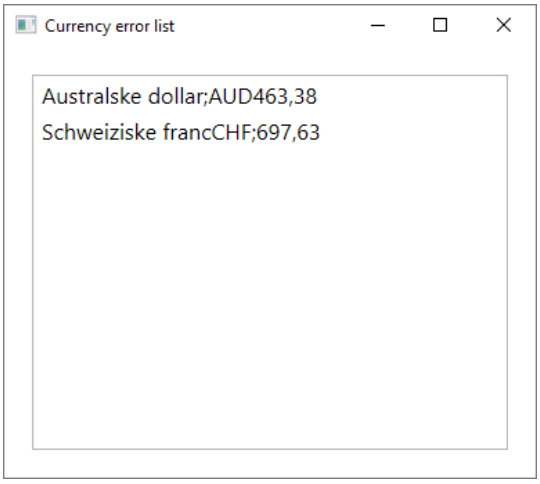
These classes should all be used without any changes.

Next you must design the window as shown here:



Each combo box must contains all the currencies defined by the class *CurrencyTable,* and when you enter an amount in the upper field, select currencies to convert from and to and click the button *Calculate* the lower *TextBox* must be updated with the result.

If the user clicks the *Update* button the program must open a file dialog so the user can browse the file system for a semicolon delimited file with currencies rates, which must be used to update the model. If the update results in errors (the method *Update()* in the class *Calculator*) the program must open a window with the error list:



4.3 INTERFACES

This section describes interfaces and in the next chapter I describes inheritance that is two object-oriented concepts already briefly mentioned in C# 1, and in fact in C# 2 I have used both concepts without explicitly draw attention to it. An interface defines the functionality that a class must have, while inheritance is a question about how to extend a class with new properties in terms of new instance variables or methods. Immediately the two thing do not seems having much to do with each other, but they have largely, and therefore I treated both concepts subsequent.

To illustrate both concepts I need some examples, and I want to use the same examples as in the book C# 1, namely classes concerning students, and classes concerning books in a library, and much of what follows will also address how these classes can be modified. The start for this section is a project *Students3*, that is exactly the same project as the project *Students2* in the book C# 1 and thus a project containing three model classes:

1. *Subject*
2. *Course*
3. *Student*

Before proceeding, it is important that you recall the project Students2 and its classes.

Technically, an interface is a type, and it is a reference type. You can therefore do basically the same with an interface as with a class, except that an interface cannot be instantiated. You cannot create an object whose type is an interface, but otherwise an interface may be used as a type for parameters, return values and variables.

Conceptually, an interface is a contract, and an interface can tell that an object has a specific property. The interface specifies only what you can do with the object, but not how that behavior is implemented, and that's exactly what you want to achieve. Who that has to use the object, only know it through the defining interface (that is the contract), but do not know anything about how the class that underlies the object is made. So long this class comply with the contract, and does not change the interface, the class can be changed without it affects the code that uses the object.

Typically, an interface has only signatures of methods (but there are other options, as explained below). An interface is defined by almost the same syntax as a class, but with the difference that the word *class* is replaced by the word *interface*. I will, as mentioned, often use the convention that I let the name of an interface start with a big I that is a standard in C#. The class *Subject* represents a teaching subject and thus a concept that can be included in a program concerning education. The concept can be defined as follows (see the project *Students3*):

```
namespace Students3
{
  public interface ISubject
  {
    /*
     * The id for this ISubject
     */
    public string Id { get; }

    /*
     * The name for this ISubject.
     * A subject must have a name and if you not specify a name the
     * property raises an exception.
     */
    public string Name { get; set; }
  }
}
```

The interface is called *ISubject*, and the only thing you can read is that a concept that is an *ISubject*, has two properties where the comments explains what these properties are used for.

Once you have the interface, a class can implement this interface:

```
public class Subject : ISubject
{
```

and it is the only change needed in the class *Subject*, because the class already (implements) the two properties defined by the interface. So the question is what you have achieved with that and so far nothing, but I will make some changes in the class *Course*. It has a variable of the type *Subject*, and I will change the definition of this variable, so that it instead has the type *ISubject*:

```
private ISubject subject;
```

When a class implements an interface, the class's type is special the type of this interface, and therefore it makes sense to say that a *Subject* object is also an *ISubject*. I have also changed all the parameters of constructors and methods whose type is *Subject*, as their type now is *ISubject*. You should note that the program still can be translated and run. The result is that the class *Course* no longer knows the class *Subject*, but it knows only *Subject* objects through the defining interface *ISubject*. A *Course* know what it can do with a *Subject* (it knows the contract), but not how a subject is implemented. The two classes are now more loosely coupled than they were before, and that means that you get a program that is easier to maintain, as you can change the class *Subject* without it matters for classes that use *Subject* objects.

The fact that in this way to define the classes by means of an interface, and other classes only know a class through its interface is a principle or pattern, commonly referred to as *programming to an interface*.

In the same way a *Course* can be defined by an interface:

```
public interface ICourse
{
  /*
    * A cource is identified by the subjects id and the year
    * Return the course ID composed of the year, the subject's id and
      year
    * separated by a hyphen.
    */
  public string Id { get; }

  /*
    * The year where the course is held.
    */
  public int Year { get; }

  /*
    * True, if the student has completed the course
    */
  public bool HasCompleted { get; }

  /*
    * Returns the character that the student has achieved
    * Throws an exception if a student has not obtained a character
    * or the course is assigned an illegal score.
    */
  public int Score { get; set; }

  /*
    * Assigns this course a character.
    * Throws exception if the score is illegal
    */
  public void SetScore(string score);

  /*
    * Returns the subject for this course
    */
  public ISubject Subject { get; }
}
```

There is not much to explain, but you should note two things:

1. there is defined a new property *Subject*, which is not part of the class *Course*
2. the method *ToString()* is not defined, and it was not defined in the interface *ISubject*

The class *Course* must implements the interface *ICourse*, and because of the first of the above observations, it is necessary to add the new property. A class that implements an interface must implement all the methods and properties that the interface defines. Below I show the class *Course* that now implements the interface *ICourse* where I have deleted all comments and most of the code:

```
public class Course : ICourse
{
  private int year;                   // year for when the course
  private ISubject subject;           // the subject that the course
                                         deals
  private int score = int.MinValue;   // the score obtained in the
                                         subject

  public Course(int year, ISubject subject)
  {
    ...
  }

  public Course(int year, string id, String name)
  {
    ...
  }

  ...

  public ISubject Subject
  {
    get { return subject; }
  }

  public override string ToString()
  {
    return subject.ToString();
  }

  ...
}
```

You should note:

- the syntax to implement an interface
- there is added a new property *Subject* that returns an *ISubject*

After these changes, the program can be translated and run.

If you have to comply with the principle of programming to an interface, the class *Student* must be altered so that all instances of the type *Course* are changed to *ICourse*. It is necessary to change the class *Student* so all references in *Student* to the type *Course* are changed to *ICourse*, but then the class *Student* is also decoupled from the class *Course* and know only a course by the defining interface *ICourse*. When you have updated the class *Student* and decoupled it from the class *Course* it is also necessary to change one statement in the test program.

An interface can in the same way as a class be public or it can be specified without visibility. In the latter case, the interface is known only within the assembly to which it belongs. In the above examples, everywhere I have defined methods in an interface as *public*, but they are by default, even if you do not write it. I prefer always to write the word *public*, as it clarifies the methods visibility.

I would then add a small change to the class *Student*. The class has a static method *StudentOk()*, which validates the name and email address of a student to test where a student is legal. The controls are quite trivial, since the method only tests whether the value is specified for both the name and address. I will now extend the control, so the method also tests whether the mail address is in the correct format:

```
public static boolean studentOk(String mail, String name)
{
  return IsMail(mail) && name != null && name.length() > 0;
}

public static bool IsMail(string mail)
{
  if (mail == null || mail.Length == 0) return false;
  string pattern = @"^[\w!#$%&'*+\-/=?\^_`{|}~]+(\.[\w!#$%&'*+\-
  /=?\^_`{|}~]+)*" + "@" + @"((([\-\w]+\.)+[a-zA-Z]{2,4})|(([0-9]
  {1,3}\.){3}[0-9]{1,3}))$";
  Regex reg = new Regex(pattern);
  return reg.IsMatch(mail);
}
```

Here, the method *IsMail()* is a method to validate whether a string may be a mail address. You must at this place just accept that method does, but it happens by using a so-called regular expressions, as discussed later.

I've also changed the property *Mail*:

```
set
{
  if (!IsMail(value)) throw new Exception("Illegal mail address");
  this.mail = mail;
}
```

Also a student can be defined by an interface *IStudent*, where I have removed the comments:

```
public interface IStudent
{
  public int Id { get; }
  public string Mail { get; set; }
  public string Name { get; set; }
  public int Count { get; }
  public ICourse this[int n] { get; }
  public ICourse this[string id] { get; }
  public List<ICourse> GetSourses(int year);
  public void Add(ICourse course);
}
```

Classes can implement an interface, which as stated corresponds to that the class comply a contract. Classes, however, can implements multiple interfaces and thus comply with several contracts. If the class implements several interfaces the syntax is to list them as a comma separated list after the colon. As an example, the following interface defines points in the ECTS system:

```
public interface IPoint
{
  public const int YEAR = 60; // number of points a year
  public int ECTS { get; set; }
}
```

The interface is not particularly interesting, as it only in principle defines an integer, but you might assume that a subject must have attached an ECTS. Note that an interface can

define a static field as a static field is not assigned to any object. The class *Subject* could then be written as follows (where I again has removed all comments):

```
public class Subject : ISubject, IPoint
{
  private string id;       // the subject id
  private string name;     // the subject's name
  private int ects = 0;    // the subtect's ECTS

  public Subject(string id, string name, int etcs = 0)
  {
    if (!SubjectOk(id, name)) throw new Exception("The subject ... ");
    this.id = id;
    this.name = name;
    this.ects = etcs;
  }

  ...

  public int ECTS
  {
    get { return exts; }
    set
    {
      if (value < 0) throw new Exception("ECTS must be none-
negative");
      ects = value;
    }
  }

  ...

}
```

The class now implements two interfaces, and must therefore implement the property, the new interface defines. Also, I have expanded a new default parameter to the constructor. You should note, that after the class *Subject* is changed the program can still be translated and executed. A *Subject* object still has the type *ISubject*, but it also has the type *IPoint*, but it is not used.

EXERCISE 4: PROGRAMMING TO AN INTERFACE

Start by creating a new Console Application project called *Library3*. In the book C# 1 you create a project called *Library2*. This project has three classes *Publisher*, *Author* and *Book*. Copy these three classes to this project and rename the namespace declaration to *Library3*.

For each of the three classes *Publisher*, *Author* and *Book*, you must write an interface, and the respective classes must implement the interfaces. After the three classes are defined using interfaces, you must modify the classes code, so the three classes is as loosely coupled as possible, and you should change the main class for something like the following:

```
public static void main(String[] args)
{
  try
  {
    IBook b1 = Create("978-1-59059-855-9", "Beginning Fedora From
    Noice to Professional", 2007, 1, 519, new Publisher(123, "The new
    Publisher"), new Author("Shashank", "Sharma"), new Author("Keir",
    "Thomas"));
    IBook b2 = new Book("978-87-400-1676-5", "Spansk Vin");
    Print(b1);
    Print(b2);
    b2.Released = 2014;
    b2.Edition = 1;
    b2.Pages = 335;
    b2.Publisher = new Publisher(200, "Politikkens Forlag");
    b2.Authors.Add(new Author("Thomas", "Rydberg"));
    Print(b2);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}

private static void Print(IAuthor a) { ... }

private static void Print(IBook book) { ... }

private static IBook Create(string isbn, string title, int released,
int edition, int pages, IPublisher publisher, params IAuthor[]
authors) { ... }
```

4.4 MORE STUDENTS

Above, I have introduced interfaces that defines the properties of an existing class, but in practice, you usually go the other way and defines an interface that defines a concept in the program area of concern, and then you (or let others do it) can write a class that implements that interface. Above, I have introduced definitions of students and courses as *IStudent* and *ICourse* objects, and then it might be natural to define a team of students, where the team must have a name and otherwise is simply a collection of students:

```
public interface IStudents : IEnumerable<IStudent>
{
  public string Name { get; }                 // the team's name
  public void Add(params IStudent[] stud);    // add a student
  public bool Remove(int id);                 // remove a student
  public IStudent this[int id] { get; }      // return a student

  /**
    * Returns all students, where the name contains the value of the
    parameter.
    */
  public List<IStudent> GetStudents(string name);

  /**
    * Returns all students, that has completed in a particular course
    a year.
    */
  public List<IStudent> GetStudents(string id, int year);
}
```

There is not much to say about the interface when the comments are explaining the use of each property / method, but you should note that an interface can inherit another interface, and in this case the interface *IStudents* inherits the interface *IEnumerable<IStudent>*. This means that a class that implements the interface, also has to implement the iterator pattern. Below is a class that implements the interface *IStudents*:

```
public class Students : IStudents
{
  private string name;
  private List<IStudent> list = new List<IStudent>();

  public Students(string name)
  {
    this.name = name;
  }

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

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

  public string Name
  {
    get { return name; }
  }

  public void Add(params IStudent[] stud)
  {
    foreach (IStudent s in stud) list.Add(s);
  }

  public bool Remove(int id)
  {
    for (int i = 0; i < list.Count; ++i)
      if (list[i].Id == id)
      {
        list.RemoveAt(i);
        return true;
      }
    return false;
  }
```

```
  public IStudent this[int id]
  {
    get
    {
      foreach (IStudent s in list) if (s.Id == id) return s;
      throw new Exception("Student not found");
    }
  }

  public List<IStudent> GetStudents(string name)
  {
    List<IStudent> lst = new List<IStudent>();
    foreach (IStudent s in list) if (s.Name.Contains(name)) lst.
Add(s);
    return lst;
  }

  public List<IStudent> GetStudents(string id, int year)
  {
    List<IStudent> lst = new List<IStudent>();
    foreach (IStudent s in list) foreach (ICourse c in s.GetCourses(year))
      if (c.HasCompleted && c.Subject.Id.Equals(id)) lst.Add(s);
    return lst;
  }
}
```

There is not much to explain, but you should note that the class implements the iterator pattern, and that the class does not know the class *Student*, but only know a student through the interface *IStudent*.

As the last concept in this family of types regarding students I will look at an institution, but this time I will not define an interface. The class should instead be written as a singleton, and the program therefore needs to know the actual class, why a defining interface is not quite as interesting. The class is called *Institution*:

```
public class Institution : IEnumerable<IStudents>
{
  public readonly static String NAME = "The Windows University";
  private static readonly Institution instance = new Institution();
  private List<IStudents> list = new List<IStudents>();

  static Institution()
  {
  }

  private Institution()
  {
  }

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

  public IStudents this[string name]
  {
    get
    {
      foreach (IStudents t in list) if (t.Name.Equals(name)) return t;
      throw new Exception("There is no team with the name " + name);
    }
  }

  public void Add(params IStudents[] studs)
  {
    foreach (IStudents t in studs) list.Add(t);
  }

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

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

The class is largely self-explanatory, but you must note that it is written as a singleton, and the pattern is implemented in exactly the same way as I have shown in connection with the currency converter. Also note that the class implements the iterator pattern and note finally that the class knows only the other program classes through there defining interfaces.

Finally, I shows the test program. The main class already has a *Print()* method, which prints an *IStudent*, and I have added the following method to print an *Institution*:

```
private static void Print()
{
  try
  {
    Console.WriteLine(Institution.NAME);
    Console.WriteLine();
    foreach (IStudents studs in Institution.Instance)
    {
      Console.WriteLine(studs.Name);
      foreach (IStudent stud in studs)
      {
        Console.WriteLine();
        Print(stud);
      }
      Console.WriteLine();
    }
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

You should note that both *Print()* methods not dependents on the specific classes. I have added a test method to the *Program* class:

```
static void Test3()
{
  Institution.Instance.Add(
    CreateTeam("Team-A 2015",
      CreateStudent("olga.jensen@mail.dk", "Olga Jensen",
        CreateCourse(2015, "PRG", "Introduction to Java", 5, 7),
        CreateCourse(2015, "OPS", "Operating Systems", 15, 2),
        CreateCourse(2015, "SYS", "System Development", 5, 2)
      ),
```

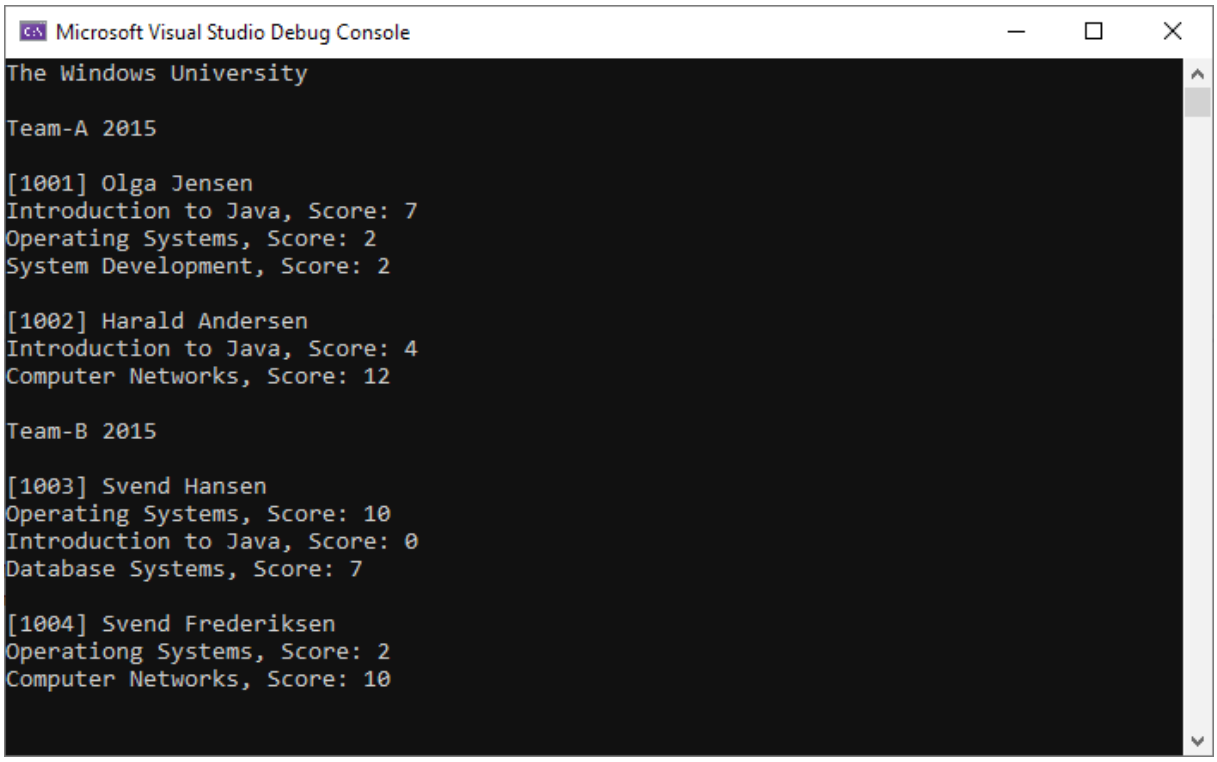
```
      CreateStudent("harald.andersen@mail.dk", "Harald Andersen",
        CreateCourse(2015, "PRG", "Introduction to Java", 5, 4),
        CreateCourse(2015, "NET", "Computer Networks", 10, 12)
      )
    ),
    CreateTeam("Team-B 2015",
      CreateStudent("svend.hansen@mail.dk", "Svend Hansen",
        CreateCourse(2015, "OPS", "Operating Systems", 15, 10),
        CreateCourse(2015, "RRG", "Introduction to Java", 5, 0),
        CreateCourse(2015, "DBS", "Database Systems", 20, 7)
      ),
      CreateStudent("svend.frederiksen@mail.dk", "Svend Frederiksen",
        CreateCourse(2015, "OPS", "Operationg Systems", 15, 2),
        CreateCourse(2015, "NETL", "Computer Networks", 10, 10)
      )
    )
  );
  Print();
}
```

adding two teams to the institution where each team has two students. Finally the method prints the institution. The method uses some helper methods to create objects:

```
private static IStudents CreateTeam(string name, params IStudent[] studs)
{
  IStudents students = new Students(name);
  students.Add(studs);
  return students;
}

private static IStudent CreateStudent(string mail, String name,params
ICourse[] course)
{
  try
  {
    return new Student(mail, name, course);
  }
  catch (Exception ex)
  {
    return null;
  }
}
```

```
private static ICourse CreateCourse(int year, string id, String name,
int ects,
  int score)
{
  try
  {
    ICourse course = new Course(year, id, name);
    course.Score = score;
    ((IPoint)course.Subject).ECTS = ects;
    return course;
  }
  catch (Exception ex)
  {
    return null;
  }
}
```

If you run the program you get the following result:



EXERCISE 5: A LIST OF BOOKS

Create a new Console Application project called *Library4* and copy the 6 types

- *IPublisher*
- *Publisher*
- *IAuthor*
- *Author*
- *IBook*
- *Book*

from the project *Library3. Remember* to change the name of the namespace. You must add an interface defining a book list:

```
public interface IBooklist : IEnumerable<IBook>
{
  /*
    * Method, that adds books to the book list. If the list already
    contains
    * a book with the same ISBN, the book should be ignored.
    */
  public void Add(params IBook[] books);

  /*
    * Method that creates a book and adds it to the list. The book is
      created
    * on the basis from a number of strings which are interpreted as
      follows
    *   ISBN
    *   title
    *   release year
    *   edition
    *   pages
    *   publisher name
    *   an number of pairs of authors' first and last name
    * Only the first two arguments are required.
    * Publisher's number extracted from the ISBN (see problem 1 in the
    book C# 1).
    */
  public void Add(params string[] elem);
```

```
  /*
    * Return the book with a certain isbn.
    */
  public IBook this[string isbn] { get; }

  /*
    * Search books in the book list and returns a list of the books
    * that matches the search values.
    * The list is searched combined so that a book matching the search
    * values if it matches all search values.
    * If a String is null or blank, the criterion is ignored. The same
   * applies to an int that is 0.
    * By searching on strings that should not be distinction between
    * uppercase and lowercase letters, and a criterion matches if the
    * book value contains the search string.
    * Especially by searching the author a book match if it has a single
    * author, whose first and last name contains the values searched.
    * The search values are:
    *   isbn The book's ISBN, which matches the book if it is null,
blank
    *          or the book's ISBN contains the value
    *   title The book's title, which matches the book if it is null,
blank
    *          or book title contains the value
    *   year The book's publication, matching the book if 0 or release
    *          year is equal to the value
    *   edition The book's edition that matches the book if 0 or the
    *          book's edition is equal to the value
    *   publisher The name of the publisher, which matches the book if
it
    *          is null, blank or publisher's name contains the value
    *   firstname Matches if null, blank or the book has an author whose
    *          first name contains the value
    *   lastname Matches if null, blank or the book has an author whose
    *          last name contains the value
    * Returns all books that matches the search values
    */
  public List<IBook> Find(string isbn, string title, int year, int
edition,
    string publisher, string firstname, string lastname);
}
```

Write a class *Booklist* that implements the interface. When you have written the class, you must writes a test method. You can define the following array in the main program, which contains data for 10 books:

```
private static string[][] data = new string[10][];

static Program()
{
  data[0] = new string[] { "978-1-59059-855-9", "Beginning Fedora From
  Novice to Professional", "2007", "1", "519", "Apress", "Shashank",
  "Sharma", "Keir", "Thomas" };
  data[1] = new string[] { "978-0-13-275727-0", "A practical guide to
  Fedora and Red Hat Enterprise Linux", "2011", "6", "519", "Prentice
  Hall", "Mark G.", "Sobell" };
  data[2] = new string[] { "978-1-4842-0067-4", "Beginning Fedora
  Desktop: Fedora 20 Edition", "2014", "1", "459", "Apress",
  "Richard", "Petersen" };
  data[3] = new string[] { "978-0-470-48504-0", "Fedora 11 and Red Hat
  Enterprise Linux Bible", "2011", "1", "1076", "Wiley Publishing",
  "Christopher", "Negus", "Eric", "Foster-Johnson" };
  data[4] = new string[] { "978-1-118-99987-5", "Linux Bible", "2015",
  "9", "859", "John Wiley & Sons", "Christopher", "Negus" };
  data[5] = new string[] { "0-534-95054-X", "Understanding Data
  Coomunications & Networks", "1999", "2", "711", "Cole Publishing",
  "William A.", "Shay" };
  data[6] = new string[] { "978-0-13-255317-9", "Computer Networks",
  "2011", "5", "952", "Prentice Hall", "Andrew S.", "Tanenbaum",
  "David J.", "Wetherall" };
  data[7] = new string[] { "0-13-148521-0", "Structured Computer
  Organization", "2006", "5", "777", "Prentice Hall", "Andrew S.",
  "Tanenbaum" };
  data[8] = new string[] { "978-0-321-54622-7", "Data Structures &
  Problem Solving Using Java", "2010", "4", "1011", "Addison-Wesley",
  "Mark Allen", "Weiss" };
  data[9] = new string[] { "978-0-321-63700-0", "LINQ To Objects Using
  C# 4.0", "2010", "1", "312", "Addison-Wesley", "Troy", "Magennis" };
}
```

Note how the array is defined in a ragged array and is initialized in a static constructor.

Next, write a test method on the basis of these data that creates a book list with 10 books and print books whose title contains the word *Fedora*.

4.5 FACTORIES

The program *Students3* now consists of classes where the coupling between the classes are defined by interfaces. None of the classes know about the other classes existence, including

how these classes are implemented, but they know what you can do with objects of the specific classes, since they know the contracts. Software consists of modules (which here can be translated into classes), and it is a goal to write software that consists of as loosely coupled modules as possible. To program to an interface is an important step in that direction, but somewhere, the specific objects must be created, and in the above test program it is done in the particular helper methods. However, it is also the only place where to references the concrete classes.

You can move the code that creates the objects to a special class, which is called a *factory* class (a class which produces objects). I've added the following class to the program:

```
public abstract class Factory
{
  public static ISubject CreateSubject(string id, string name, int
  ects = 0)
  {
    return new Subject(id, name, ects);
  }

  public static ICourse CreateCourse(int year, ISubject subject)
  {
    return new Course(year, subject);
  }

  public static ICourse CreateCourse(int year, string id, string name,
  int ects, int score)
  {
    ICourse course = new Course(year, CreateSubject(id, name, ects));
    course.Score = score;
    return course;
  }

  public static IStudent CreateStudent(string mail, string name,
  params ICourse[] course)
  {
    return new Student(mail, name, course);
  }

  public static IStudents CreateStudents(string name, params
  IStudent[] studs)
  {
    IStudents students = new Students(name);
    students.Add(studs);
    return students;
  }
}
```

The class consists only of static methods that creates objects of a concrete type, but returns the objects as interface types. The class is defined abstract, and this means that the class cannot be instantiated.

With the class *Factory* available, the test program can be written as follows, where I have not shown the print methods, as they are not changed:

```
static void Test4()
{
  try
  {
    Institution.Instance.Add(
      Factory.CreateStudents("Team-A 2015",
        Factory.CreateStudent("olga.jensen@mail.dk", "Olga Jensen",
          Factory.CreateCourse(2015, "PRG", "Introduction to Java", 5, 7),
          Factory.CreateCourse(2015, "OPS", "Operating Systems", 15, 2),
          Factory.CreateCourse(2015, "SYS", "System Development", 5, 2)
        ),
        Factory.CreateStudent("harald.andersen@mail.dk", "Harald
        Andersen",
          Factory.CreateCourse(2015, "PRG", "Introduction to Java", 5, 4),
          Factory.CreateCourse(2015, "NET", "Computer Networks", 10, 12)
        )
      ),
      Factory.CreateStudents("Team-B 2015",
        Factory.CreateStudent("svend.hansen@mail.dk", "Svend Hansen",
          Factory.CreateCourse(2015, "OPS", "Operating Systems", 15, 10),
          Factory.CreateCourse(2015, "RRG", "Introduction to Java", 5, 0),
          Factory.CreateCourse(2015, "DBS", "Database Systems", 20, 7)
        ),
        Factory.CreateStudent("svend.frederiksen@mail.dk", "Svend
        Frederiksen",
          Factory.CreateCourse(2015, "OPS", "Operationg Systems", 15, 2),
          Factory.CreateCourse(2015, "NETL", "Computer Networks", 10, 10)
        )
      )
    );
    Print();
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Then there are no coupling between the program and the concrete classes (all the create methods in the class *Program* are not needed). In this case, the class *Factory* is trivial with simple methods that do nothing but to encapsulate a *new* operation, but other classes may be more complex, for example, to read data from a file or database.

The name of the above class is *Factory*, which may be an unfortunate name as it usually refers to one of two design patterns called *Abstract Factory* or *Factory Method*. There are reasons to mention that the above class does not implement one of these patterns, but looks a bit like the *Factory Method* pattern.

EXERCISE 6: A FACTORY

Create a new project called *Library5*, and add all classes and interfaces (except *Program*) from the *Library4* project to this project. Remember to rename the types namespace. You should then create the following factory class:

```
public static class Factory
{
  public static IPublisher CreatePublisher(int nummer, string name) {
  ... }

  public static IAuthor CreateAuthor(string firstname, string lastname)
  { ... }

  public static IBook CreateBook(params string[] elem) { ... }

  public static IBooklist CreateBooklist() { ... }
}
```

Here are three of the methods trivial, but the method that creates a book is relatively complex. You can get the most of the required code from the class *Booklist*.

After writing the class, change the main program, so all objects in the class *Program* are created by calls to the *Factory* class. You must also change the class *Booklist*, so that it also applies the *Factory* class.

5 INHERITANCE

If you have a class and you want another class, similar to the first, but expands with new variables or methods, or you may want one or more methods to work in a different way, the new class can inherit the first. As explained in C# 1, the class that inherits, is called for a derived class, while the class it is inherited from is called the base class. Other words for the same are respectively sub-class and super-class. I will in this chapter illustrate inheritance through classes, which represent loans in a bank. To make it simple, I will assume that a loan is characterized by the formation expenses and an amount that I together will call the loan's principal, an interest rate which I would assume is constant throughout the loan period, and a number of years and a number of periods a year, and the product of these two numbers are the number of payments to repay the loan. It thus corresponds to the same requirements as I assumed in the loan calculation program in the book C# 2. Under these conditions a loan is defined as the following class:

```
public class Loan
{
  private double cost;          // the formation cost
  private double amount;        // the amount for the loan
  private double interestRate;  // interest rate as a number between 0
                                   and 1
  private int years;            // the number of years
  private int terms;            // the number of periods a year

  public Loan(double cost, double amount, double interestRate, int
  years, int terms)
  {
    this.cost = cost;
    this.amount = amount;
    this.interestRate = interestRate;
    this.years = years;
    this.terms = terms;
  }

public double Cost
  {
    get { return cost; }
  }

  public double Amount
  {
    get { return amount; }
  }
```

```
  public double Principal
  {
    get { return cost + amount; }
  }

  public double InterestRate
  {
    get { return interestRate; }
  }

  public int Years
  {
    get { return years; }
  }

  public int Terms
  {
    get { return terms; }
  }

  public int Periods
  {
    get { return years * terms; }
  }

  public virtual double Repayment(int n)
  {
    return 0;
  }

  public virtual double Interest(int n)
  {
    return 0;
  }

  public virtual double Payment(int n)
  {
    return 0;
  }

  public virtual double Outstanding(int n)
  {
    return 0;
  }
}
```

The seven properties should be self-explanatory, and the last four methods returns

1. repaid by the payment of the *n*th payment
2. interest by the payment of the *n*th payment
3. the *n*th payment
4. the outstanding debt immediately after the payment of the *n*th payment

If you look at the last four methods, they are all trivial and returns all 0. A loan can be repaid in several ways, and to write the code for the last four methods, it requires that you know what it is for a kind of loan in question. The class *Loan* do not know that, and therefore it is not able to implement these methods in a meaningful way. It may on the other hand be possible in a derived class. You should note that all four methods are defined *virtual* which means the methods can be overwritten in a derived class and then be implemented in another way. Note that it is the programmer of the class *Loan* that with *virtual* opens up this opportunity.

The most common loan is an annuity that is characterized by the fact that you for each payment pays the same every time. When you all the time have to pay interest on what is outstanding, it means that the relationship between the repayment and the interest are changed throughout the repaid period. If b is the size of the loan (the loan principal), r is the interest rate each period and n is the number of periods, the payment can be determined using the following formula:

$$y = \frac{br}{1-(1+r)^{-n}}$$

The outstanding debt after you have paid the kth payment can be calculated using the formula

$$rest = b(1+r)^k - y\frac{(1+r)^k - 1}{r}$$

So it is the kind of loan that I looked at in C# 2.

I will now write a class that represents an annuity when the class will inherit the class *Loan:*

```
public class Annuity : Loan
{
  public Annuity(double cost, double amount, double interestRate, int
  years, int terms) : base(cost, amount, interestRate, years, terms)
  {
  }

  public override double Payment(int n)
  {
    return Principal * InterestRate / (1 - Math.Pow(1 + InterestRate,
    -Periods));
  }

  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);
  }
}
```

You should note that the class inherits *Loan*, and how to specify that with same syntax as implementing an interface. The class consists only of a constructor and the four methods which should work in a different way. We say that the class overrides the four methods from the base class, and note how you specify that with the word *override*. You can do that because the methods in the base class are defined *virtual*. That the class inherits means it beyond what it itself defines also can use *public* variables and methods from the base class. Because a derived class can only refer to what in the base class is defined as *public*, it can therefore not reference the base class's variables, but it must use the *get*-properties.

When creating an *Annuity* object, the instance variables must be initialized, but they are in the base class, and are initialized using the base class's constructor. It is therefore necessary in one way or another to get this constructor performed. It must be called from

the constructor of the class *Annuity*, but since you cannot directly call a constructor, it is necessary with a special syntax:

```
: base(cost, amount, interestRate, years, terms);
```

after the declaration of the constructor, that calls the base class's constructor.

The biggest challenge in writing the class *Annuity* is to implement the above formulas. Here you must notice the method *Math.Pow()*, which calculates the power of an argument. With this method available, it is quite simple to write the code to the four calculation methods.

If you have an object of the type *Annuity*:

```
Annuity loan = new Annuity(1000, 10000, 0.015, 5, 2);
```

then you can write a statement like the following:

```
Console.WriteLine(loan.Repayment(5));
```

and the method *Repayment()* in the class *Annuity* is performed. You can also write

```
Console.WriteLine(loan.Principal);
```

as *loan* is an *Annuity* that inherits *Loan* and the property *Principal* is also available.

Sometimes it can be convenient directly to refer to the variables in the base class from a method in a derived class, but without making them *public*. This is possible if the variables are defined *protected*:

```
public class Loan
{
  protected double cost;           // the formation cost
  protected double amount;         // the amount for the loan
  protected double interestRate;   // interest rate as a number
                                      between 0 and 1
  protected int years;             // the number of years
  protected int terms;             // the number of periods a year
```

A protected variable can not be referenced outside the class, but it can be referenced from derived classes, even if they belong to a different namespace. As an example the class *Annuity* now can be written as follows:

```
public class Annuity : Loan
{
  public Annuity(double cost, double amount, double interestRate, int
  years, int terms) : base(cost, amount, interestRate, years, terms)
  {
  }

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

  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 context, there are no big differences, but in other situations it may be necessary to open up the protection, so derived classes can reference variables in the base class.

Note that the methods also can be *protected*.

An amortization is a table showing a summary of the payments of a loan and for each payment shows the payment, repayment, interest and outstanding debt. The following class represents an amortization:

```
public class Amortization
{
  private Loan loan;

  public Amortization(Loan loan)
  {
    this.loan = loan;
  }

  public void Print()
  {
    Console.WriteLine(string.Format("Principal: {0, 12:F2})", loan.
    Principal));
    ...
  }
}
```

There is not much to explain, but note that the parameter to the constructor has the type *Loan* and the class *Amortization* thus is not aware of the specific loan types such as *Annuity*. I will return to that shortly.

With the class *Amortization* available, you can perform the following program:

```
static void Main(string[] args)
{
  new Amortization(new Annuity(0, 10000, 0.015, 10, 2)).Print();
}
```

and the result is as shown below:

```
Principal:                      10000,00
Interest rate:                      1,50 %
Number of periods:                    10
```

Periods	Payment	Repayment	Interest	Outstanding
1	1084,34	934,34	150,00	9065,66
2	1084,34	948,36	135,98	8117,30
3	1084,34	962,58	121,76	7154,72
4	1084,34	977,02	107,32	6177,70
5	1084,34	991,68	92,67	5186,02
6	1084,34	1006,55	77,79	4179,47
7	1084,34	1021,65	62,69	3157,82
8	1084,34	1036,97	47,37	2120,85
9	1084,34	1052,53	31,81	1068,32
10	1084,34	1068,32	16,02	0,00

In addition to the result, note that it is the methods in the class *Annuity* that are performed and it is not obvious. The class *Amortization* only know the type of *Loan* that have trivial methods, all of which returns 0. Nevertheless, it is the methods of the class *Annuity* that are executed, and it is, even if the class *Amortization* only know the type *Loan.* When such it is the run-time system that remembers that the object *loan* actually has the type *Annuity* and uses therefore the object with the right methods. It is one of the key concepts of object-oriented programming and is called *polymorphism*.

If you considers the class *Loan*, then, as mentioned above the four calculation methods are trivial and all simply returns 0. These methods are somehow meaningless, and it makes no sense to create objects whose type is *Loan*. If you did that, and sent such an object over to an *Amortization* object, you would get an amortization table, where all numbers are 0. The problem can be solved by making the four methods *abstract*:

```
public abstract class Loan
{
  private double cost;          // the formation cost
  private double amount;        // the amount for the loan
  private double interestRate;  // interest rate as a number between 0
                                   and 1
  private int years;            // the number of years
  private int terms;            // the number of periods a year

  public Loan(double cost, double amount, double interestRate, int
  years, int terms)
  {
    this.cost = cost;
    this.amount = amount;
    this.interestRate = interestRate;
    this.years = years;
    this.terms = terms;
  }

  public double Cost
  {
    get { return cost; }
  }

  public double Amount
  {
    get { return amount; }
  }

  public double Principal
  {
    get { return cost + amount; }
  }

  public double InterestRate
  {
    get { return interestRate; }
  }
```

```
  public int Years
  {
    get { return years; }
  }

  public int Terms
  {
    get { return terms; }
  }

  public int Periods
  {
    get { return years * terms; }
  }

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

An *abstract* method is just a prototype or a signature for a method in the same way as you defines methods in an interface. When a class contains abstract methods, the class is abstract, and it must be declared as an abstract class:

```
public abstract class Loan
```

An abstract class cannot be instantiated, and you cannot with *new* create an object whose type is *Loan* but an abstract class can easily be parameter to a method. That class *Loan* has four abstract methods means that anyone who knows anything that is a *Loan* knows that the object in question, has the four methods that the abstract class specify.

If the class *Loan* is changed to an abstract class as shown above, the program can still be translated and run, and it gives the same result.

One way to think of abstract classes is that you have to write a class *Loan* and know what properties and methods a *Loan* must have, but some of the methods you are not able to implement, because you do not have sufficient knowledge. These methods can then be

defined abstract and you can postpone the implementation to the specific derived classes, who have the necessary knowledge.

I will write another class that will represent a mix loan. It is a loan, where in the first periods, there is no repayment, and where you must pay interest only. After the periods without repayment the loan is an annuity, but with a fewer periods. The class may, for example be written as follows:

```
public class MixLoan : Loan
{
  private int free;
  private Loan loan;

  public MixLoan(double cost, double amount, double interestRate, int
  years, int terms, int free) : base(cost, amount, interestRate,
  years, terms)
  {
    this.free = free;
    loan = new Annuity(cost, amount, interestRate, years - free, terms);
  }

  public int Free
  {
    get { return free * Terms; }
  }

  public override double Payment(int n)
  {
    return n <= Free ? Principal * InterestRate : loan.Payment(n - Free);
  }

  public override double Interest(int n)
  {
    return n <= Free ? Principal * InterestRate : loan.Interest(n - Free);
  }

  public override double Repayment(int n)
  {
    return n <= Free ? 0 : loan.Repayment(n - Free);
  }

  public override double Outstanding(int n)
  {
    return n <= Free ? Principal : loan.Outstanding(n - Free);
  }
}
```

You should note that a derived class can add new variables (in this case two) and new methods (here it is a property *Free* that returns the number of periods without repayment). The implementation of the four abstract methods are simple and are not further explained. You can test the class in *Main()*:

```
static void Main(string[] args)
{
  new Amortization(new MixLoan(1000, 10000, 0.015, 10, 1, 3)).Print();
}
```

As another example of a loan, one can consider a serial loan that is a loan where the repayment is the same for every time. This means that the payment is greatest at the beginning and decreases through the loan period. Just as an *Annuity* the project has a class *Serial* that is a derived class that inherits *Loan* and thus is a concrete class. I will not show the code here.

The abstract class *Loan* can be defined by an interface:

```
public interface ILoan
{
  double Cost { get; }
  double Amount { get; }
  double Principal { get; }
  double InterestRate { get; }
  int Years { get; }
  int Terms { get; }
  int Periods { get; }
  double Repayment(int n);
  double Interest(int n);
  double Payment(int n);
  double Outstanding(int n);
}
```

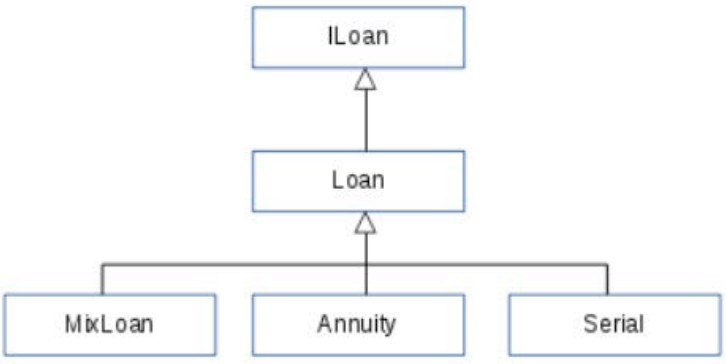
The class *Loan* must then implement the interface *ILoan*, but you must note that you cannot delete the four abstract methods in the class *Loan*, as a class that implements an interface must implements all methods which the interface defines, and the class must still be abstract because it does not implement all of the interface's methods:

```
public abstract class Loan : ILoan
{
  private double cost;          // the formation cost
  private double amount;        // the amount for the loan
  private double interestRate;  // interest rate as a number between 0
                                   and 1
  private int years;            // the number of years
  private int terms;            // the number of periods a year

  public Loan(double cost, double amount, double interestRate, int
  years, int terms)
  {
    this.cost = cost;
    this.amount = amount;
    this.interestRate = interestRate;
    this.years = years;
    this.terms = terms;
  }

  public double Cost
  {
    get { return cost; }
  }

  public double Amount
  {
    get { return amount; }
  }

  public double Principal
  {
    get { return cost + amount; }
  }

  public double InterestRate
  {
    get { return interestRate; }
  }
```

```
  public int Years
  {
    get { return years; }
  }

  public int Terms
  {
    get { return terms; }
  }

  public int Periods
  {
    get { return years * terms; }
  }

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

It is then possible to change the class *Amortization*, where the type of the variable *loan* is changed to *ILoan,* and the same for the parameter in the constructor.

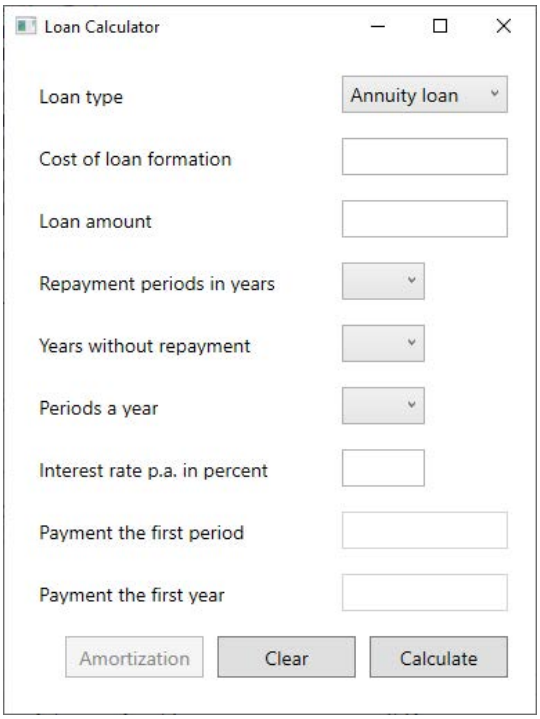
The kind of inheritance, as discussed above, is called for *linear inheritance*, which means that a class can only inherit one class, classes can have only one base class. On the other hand, a class can have all of the derived classes, as may be needed, and you can inherit in all the levels as you wish. Inheritance leads to a hierarchy of classes, and in this example the classes concerning loans has the following hierarchy:



ILoan is an interface, but it is included in the type hierarchy in the same manner as classes.

EXERCISE 7: A LOAN CALCULATOR

In problem 3 in the book C# 2 you have written a loan calculation program that calculates the payment of an annuity and shows an amortization table. In this exercise you must create an expansion of this program so that it also can perform loan calculations for a mix loan and a serial loan. The program must open the following window:



You must then expand the window from the previous version of the program with two new controls: A combo box to select the loan type, and a combo box to select the number of years without repayment. The last combo box should only be used for a mix loan.

Start by creating a copy of the project *LoanCalculator* from C# 2 and open the copy in Visual Studio. Modify the *MainWindow* as above. The project has a class *Loan*. Delete that class and add instead the following types from the above project:

- *ILoan*
- *Loan*
- *Annuity*
- *MixLoan*
- *Serial*

Then change the code to use these types. It is primarily the controller class that needs to be changed.

PROBLEM 1: NUMBER SEQUENCES

The concept of a number sequence is known from mathematics and is a sequence of numbers. One example is the sequence of even numbers:

0, 2, 4, 6, 8, 10, ...

where you constantly determines the next number by adding 2. In the same way one can consider the sequence of powers of 2 and the factorials which are

1, 2, 4, 8, 16, 32, 64, 128, ...

1, 1, 2, 6, 24, 120, 720, 5040, ...

At the powers of 2 you always go one step forward by multiplying by 2, while at the factorials you go a step forward by multiplying with the next positive integer. More precisely, a number sequence is a sequence of numbers

$$a_0, a_1, a_2, \ldots, a_n, \ldots$$

where there for any none negative integer is attached a (usually) real number. Taking the as example the sequence consisting of the powers of 2, you sometimes writes

$$\{2^n\}$$

In this sequence as example is $a_{10} = 1024$.

In this problem, you should write some classes that represent sequences, but only sequences whose values are integers.

Some sequences, like factorial, is growing very rapidly, and must sequences implemented in C#, is limited in how big integers you can represent. Since you here only works with sequences with integer values, the size is limited by the type *long*. If you again consider the factorials

$$20! = 2432902008176640000$$

and it is the largest factorial that can be represented by a *long*. C#, however, has a solution as the class *BigInteger*, which is a software implementation of an integer. In principle, the class represents arbitrary large integers, but it costs obviously something since calculations on very large numbers may be slow. Nevertheless, the class is worth knowing, so this example.

In the following you can think of a sequence as a sequence of numbers where each number is identified by an index, but only one of these numbers are visible and is the sequence's current value. Below I have illustrated the sequence consisting of powers of 2, and it is the number 256 with the index 8, that is visible:

Create a WPF project called *Sequences*. Add the following interface, where the comments explains what the methods should do:

```
namespace Sequences
{
  /*
   * Defines a number sequence with integer numbers where the first
     index is 0.
   */
  public interface ISequence
  {
    string Name { get; }
    uint Index { get; }
    BigInteger Value { get; }

    void Reset();

    /*
     * Returns the number in the sequence having the index n. At the
       same time,
     * this number is the actual value.
     */
    BigInteger this[uint n] { get; }

  /*
   * Steps the sequens one step forward
   */
  void Next();

  /*
   * Step the sequence one step back
   */
  void Prev();
```

```
  /*
   * Sets the sequence of the first value that is greater than or
equal
   * to number. This value is then the sequence's current value.
   */
  void Next(BigInteger number);

  /*
   * Sets the sequence of the first value that is less than or equal
to
   * number. This value is then the sequence's current value.
   */
  void Prev(BigInteger number);
  }
}
```

The class *BigInteger* is defined in an assembly *System.Numerics* and it is required in Solution Explorer to set a reference to this assembly. You must also have a *using* statement

```
using System.Numerics;
```

in all source files where you uses the class.

A sequence is characterized by three properties, an indexer and five methods, and some of these methods does not depend on a specific sequence. You should then write an abstract class *Sequence*, which implements everything that is common to all sequences. This class will then be a common base class for concrete sequences.

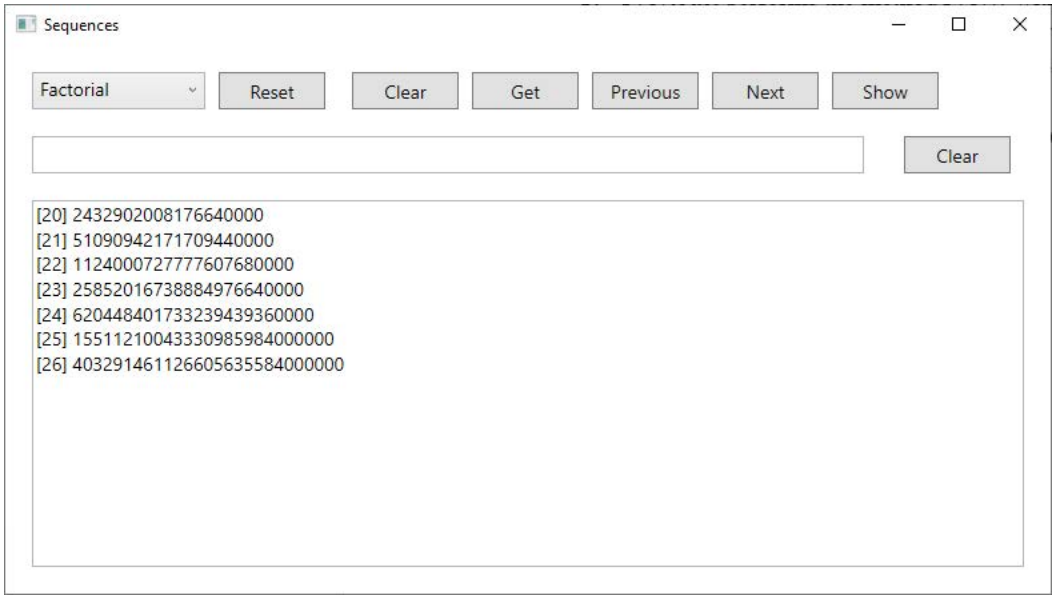
As a next step you must implement the sequence consisting of the factorials. The class is a concrete sequence and should be implemented as a class that inherits *Sequence*.

Once you have written the class, you should write the window:

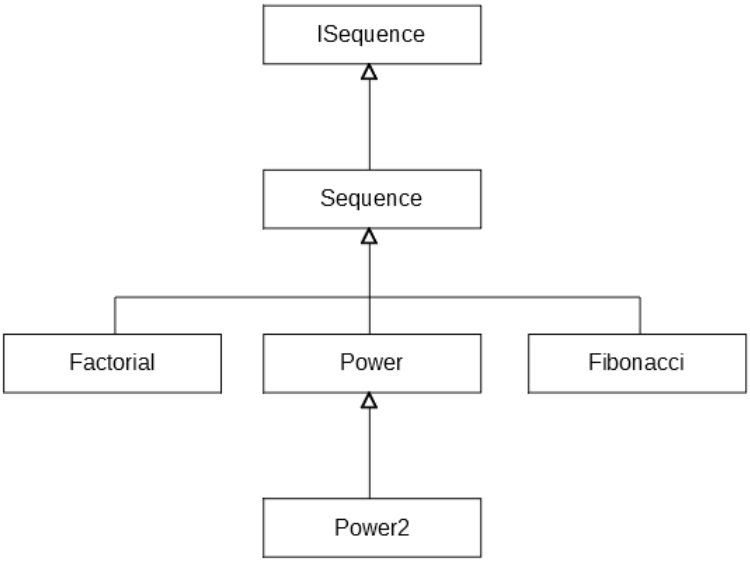
The upper *Clear* button must clear the field for the result (the lower field which is a multi-line *TextBox*). The other *Clear* button should only clear the upper *TextBox* which is used to enter an argument for the methods that need an argument. The other five buttons should be used for

1. *Reset*, reset the current sequence
2. *Get*, set the sequence's index to the argument, that is call the indexer
3. *Previous*, performs the method *Prev()* with or without an argument
4. *Next*, performs the method *Next()* with or without an argument
5. *Show*, show the current value of the sequence

Below is shown the window after I have entered 20 as argument, clicked on *Get* and clicked *Next* a few times:

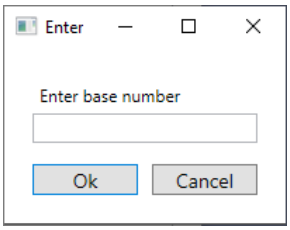


Next, you must write three new concrete sequences so the result is the following class hierarchy:



1. *Power*, is the power sequence $\{a^n\}$ and the constructor must have a parameter of the type *unit* for the base number. If this parameter is called *t*, the sequence represents the numbers $1, t, t^2, t^3, t^4, t^5, \ldots$
2. *Power2*, which is the same as the above, just where the base number is 2.
3. *Fibonacci*, that represents the Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

It should be possible to select one of the four sequences in the combo box. If you select a sequence the program must create a sequence of that type, and it is then the current sequence. If you select the sequence *Power* it must be possible to enter the base number. One solution is to add a simple dialog box for that purpose:



5.1 CONSIDERATIONS ABOUT INHERITANCE

I will conclude all what is said about interfaces and inheritance with a few remarks. Seen from the programmer consists a program of classes that define the objects used by the application

to perform its work. For the sake of maintenance of the program you are interested that these classes is as loosely coupled as possible, and that is, that the classes should know so little about each other as possible. In that way you can change the classes without it affects the rest of the program. You should therefore strive to write the classes, so a change only affects the class itself and thus only has local significance.

Now one cannot write programs without there are couplings between the classes, the meaning of a class is precisely that it must make services available for others to use, but you can ensure that the coupling only takes place through public methods. That's why I sometimes have talked about data encapsulation where instance variables are kept private, and thus it becomes the programmer who through the class's public methods decides which couplings to be possible. This principle can be further supported through an interface if all the classes are defined by an interface. An interface defines through abstract methods (methods in an interface are by default *public abstract*), which couplings should be possible, and adhere to it and always define references to objects using a defining interface, you get a looser coupling of the program's components as an object alone is known as an interface and the class that implements the interface is not known and may be changed without it affects the rest of the program. Therefore, it is a design principle that you should program to an interface.

To program to an interface is a good design principle, but everything can be overstated. A program can easily consist of several hundred classes, and if each (or most) of these classes defines an interface, the number of types will be huge and may lead to code that is difficult to grasp. One should therefore consider, when a class must define an interface and concerning primary classes, which represent key concepts in which there is a chance that there will be changes. A program will always consist of many classes that are using other classes that are local in nature, and these classes should not be defined with an interface. The upshot is that programming to an interface is an important principle and one of the best ways to secure loose couplings between parts of the program, but also that, while developing, consider whether an interface makes sense.

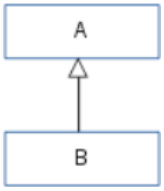
Another use of interfaces is, that using interfaces at design time you can define different contracts that the program's objects must comply. As a class can implement multiple interfaces (all of them you need), you can create objects that meet multiple contracts, and as some of the most important thing you can test whether an object meets one or the other contract. With reference to the first book, an object can be defined *IComparable*, and classes that implements this interface, instantiates objects that can be arranged in order and hence, for example be sorted. C# defines a number of such interfaces, which defines one or another property that an object may have. You will later in the book C# 4 about collection classes see classes that implements many interfaces. An interface is a concept that defines contracts which objects must comply.

If you look at the interfaces defined in this book, they reflect very closely the classes that implements these interfaces. It does absolutely not necessarily be the case, and many interfaces define a contract in form of a few and perhaps only a single method and as an example *IComparable*.

With regard of inheritance one can say that there are two uses. In one situation you have some classes which are similar, such that they somewhat are comprised of the same variables and methods. In such a situation you can take what the classes have in common, and move it into a common base class, that the others inherits. That way, you avoid the same code can be found in several places and thus have a program that takes up less space, but the important thing is that if you have to change the code, you should only change it at one place, and you are sure that you do not forget to change somewhere. The second situation is that you have defined a concept in the form of a class, and then you need a different concept, similar to the first but might need an extra variable or method, or there may be a method that should work in another way. In this case, the new concept can be defined as a class that inherits the first class. The two situations are really two sides of the same coin, but in the first case we speak of generalization, while in the second case, we talk about specialization. Whatever inheritance reflects the fact that a program consists of classes that have something in common and to model concepts of the same kind, and may be useful to define a hierarchy of classes that inherit each other.

At the design level the implementation of an interface also is a form of inheritance as an interface says that an object satisfies a particular contract. Because C# only supports linear inheritance, interfaces help during the design to work with multiple inheritance.

The most important of inheritance is not so much the expansion of existing code (although it is of course also important), but it is the relationship between super class and subclass. If you have a specialization

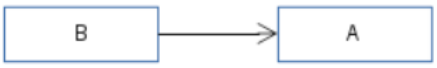


(and here it is not important, if A is a class or an interface), it is important that a *B* is also an *A* and a *B* anywhere can be treated in the same way as an *A*. It means that you can write a method and thus code which handles an *A* object:

```
type Method(A a)
{
}
```

and the method will work regardless of whether the parameter is a *A* object or a *B* object or another type that directly or indirectly inherits *A*. The method does not know the specific type, and it does not have to do it. It is called *polymorphism* and is one of the most important concepts in object-oriented programming and allows you to write program code that is very flexible and general.

Inheritance is one of the basic principles behind object-oriented programming and more than that, for it is a requirement that an object oriented programming language supports inheritance. However, it is not the solution to everything, and there is also criticism of inheritance. Taking the above design where *B* inherits *A*. It reflects a strong connection and it especially if *A* defines *protected* members. There is a high probability that changes in *A* will have an impact on derived classes. Moreover, one can mention that the connection between *A* and *B* is very static and determined at compile-time. Sometimes one thinks, therefore, on a design as



where an object *B* knows an object *A* through a reference. Here the reference can be replaced at run-time, providing the opportunity to the object *B* that it can refer to another *A* object, and the result is a code that is more flexible. One speaks sometimes that *B* delegate tasks to the *A* object. The two design principles are not directly each other alternatives, but in the last years there has probably been briefed on using delegation in situations where you would previously use of inheritance.

PROBLEM 2: GEOMETRIC OBJECTS

In this task, you must write some classes that are organized in a hierarchy. The classes are simple and represents geometric objects such as triangles and squares, and to the extent that there is a need for mathematical formulas, it is simple formulas, which are known from primary school. The goal is to show many of the concepts on classes handled in this book.

Start by creating a Visual Studio Console Application project, you can call *Geometry*. The example handles as mentioned geometric shapes, but you should only be interested in a shape's perimeter and area. Correspondingly, a shape can be defined as follows:

```
interface IShape
{
  double Perimeter { get; }  // returns the circumference of the shape
  double Area { get; }       // return the area of the shape
}
```

and you need to add this interface for your project. You must then add an auxiliary type that should be called *Point* and represents a point in a plane coordinate system:

```
struct Point
{
  public const double ZERO = 1.0e-20; // represents zero

  public double x;
  public double y;

  public Point(double x, double y)
  {
    this.x = x;
    this.y = y;
  }

  public double GetLength(Point p)
  {
    return Math.Sqrt(Sqr(x - p.x) + Sqr(y - p.y));
  }

  public override string ToString()
  {
    return string.Format("(%.4f, %.4f)", x, y);
  }

  public override bool Equals(object obj)
  {
    if (obj == null) return false;
    if (obj is Point)
    {
      Point p = (Point)obj;
      return Math.Abs(x - p.x) <= ZERO && Math.Abs(y - p.y) <= ZERO;
    }
    return false;
  }
```

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

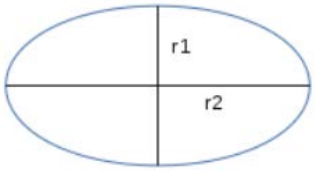
Regarding the method *GetLength()*, note that given two points (x_1, y_1) and (x_2, y_2) you determines the distance between the points as:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

Also note that the class has a static method *Sqr()* which returns the square of a number. In principle, such a method has nothing to do with a point and is also just an auxiliary method to implement *GetLength()*, but because it may be useful in other contexts, it is defined as a *static public* method.

Write an *abstract* class *Shape* that implements the interface in *IShape*. The class should only overrides the method *Equals()*, such two shapes are the same, if the objects has the same class, and if the two shapes have the same perimeter and area.

The next class will represent a concrete geometric shape. The class should be named *Ellipse* and must represents an ellipse, when an ellipse is solely defined by the two radii that are transferred as parameters to the constructor:



You can calculate the ellipse's perimeter and area on the basis of the following formulas:

$$areal = r_1 r_2 \pi$$

$$omkreds = 2\pi\sqrt{0.5(r_1^2 + r_2^2)}$$

With the ellipse in place, you can write a class *Circle*, representing a circle when a circle is simply an ellipse where the two radii are equal. The class *Circle* can be written as a class that inherits *Ellipse*.

As a next step, you should write classes to triangles. Start with a class *Polygon*, which can represent an arbitrary polygon. It can be represented by *Point* objects that are the polygon's vertices. The class should not validate the vertices and test whether they lead to irregular polygons, for example polygons where sides intersect. A polygon can be written as the following class:

```
abstract class Polygon : Shape
{
  protected Point[] p;

  public Polygon(params Point[] p)
  {
    this.p = p;
  }

  public override double Perimeter
  {
    get
    {
      double sum = p[p.Length - 1].GetLength(p[0]);
      for (int i = 1; i < p.Length; ++i) sum += p[i - 1].GetLength(p[i]);
      return sum;
    }
  }

  public override string ToString()
  {
    String text = "";
    foreach (Point q in p) text += " " + q;
    return text;
  }
}
```

Knowing the vertices, one can immediately implements the property *Perimeter* as the circumference can be determined as the sum of the lengths of the edges. However, you cannot immediately implement the property *Area*, and the class is therefore defined *abstract*.

Next, write a class *Triangle*, which must be a class that inherits *Polygon*. The class must have a constructor that has three points as parameters, and apart from the constructor the class alone has to override the *ToString()* and implement the property *Area*. Regarding the last you can determine the area of a triangle with sides a, b and c as follows:

$$areal = ½ab\sqrt{1 - (\frac{a^2 + b^2 - c^2}{2ab})^2}$$

In the interest of other classes the area calculation should be done by means of using a *public static* method that has as parameters has the triangle's vertices.

The class *Triangle* represents an arbitrary triangle defined by three points. One can also have more specialized classes for the triangles, such as *Equilateral* class representing an equilateral triangle, which should be a class that inherits the class *Triangle*. The class's constructor must have one parameter that is the side length (note that it is easy to define three points which represents the vertices in a triangle with a certain side length, start with the point (0, 0)). For performance reasons the class should override the property *Perimeter*, since it can be implemented more effectively if you know the side length. Write the class *Equilateral*.

Write similar to a class *RightAngled* that represents a right angled triangle.

As the last shapes you should write some classes to squares. Start with a class *GeneralSquare*, which inherits the class *Polygon*. The class must have a constructor, which parameters that are four *Point* objects. Furthermore the class must implements the property *Area*, which is abstract in the base class. There is no general formula to determine the area of a general square, but you draw can a diagonal that divides the square into two triangles (because I ignore the problem that a square may be concave). One can therefore implement the property *Area* by determining the area of the two triangles.

You must then write a class *Rectangle*, which inherits the class *GeneralSquare* when the class's constructor has two parameters that respectively are the width and height. Also, write a class *Square* when a square just is a rectangle where the sides are of equal length.

You've written some classes that represents geometric objects, and you must now define a composite shape, which consists of other shapes. The following interface inherits *IShape* and defines a composite shape called *IDrawing*:

```
interface IDrawing : IShape
{
  void Add(params IShape[] shape);
}
```

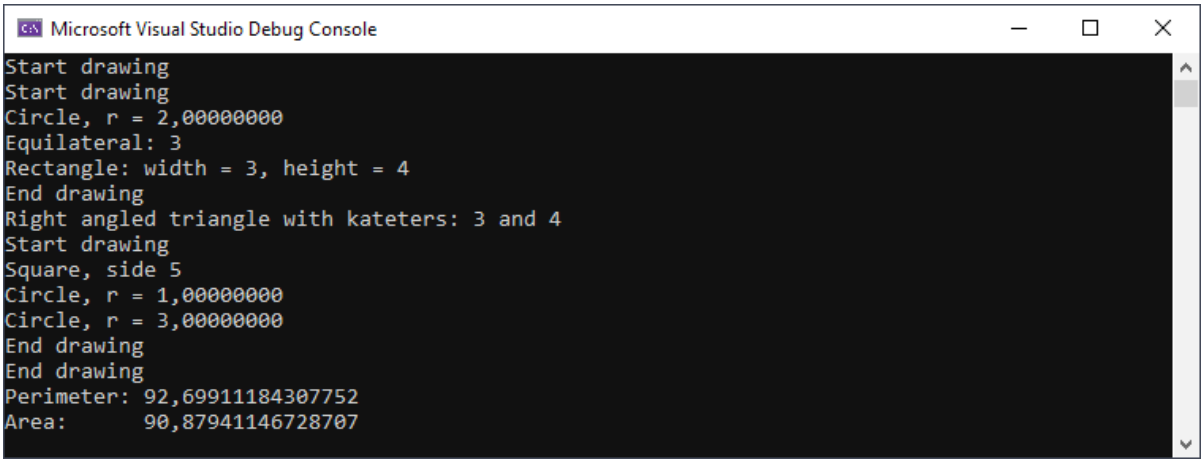
Write a class *Drawing* that implements this interface when the circumference of a drawing is defined as the sum of the perimeters of all the drawing's shapes and the same for the area.

A drawing is therefore especially a *IShape*, and thus a drawing contains other drawings. Below is a test program that creates a drawing that consists of

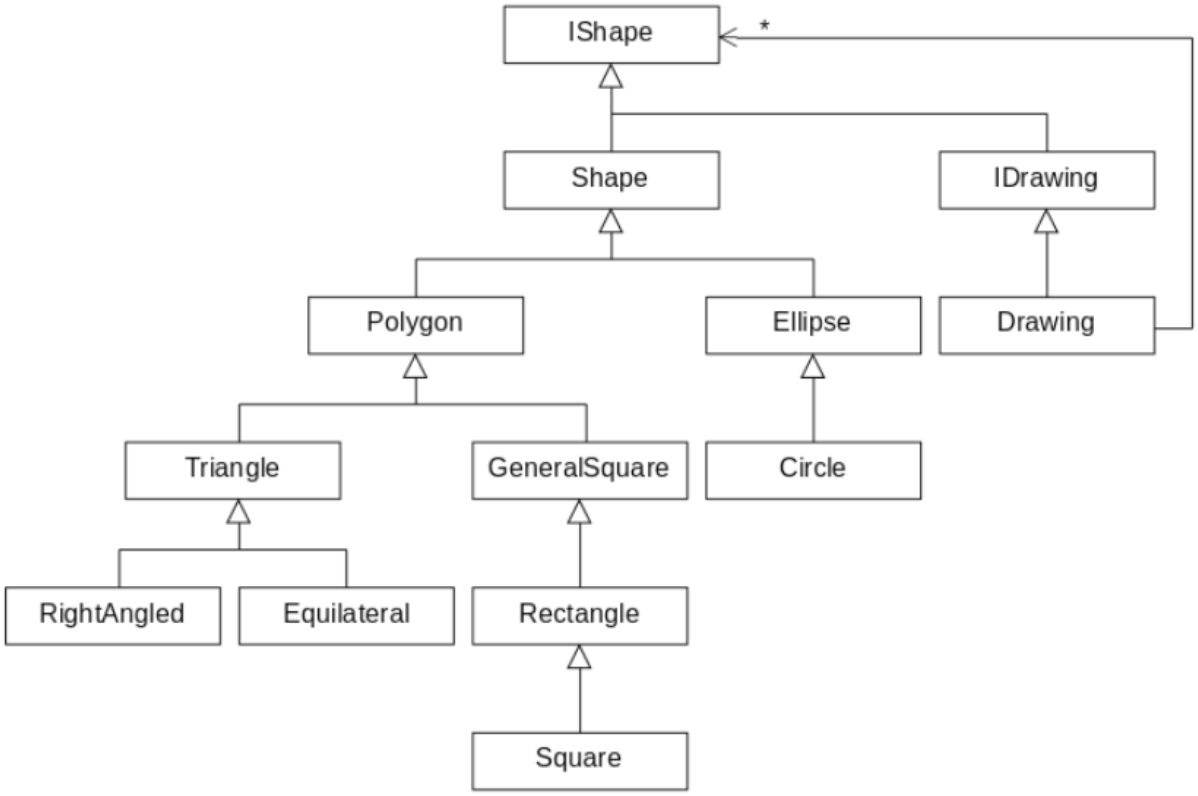
1. a drawing that contains a circle, a equilateral and a rectangle
2. a right angled triangle
3. a drawing that contains a square and two circles

```
class Program
{
  static void Main(string[] args)
  {
    Print(Create(Create(new Circle(2), new Equilateral(3), new
    Rectangle(3, 4)), new RightAngled(3, 4), Create(new Square(5), new
    Circle(1), new Circle(3))));
  }

  private static void Print(IShape shape)
  {
    Console.WriteLine(shape);
    Console.WriteLine("Perimeter: " + shape.Perimeter);
    Console.WriteLine("Area: " + shape.Area);
  }

  private static IShape Create(params IShape[] shapes)
  {
    IDrawing d = new Drawing();
    d.Add(shapes);
    return d;
  }
}
```

Test the program. The result should be something like the following:



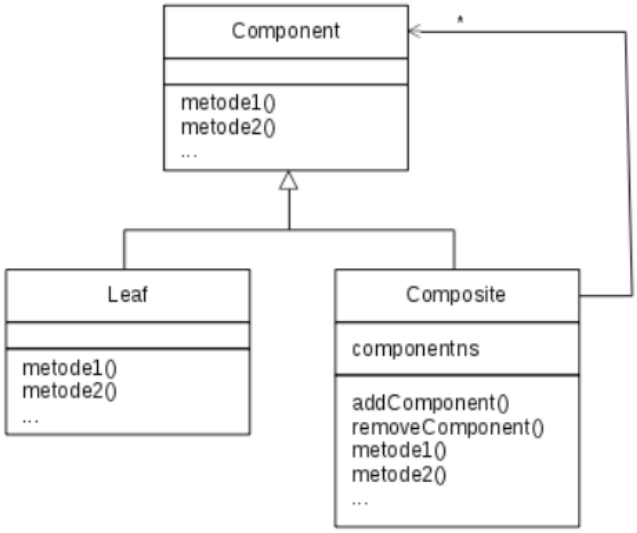
The types in this task is organized in a class hierarchy and can be illustrated as shown below. It's simple to expand the class hierarchy with new classes, including classes for new shapes, and as long as these classes complies with the contract *IShape*, the class *Drawing* can use these classes as it knows nothing about the concrete classes.



5.2 THE COMPOSITE PATTERN

In fact it is quite often that, in practice, you meat a situation similar to above and think for example on a part list where you have a product that is composed of sub-components, or think of a menu where a menu consisting of menu items or other menus. Another example is a file-system consisting of directories and files. Therefore, it is natural to express the situation in a design pattern which is called a *composite pattern*.

The problem is thus to manipulate a hierarchy of objects, where the bottom of the hierarchy have some concrete objects that we call the leaf objects, and there are some other objects that we call composite objects consisting of leaf objects and other composite objects. Both leaf objects and composite objects have a common base class (and possibly it is an interface or an abstract class), and the idea is that composite objects handles their child objects alike, whether in the case of leaf or composite objects. The solution is illustrated with the following figure:



and the solution is referred as a *composite pattern*.

In practice, there may be different flavors, and often there may be several kinds of leaf objects that all are part of a hierarchy. This is the case in the above example with geometric shapes. Here, the *Component* class corresponds to the interface *IShape*, and the class *Leaf* corresponding to *Shape*. In the figure corresponds the class *Composite* to *IDrawing*. In principle, it is a very simple pattern, but in practice there are still some considerations on how to implement the pattern. The goal is as mentioned that the *Leaf* and the *Composite* objects must have the same behavior, and therefore the methods *addComponent()* and *removeComponent()* is defined in *Component*. It provides, however, a problem since they do not make sense for *Leaf* objects. Where necessary, they must in *Leaf* classes be implemented as methods that do not perform anything and possible raises an exception. If you instead uses a design as shown above, it then is necessary to test whether specific objects are *Leaf* or *Composite* objects.

6 THE CLASS OBJECT

As mentioned C# has a class called *System.Object* (also known by the compiler as *object*), and all other classes inherits directly or indirectly this class. If you, as an example looking at the class *Loan*

```
public abstract class Loan : ILoan
```

the class automatically inherits the class *Object*. It means that you could have written

```
public abstract class Loan : Object, ILoan
```

and indeed it is allowed to write, but unnecessary. This means several things, including as perhaps the most important that all classes in C# are linked in a large hierarchy with the class *Object* as the root. All classes without exception is an *Object* and inherits the methods that an *Object* has.

A class can inherits another class, but only one class. The class can also implements interfaces and all it want, but if a class both inherits a class and implements one or more interfaces, the class must be written first after the colon.

The class *Object* defines a few basic methods that all has a trivial implementation, and it is then up to the individual classes to override the methods so they works in a reasonable way. The class has no *public* fields, but it has two static methods and six instance methods where four are defined virtual and as so can be overwritten in the derived classes:

1. *public virtual bool Equals(object obj)*
2. *protected virtual void Finalize()*
3. *public virtual int GetHashCode()*
4. *public virtual string ToString()*
5. *public Type GetType()*
6. *protected object MemberwiseClone()*
7. *public static bool Equals(object objA, object objB)*
8. *public static bool ReferenceEquals(object objA, object objB)*

Equals() has one argument which is an *object*, and the method tests where this object is equal to the current object. The default implementation compare object references and as so returns *true* if an object is compared with itself. Typical this method is overwritten to compare the objects state, so it is the objects values which are compared. You should note that if you override *Equals()* you get a warning to also override *GetHashCode()* as some of the collections classes requires that both methods are overwritten.

Finalize() is a method which is called from the garbage collector before an object is removed. You rarely want to override this method, but the point is that the method can perform some sort of cleanup before an object is deleted on the heap.

GetHashCode() is a method which returns an *int* called a hash code. The default implementation cannot and should not be used for anything, but the idea is that the method must be overridden in derived classes so that objects can be identified by an integer. In the next book I will return to the method associated with collection classes, but the protocol is that if two objects are equals they must also have the same hash code.

ToString() has been discussed earlier as a method that represents an object as a *string*. The default implementation returns the full name of the type and the meaning is that you in your own classes should override the method to return a value representing the object's state.

GetType() is a none virtual method that returns a *Type* object which is a complete description of the type for current object. This means, among other things, that you for a specific object can be told from type it is instantiated.

MemberwiseClone() is a method that returns a copy of an object. I will discuss this method later.

Equals() as a static method is a method with two objects as parameters, and the method tests if these objects are equal.

ReferenceEquals() works in the same way, but is used to tests if two objects references to the same object on the stack.

The project *ObjectProgram* is a simple test program to test the action of some of the above methods. The program defines a very simple interface which defines a single property:

```
interface Person
{
  string Name { get; }
}
```

It also defines a class which implements this interface:

```
class Person1 : Person
{
  private string name;

  public Person1(string name)
  {
    this.name = name;
  }

  public string Name
  {
    get { return name; }
  }
}
```

You should note that the class implicit inherits the class *Object* and then the methods defined in *Object*, but the class *Person1* does not override some of the methods. By contrast, so does the class *Person2*:

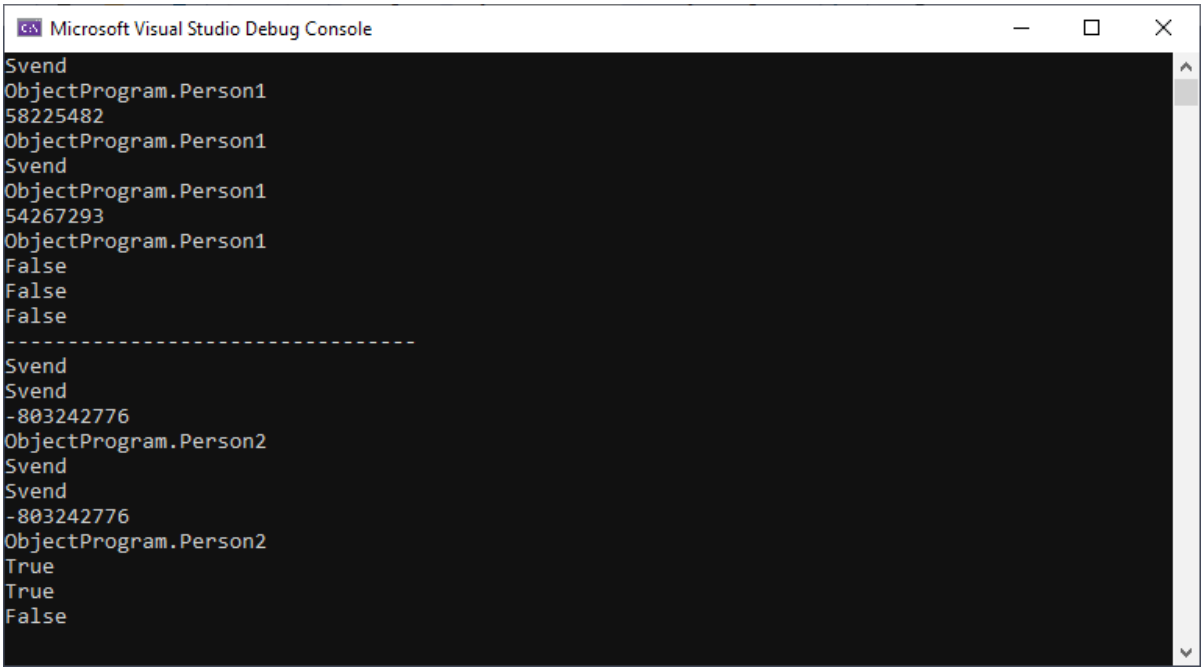
```
class Person2 : Person
{
  private string name;
  public Person2(string name)
  {
    this.name = name;
  }

  public string Name
  {
    get { return name; }
  }

  public override bool Equals(object obj)
  {
    if (obj is Person2) return ((Person2)obj).name.Equals(name);
    return false;
  }

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

```
  public override string ToString()
  {
    return name;
  }
}
```

The class overrides *ToString()* so it returns the *name* and then the state of an object. *Equals()* tests if the argument is a *Person2* and if so it compare the two objects *name* property and thus compare the state of objects. Also note the override of *GetHashCode()* that simple returns the *GetHashCode()* of the *name* property. This is fine as the class *String* overrides *GetHashCode()*. If you run the test program:

```
class Program
{
  static void Main(string[] args)
  {
    Test(new Person1("Svend"), new Person1("Svend"));
    Console.WriteLine("---------------------------------");
    Test(new Person2("Svend"), new Person2("Svend"));
  }

  static void Test(Person pers1, Person pers2)
  {
    Print(pers1);
    Print(pers2);
    Console.WriteLine(pers1.Equals(pers2));
    Console.WriteLine(object.Equals(pers1, pers2));
    Console.WriteLine(object.ReferenceEquals(pers1, pers2));
  }

  static void Print(Person pers)
  {
    Console.WriteLine(pers.Name);
    Console.WriteLine(pers);
    Console.WriteLine(pers.GetHashCode());
    Console.WriteLine(pers.GetType());
  }
}
```

you get the following result:



You are encouraged to study the result and compare it with the code, so you are sure that you understand the difference when using the method *Test1()* is performed with *Person1* objects and *Person2* objects.

7 EXCEPTION HANDLING

When writing a program, there may be errors. For example you can write the program code incorrectly, so the program can't be translated. This is because the program is written with an incorrect syntax, and that kind of mistake is rarely unproblematic, as they are caught by the compiler. One speaks therefore also of a compiler time errors. These errors must obviously be addressed, which, until you get trained, can be difficult enough, but when I call them unproblematic, it is because the compiler can find them, and the program can't execute before they are corrected.

Another type of errors are logical errors where the program can be translated and run, but when it does something else than the idea was. It may, for example be a calculation that gives a wrong result or an incorrect value that is stored in a database. It's the hardest errors because they can't be caught by the compiler, and because the program can actually run for a period before the error is acknowledged, but also because it may be errors that are difficult to locate. The user of the program detects the symptom, but it can be hard to find where in the code it is that it goes wrong. The remedy for this kind of errors is test, test and test again. To test a program is by no means simple, and it requires both time and procedures.

A third kind of error is caused by environmental conditions and are errors you as a programmer can't really guard against, but conversely also errors that you in one way or another must take care of. As an example, it may be a user who enters something, for example a number, and you have no control over what the user enters. As another example, one can imagine a program that will use a file that does not exist. In those situations, the program will handle the error, which means that it must be able to find that there is an error and if so, decide what is to happen. Typically it will be such that a method can capture that there is an error, but the method can't know how the error should be handled, but it must instead let the code from where it was called, know that there has been an error and leave the error handling to the calling code. One way to solve the problem is to let the method that can detect the error return an error code and the calling code can then test the error code and take an appropriate action determined by the error code. This strategy is really good, but it can only be used in situations where the method would not else have a return value.

Another strategy is to use *exception handling* (and I have done it many times already). The idea is pretty simple. A method can, in the case of a fault throw an exception, and if it does so, the method is immediately interrupted. The place from where the method was called can then choose to catch the exception and take an appropriate action. Consider as an example (from the project *ExceptionProgram*) the following method that calculates the ratio of two integers:

```
static int Div(int a, int b)
{
  if (b == 0) throw new ApplicationException("Division med 0...");
  return a / b;
}
```

One can, as we know not divide by 0. If *b* is 0, the method can't perform the calculation, but it may also not know what to do. It can't just return something, because it could by the calling code be interpreted as a result that would be illegal. The method may test the value of *b*, and if it is 0 it raises an exception. This means that the method stops with an exception that is sent to the calling code. Below is a code that uses the above:

```
static void Test1()
{
  try
  {
    Console.WriteLine(Div(23, 5));
    Console.WriteLine(Div(23, 0));
  }
  catch (ApplicationException ex)
  {
    Console.WriteLine(ex.Message);
    Console.WriteLine("The method is completed with an error...");
  }
}
```

The code that may raise an exception, here the method *Div()*, can be placed in a try block. If *Div()* raises an exception, the control is transferred to the subsequent *catch* handler that then performs an error handling, which here is merely to print a message on the screen.

An exception is a type, which is a class. Above is the type *ApplicationException*, but there are other options that are explained below. For an exception to be caught in the calling code, there must be consistency between the type of the exception that is raised, the type after the *throw* and the type after *catch*.

The calling code does not need to catch an exception. Is it the case and an exception are not treated at any level, the program will be terminated with a default exception handling:

```
static void Test2()
{
  Console.WriteLine(Div(23, 5));
  Console.WriteLine(Div(23, 0));
}
```

As mentioned, there are multiple exception types, actually many and most of them are classes in the .NET Framework that may raise a variety of exceptions. The fundamental base class is called *Exception*, and belongs to the *System* namespace. *ApplicationException* is a derived class, and although a custom code may as well raise an *Exception* (and often do), it is the thought that the type of an exception raised by a custom method must either be an *ApplicationException* or a type derived there from.

As an example of how the exception handling works in C#, I will use the project *Students3* with classes representing courses and students at an educational institution. I have created a copy of the project, and that is a copy of the folder *Studens3* with all its content. The copy is called *Students4*, but you should note that the name of the project still is *Students3* and can be opened and executed in Visual Studio as it is.

The principle in exception handling is that a method can fail, and if so, it raises an exception, which means that the method immediately is interrupted with a message to the method that called it. Calling a method that can raise an exception, the calling code can handle this exception, which typically consists of encapsulating the method call in a try / catch. Otherwise, the calling code is sending an exception further up in the hierarchy of the calling methods. The idea is that the method that can raise an exception can see that something is wrong, but the method cannot know how the error should be handled. It can, however, send an exception to the calling code in the hope that it knows how the error should be handled, and does it not that, it can forward the message up in method hierarchy.

Above, (in this and the previous books) I have everywhere let methods raise an exception of the type *Exception*, and although it works, it's not the idea. The .NET runtime classes raises many exceptions and they have types that tell us something about what is the reason for the exception. These exception types are all directly or indirectly derived from the class *Exception*, and although one probably can use some of these types of exceptions, it is typical that you in the context of an application defines your own exception types. The reason is partly that in this way you can classify the various exceptions and treat them differently but also that you get the chance to send values to the calling code. When you write your own code and you want methods to raise exceptions these exceptions must have the type *Exception*

or *ApplicationException* that is derived from *Exception*, but better custom exceptions that must have type that is directly or indirectly derived from *ApplicationException* or *Exception*.

In the project concerning students I have added the following class:

```
public class StudentsException : ApplicationException
{
  public StudentsException(string message) : base(message)
  {
  }
}
```

It is an exception type for exceptions for this program. You should note that the class does not add anything new, and its sole purpose is to characterize the exceptions raised by methods of the program's classes. In this way, these exceptions are distinguished from exceptions raised by other classes from the .NET framework or classes from other projects. Also note that the base class is *ApplicationException*.

I also added the following exception type (in the same file as *StudentsException*) which defines the exceptions for the class *Subject*:

```
public class SubjectException : StudentsException
{
  private string field;

  public SubjectException(string message, string field) : base(message)
  {
    this.field = field;
  }

  public string Field
  {
    get { return field; }
  }
}
```

The class *Subject* raises an exception if one of its fields are assigned an illegal value. Therefore extends this class *StudentsException* with a variable that indicates which field is illegal. With

these types the class *Subject* must be updated as follows, where I have only shown the methods that raises an exception:

```
public class Subject : ISubject, IPoint
{
  public Subject(string id, string name, int etcs = 0)
  {
    if (!SubjectOk(id, name)) throw new SubjectException("The subject
    must have both an ID and a name", "name");
    ...
  }

  public string Name
  {
    get { return name; }
    set
    {
      if (!SubjectOk(id, name)) throw new SubjectException("The subject
      must have a name", "Name");
      name = value;
    }
  }

  public int ECTS
  {
    get { return ects; }
    set
    {
      if (value < 0) throw new SubjectException("ECTS must be non-
      negative", "ECTS");
      this.ects = value;
    }
  }
}
```

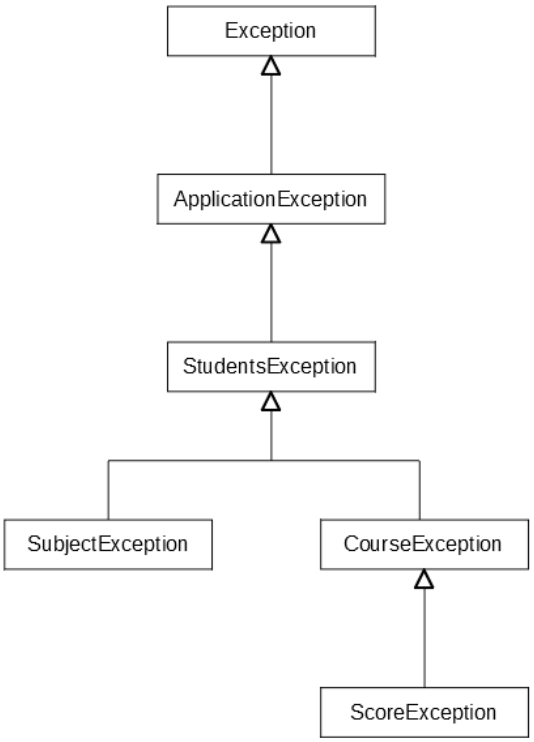
Note that the program can compile and run, although the test methods refers to *Exception* instead of *SubjectException*, as a *SubjectException* is especially an *Exception*.

For courses I have defined the following exception types:

```
public class CourseException : StudentsException
{
  public CourseException(string message) : base(message)
  {
  }
}

class ScoreException : CourseException
{
  private int score;

  public ScoreException(string message, int score) : base(message)
  {
    this.score = score;
  }

  public int Score
  {
    get { return score; }
  }
}
```

Next, the class *Course* should be updated with these types.

I then have defined four exception classes which form a hierarchy:



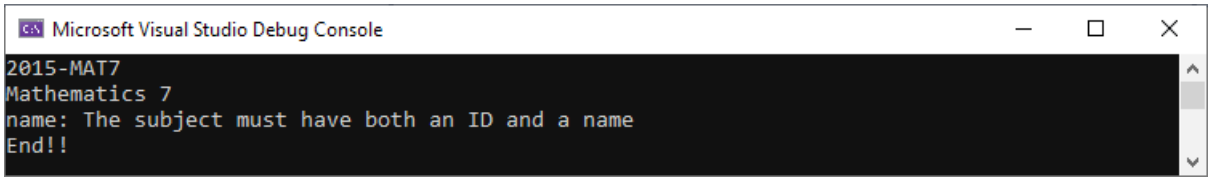
It is very common that exception types for a program in this way consists of a hierarchy of classes with *Exception* (or as here *ApplicationException*) as a common base class. On the other hand, it depends on the task, programmer, and so on how far you go and how many types you define, and above I have probably gone too far as the benefits of the many types is modest.

When you have a method that may raise an exception, the calling code can be placed in a try / catch. Thereby forcing the programmer to deal with the exceptions that a method can raise. It may seem cumbersome, but it helps to provide a robust code. Is there anything that can result in an error, then the programmer must necessarily consider what to do if the error occurs. It's the whole fundamental principle of exception handling. Below is a test method:

```
private static void Test5()
{
  try
  {
    ICourse course1 = Factory.CreateCourse(2015, "MAT7", "Mathematics",
    10, 4);
    course1.Score = 7;
    Console.WriteLine(course1.Id);
    Console.WriteLine(course1 + " " + course1.Score);
    ICourse course2 = Factory.CreateCourse(2015, "MAT6", "", 10, 7);
    Console.WriteLine(course2.Subject);
  }
  catch (SubjectException ex)
  {
    Console.WriteLine(ex.Field + ": " + ex.Message);
  }
  catch (ScoreException ex)
  {
    Console.WriteLine(ex.Message);
  }
catch (CourseException ex)
  {
    Console.WriteLine(ex.Message);
  }
  finally
  {
    Console.WriteLine("End!!");
  }
}
```

The code in the try block can raise three kinds of exceptions, and there is, therefore, three catch blocks. It is the type of a possible exception that determines which catch block is performed, and it is one of the main reasons to have more exception types because in this way you can control the action that should happen. In this case there is no particular reason for it, since the error handler each time consists of printing an error message. Note, however, the first catch, where I use, that the exception has a property *Field*. You should also notice that the handler for *ScoreException* must come before the handler for *CourseException* because a *ScoreException* also is a *CourseException* and the runtime system looks for a match from start to bottom.

Finally, there is a *finally* block. It does not have to be there, and in this case, it makes no sense, but a finally block is always performed regardless of whether there is an exception or not. It may, for example be used to close files, terminate connections to databases etc. If the method is executed the result is:



Here you should note that the last statement in the try block is not executed because it raises an exception and thus interrupt to the try block. Note also that the *finally* block is performed.

As mentioned, there in this example is no particular need to distinguish between the three types of exceptions, and the code could instead be written as follows:

```
private static void Test6()
{
  try
  {
    ICourse course1 = Factory.CreateCourse(2015, "MAT7", "Mathematics",
    10, 4);
    course1.Score = 7;
    Console.WriteLine(course1.Id);
    Console.WriteLine(course1 + " " + course1.Score);
    ICourse course2 = Factory.CreateCourse(2015, "MAT6", "", 10, 7);
    Console.WriteLine(course2.Subject);
  }
```

```
  catch (StudentsException ex)
  {
    Console.WriteLine(ex.Message);
  }
  finally
  {
    Console.WriteLine("End!!");
  }
}
```

for all three kinds of exception's are also a *StudentsException*. As a last remark you can also write:

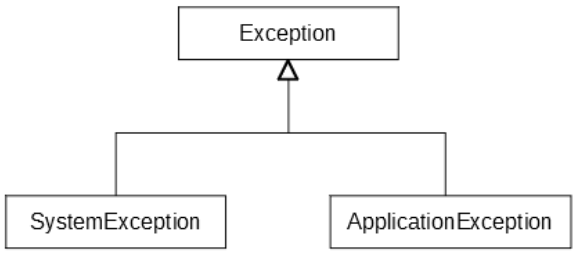
```
private static void Test7()
{
  try
  {
    ICourse course1 = Factory.CreateCourse(2015, "MAT7", "Mathematics",
    10, 4);
    course1.Score = 7;
    Console.WriteLine(course1.Id);
    Console.WriteLine(course1 + " " + course1.Score);
    ICourse course2 = Factory.CreateCourse(2015, "MAT6", "", 10, 7);
    Console.WriteLine(course2.Subject);
  }
  catch
  {
    Console.WriteLine("Error");
  }
  finally
  {
    Console.WriteLine("End!!");
  }
}
```

Here the catch handler handles all exception whether they come from the current program or system.

As discussed above a user-defined should inherit *ApplicationException*, but many let user defined exceptions directly inherit from *Exception* and both options are acceptable.

7.1 SYSTEM EXCEPTION

When an application raises an exception, it should as mentioned always be of a type that inherits *ApplicationException* (or *Exception*) or a type that is directly or indirectly derived from *ApplicationException*. The class is, as mentioned derived from the class *Exception* and there is nothing that prevents that one can raise an exception of the type *Exception*. When you should not do, it is because there is another exception class called *SystemException* that are also derived from *Exception*. Exceptions of this type are intended for exceptions raised by the runtime system or the basic classes from the framework, and they indicate fundamental and serious problems where the typical handling is to terminate the program with an error message. The following hierarchy separates the exceptions into two categories, one category indicate failure, which typically results in that the program exits with an error message while the second category indicates the error caused by the program's environment or use, and thus an error that should be handled by the program.



As an example is below shown a method which results in an error, because it are indexing outside an array (the project *ExceptionProgram*):

```
static void Test3()
{
  int[] t = { 2, 3, 5, 7, 11, 13, 17, 19 };
  try
  {
    int s = 0;
    for (int i = 0; i < 10; ++i) s += t[i];
    Console.WriteLine(s);
  }
  catch (Exception ex)
  {
    Console.WriteLine(ex.GetType().Name);
    Console.WriteLine(ex.GetType().BaseType.Name);
    Console.WriteLine(ex.GetType().BaseType.BaseType.Name);
  }
}
```

The result is as follows:

7.2 THE EXCEPTION CLASS

Exception is a class and a class that inherits directly from *Object*. As shown above the class has a property *Message* and it has also other methods as I will not mention here, but it has a property *Data* that you should know.

If you develop an application you should as shown above write your own exception classes for two reasons

1. You can in catch handlers distinguish between the different exceptions and especially between exceptions raised by the current application and system exceptions.
2. Exception types can have their own properties and you can thus associate data with an exception.

Here is the first reason the most important, but you can also associate data with an object of the type *Exception*. Such an object has a property called *Data* and the type is an *IDictionary* which is a collection class and is first explained in the next book, but it is a collection containing key / value pairs where you associate a value to a key. Both the key and the value can be an arbitrary *object*. As an example I will use a class which I have looked at before, namely the class *Annuity* (the project *ExceptionProgram*) but in a slightly reduced version and where the individual methods can raise an exception:

```
class Annuity
{
  private double principal;
  private double interestRate;
  private int periods;

  public Annuity(double principal, double interestRate, int periods)
  {
    if (principal < 0 || interestRate <= 0 || interestRate > 0.5 ||
    periods < 1 || periods > 120)
    {
      Exception ex = new Exception("Illegal annuity");
      if (principal < 0) ex.Data.Add("Principal", principal);
      if (interestRate <= 0 || interestRate > 0.5) ex.Data.
      Add("InterestRate", interestRate);
      if (periods < 1 || periods > 120) ex.Data.Add("Periods", periods);
      throw ex;
    }
    this.principal = principal;
    this.interestRate = interestRate;
    this.periods = periods;
  }

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

  public double Outstanding(int n)
  {
    if (n < 1 || n > periods)
    {
      Exception ex = new Exception("Illegal argument");
      ex.Data.Add("Periods", n);
      throw ex;
    }
    return principal * Math.Pow(1 + interestRate, n) - Payment() *
    (Math.Pow(1 + interestRate, n) - 1) / interestRate;
  }
```

```
  public double Interest(int n)
  {
    if (n < 1 || n > periods) { ... }
    return Outstanding(n - 1) * interestRate;
  }

  public double Repayment(int n)
  {
    if (n < 1 || n > periods) { ... }
    return Payment() - Interest(n);
  }
}
```

The constructor checks the parameters, and if one of them are illegal it raises an exception of the type *Exception*. First is created an *Exception* object (or instead an object of any other exception type), and to this object's *Data* property is added a key / value pair for each illegal parameter and then the method throws the exception. The last three methods works in the same way and throw an exception with an illegal value if the parameter is illegal. The method *Test4()* creates an *Annuity* object where two of the parameters are illegal, and as so the constructor in the class *Annuity* raises an exception. The *catch* handler prints the data associated with this exception, and for now you should just accept the syntax:

```
static void Test4()
{
  try
  {
    Annuity a = new Annuity(10000, 0.6, 150);
  }
  catch (Exception ex)
  {
    foreach (object key in ex.Data.Keys) Console.WriteLine("{0}: {1}",
    key, ex.Data[key]);
  }
}
```

If you want to associate data with an exception, it is recommended to write your own exception types with properties for the data in question rather than the *Data* dictionary associated with the *Exception* class, as it provides a more flexible solution, but the latter option certainly has its uses.

EXERCISE 8: LIBRARY EXCEPTIONS

Create a copy *Library6* of your folder *Library5* with the solution of exercise 6 and the classes for the book library. Remember that your project still is called *Library5*. Open the copy in Visual Studio. The classes

- *Publisher*
- *Author*
- *Book*
- *Booklist*

can all throw exceptions that are of the type *Exception*. You must now define a hierarchy of exception types for these exception and use them in the four classes. For example, you can do something like:

```
public class LibraryException : ApplicationException
{
  public LibraryException(string message) : base(message) {}
}

public class PublisherException : LibraryException
{
  public PublisherException(string message) : base(message) {}
}

public class AuthorException : LibraryException
{
  public AuthorException(string message) : base(message) {}
}

public class BookException : LibraryException
{
  public BookException(string message, string isbn, string title, int?
  released, int? edition, int? pages) : base(message) {}
}

public class BooklistException : LibraryException
{
  public BooklistException(string message) : base(message) {}
}

public class BookCreatedException : BooklistException
{
  public BookCreatedException(string message) : base(message) {}
}
```

```
public class BookNotFoundException : BooklistException
{
  public BookNotFoundException(string message, string isbn) :
  base(message) {}
}
```

You must also update the test program to handle the various exceptions that may occur.

8 COMMENTS

Consider as an example the following class, which is a class from the project *ExceptionProgram*:

```
/// <summary>
/// Class which represents an annuity loan as a
///   principal value that must be none negative
///   interestRate that most be a decimal value between 0 and 0.5
///   periods that is the number of payments
/// The class assumes that the first periods falls one period after
    the loan's
/// foundation and that the interest rate is constant throughout the
    loan period.
/// </summary>
class Annuity
{
  private double principal;     // the loans principal
  private double interestRate;  // the interest rate as a decimal
                                   number
  private int periods;          // the number of payments

  /// <summary>
  /// Creates a annuity and initialize the loans parameters.
  /// </summary>
  /// <param name="principal">The principal for this loan</param>
  /// <param name="interestRate">The interest rate for this loan</
      param>
  /// <param name="periods">The number of payments for this loan</
      param>
  /// <exception cref="Exception">If one af the parameters are
      illigal</exception>
  public Annuity(double principal, double interestRate, int periods)
  {
    if (principal < 0 || interestRate <= 0 || interestRate > 0.5 ||
    periods < 1 || periods > 120)
    {
      Exception ex = new Exception("Illegal annuity");
      if (principal < 0) ex.Data.Add("Principal", principal);
      if (interestRate <= 0 || interestRate > 0.5) ex.Data.
      Add("InterestRate", interestRate);
      if (periods < 1 || periods > 120) ex.Data.Add("Periods", periods);
      throw ex;
    }
```

```
    this.principal = principal;
    this.interestRate = interestRate;
    this.periods = periods;
  }

  /// <summary>
  /// Returns the value of a payment. For an annuity has all payments
      the same value.
  /// </summary>
  /// <returns>The value of a payment</returns>
  public double Payment()
  {
    return principal * interestRate / (1 - Math.Pow(1 + interestRate,
    -periods));
  }

  /// <summary>
  /// Returns the outstanding immediately after the n'th payment.
  /// n must be greater than 0 and less or equal number of periods.
  /// </summary>
  /// <param name="n">Index for payment</param>
  /// <returns>Outstandig after the n'te payment</returns>
  /// <exception cref="Exception">If the index is illegal</exception>
  public double Outstanding(int n)
  {
    if (n < 1 || n > periods)
    {
      Exception ex = new Exception("Illegal argument");
      ex.Data.Add("Periods", n);
      throw ex;
    }
    return principal * Math.Pow(1 + interestRate, n) - Payment() *
    (Math.Pow(1 + interestRate, n) - 1) / interestRate;
  }

  /// <summary>
  /// Returns the interest part for the n'th payment.
  /// n must be greater than 0 and less or equal number of periods.
  /// </summary>
  /// <param name="n">Index for payment</param>
```

```
  /// <returns>Interest for the n'te payment</returns>
  /// <exception cref="Exception">If the index is illegal</exception>
  public double Interest(int n)
  {
    if (n < 1 || n > periods)
    {
      Exception ex = new Exception("Illegal argument");
      ex.Data.Add("Periods", n);
      throw ex;
    }
    return Outstanding(n - 1) * interestRate;
  }

  /// <summary>
  /// Returns the repayment part at the n'th payment.
  /// n must be greater than 0 and less or equal number of periods.
  /// </summary>
  /// <param name="n">Index for payment</param>
  /// <returns>Repayment at the n'te payment</returns>
  /// <exception cref="Exception">If the index is illegal</exception>
  public double Repayment(int n)
  {
    if (n < 1 || n > periods)
    {
      Exception ex = new Exception("Illegal argument");
      ex.Data.Add("Periods", n);
      throw ex;
    }
    return Payment() - Interest(n);
  }
}
```

I have this time inserted comments in the code. Comments have no effect on the translated program, but will only have affect for us humans to read and understand the code. Much can be said about comments, so some words about it.

It is important with comments in the code, and more than that, it should actually be a permanent part of the programming task quite on pair with writing the code itself, and there are at least two reasons:

1. All program code must be maintained over time, and often by other than who wrote the code. Therefore it is extremely important that the code has comments that tell about important decisions, and why the code is written as

it is. Comments are important not only for others but also because you do not remember your own code even if it is just a few months old.

2. The actual process of commenting the code is important because during this process, you think through the code and wonder why the code is written as it is. It is an extremely efficient method to find errors and discrepancies in the code, and get them corrected.

If you see my examples, you will in most cases not see any comments. This is partly because they are small examples and not actual applications that must solve practical everyday problems, and secondly, that the code is precisely explained in the books. It is true to say that a part of the books content could instead be comments in the examples.

In C# you have three kinds of comments. Above I have for the most written comments with the syntax:

```
/*
  This namespace contains several classes with miscellaneous methods.
  One can perceive the namespace as a custom class library.
*/
```

It's a little older kind of comment that has been inherited from the C programming language, but the characters /* start a comment and it will continue until you meet the characters */ and between these two markers can be all the text you want like a single line or spread over several lines. This kind of comment is not used as often anymore, but there's nothing wrong with it, and if you need at the beginning of a source file to write a lengthy documentation, it is an excellent form of comments.

The most common type of comment in C# is used in front of each method. This is partly due that Visual Studio auto generate a skeleton, which you must complete. If, for example you place the cursor in front of the method *Outstanding()* and press the / three times, Visual Studio generates the following skeleton to a comment:

```
/// <summary>
///
/// </summary>
/// <param name="n"></param>
/// <returns></returns>
```

Here, the programmer has to comment the following:

- A description of the method and what it does and including generally what the user of the method needs to know.
- An explanation of the method's parameters, including which rules (pre-conditions) that the parameters must satisfy.
- An explanation of the method's return value.

The advantage of using this kind of documentation, not just to methods, but for all elements, is

- that the documentation has a standard form and that you remember to document all important elements as parameters and return value
- that the documentation is used by IntelliSense in Visual Studio
- that the documentation is in XML form, and therefore can use of a tool to form a complete documentation for an entire program, for example as HTML

The last type of comment is simpler and consists of everything after // and to the end of the line is a comment (see the variables above). This comment is typically used to document the individual statements in for example a method, or the description of a variable or its equivalent.

It is easy to write comments, but quite another thing is what you should write, and there are many attitudes, and the following must then be mine. Generally, you should write what you believe that others and including even you self needs to know to read and understand the code and thus could maintain it. You should not write what is clear. You must assume that who must read the code knows the language, and you should not document the language itself, but the explanations for the choices made you must explain how the algorithms are used and how they work. Are there solutions that are difficult to comprehend, then you should add explanatory comments. It is also wise to always document the variables and what they are used for, at least instance variables. It can also be a good idea to add a comment, telling about modification of the code, when changes are made, to whom and why, and of course what's changed.

You should special be consistent about the auto-generated comments and include them, at least for all *public* program elements. It is especially important for class libraries, which often must be used by anyone other than the programmer who wrote the classes. It can be hard to write that kind of evidence simply because it can be hard to find something to write (many methods and properties are obvious and self-explanatory) and you often think that you do not have anything to write. Yet it is a place where you should be consistent

and include these comments. One should be aware that this kind of comments is intended for those who must use the code, and not to those who need to maintain the code.

The program's code readability is extremely important, and you can even go so far that the code that is not easy to read is worthless. However, you can do many things to make the code readable than writing comments and including the following few guidelines:

- a block starts on a new line containing only the character {
- from the next line makes an indentation of two characters
- when a block ends repealed the indentation, and you move two characters to the left
- a block always end on a new line containing only the character }
- add always a space on either side of an operator
- a variable name always starts with a lowercase letter
- a name of a method, property, and a user-defined type always starts with a capital letter
- be consistent in capitalization
- use good and explanatory names, but not too long names - they are hard to read
- use blank lines where you think it increases the readability

And so be consistent and have a style. Guidelines are good, but there will always be places where you may depart from them, but if you do it consistently, it is excellent. The above guidelines are to make the code self-documenting, and one can say that the comments should only be used if the code can't explain itself.

One can hardly say enough about the importance of writing readable program code, but in terms of comments, you can also go too far. Generally I feel that a comment inside a method makes the code harder to read. It can be difficult to see what is program code and what's comments, the comments shadows the code. Comments inside the code I include only where I think they are absolutely necessary, and it is certainly often the case. The conclusion is that documentation is important, but exaggeration may have the opposite effect.

9 MORE ON C# SYNTAX

There are a number of constructs and special syntax in C# that can help make life easier for the programmer. This chapter addresses the most important of these concepts. It's all something you can do without, but used sensibly, it's also something that can help make your code simpler and easier to read.

Each time a new version of the language comes, new concepts have been expanded, some of which are more crucial than others, but together they all have in common the desire to make writing the program easier for the programmer. It is among other things happened by making the compiler more and more able to "guess" what the programmer thinks with the code that is written. Put slightly differently, that is, the compiler is able to automatically generate parts of the code from the context, and the following deals in large part with that kind of extensions.

In principle, it is good, as the work in this way becomes easier for the programmer, but far more important also because it can otherwise help produce programs that are easier to read and understand, and that in turn means programs that are easier of troubleshooting and maintaining. However, there is also a downside as the result is a very large and complex language, so it can be very comprehensive to learn, and as complexity increases, there may also be a risk that solutions will be chosen that either lead to errors in the program or are inappropriate. The result is that C# has gradually evolved into something of a monster.

It is also worth thinking that this development is not new. Previous programming languages suffered from being too complex, which led to the development of simpler languages, and moreover, the same has been seen in the development in the processor field.

9.1 AUTOMATIC PROPERTIES

It is a good practice to make a class's instance variables private and then define the required access to the variables in the form of properties. In this way, it becomes the programmer who takes the responsibility for giving access to variables and thus also opens up possible side effects. The programmer can also enter control code in a set property and possibly raise an exception if a variable is changed to an illegal value.

The disadvantage of enforcing this principle is that one often has to write trivial properties, since many classes require both get and set properties for the class's variables. To remedy

this, C# has opened the way for the compiler to automatically generate these properties. Consider the following class, which represents a book:

```
public class Book
{
  public string Isbn { get; }
  public string Title { get; private set; }
  public int Edition { get; set; }

  public Book(string isbn, string title, int edition)+
  {
    Isbn = isbn;
    Title = title;
    Edition = edition;
  }
}
```

If you look at the line

```
public int Edition { get; set; }
```

the compiler will automatically create a variable of the type *int* as well as *get* and *set* properties. The same goes for the first property (*Isbn*), but here the type is *string* and is read-only. The property *Title* also has the type *string*, but here the *set* property is private, so the variable can only be changed from within the class.

Automatic properties are fine as they avoid having to write a number of trivial properties. However, there is reason to point out that if a property requires a code other than a return statement or a simple assignment, it is necessary to write specific properties yourself.

9.2 THE INDEX OPERATOR

I will then mentioned overriding the index operator or an indexer which I have already used many times as I find it a very useful notation. Given an array:

```
int[] v = new int[10];
```

Then you refer to the individual elements using an index, for example

```
v[3] = 29;
```

In this context, the square brackets are called the index operator.

The index operator is also used elsewhere, for example with a *List*:

```
List<int> list = new List<int>();
list.Add(2);
list.Add(3);
list.Add(4);
list.Add(5);
for (int i = 0; i < list.Count; ++i) Console.WriteLine(list[i]);
```

which means that you can essentially use a *List* as if it were an array.

Generally, the operator can be overridden to apply to custom types, so you can use custom types as an array. The syntax is a bit special, but or is quite simple to override the operator.

As an example a perfect number is a positive integer with the property that the number is the sum of its own real divisors, for example

6 = 1 + 2 + 3
28 = 1 + 2 + 4 + 7 + 14

In fact, it is not so easy to determine the perfect numbers and we do not actually know how many there are and if there is a greatest perfect number. As an example, I want to write a method that can print the first 8 perfect numbers (the project *SyntaxProgram*):


```
static void Test02()
{
  Perfect p = new Perfect();
  for (int i = 0; i < p.Length; ++i) Console.WriteLine(p[i]);
}

class Perfect
{
  private static ulong[] numbers = { 6, 28, 496, 8128, 33550336,
  8589869056, 137438691328, 2305843008139952128 };

  public int Length
  {
    get { return numbers.Length; }
  }

  public ulong this[int n]
  {
    get { return numbers[n]; }
  }
}
```

The class *Perfect* store the first 8 perfect numbers as an array. Note that it is a private array and thus it is not accessible outside the class. The class has a property *Length*, which specifies the number of perfect numbers that the class knows, and then there is an override of the index operator. Note the syntax, which is a bit odd, but the result is a property that returns the *n*th perfect number. In this case, the property has only a *get* part, but like all other properties, it can have a *set* part.

The Method *Test02()* creates a *Perfect* object and prints the numbers using an ordinary *for* loop, and the most important is that the object *p* is used as was it an array.

The class *Perfect* does not calculate the perfect numbers, but simply knows the numbers as constants in an array. As mentioned, we do not know the perfect numbers, and it is actually

quite complex (requires many calculations) to determine whether a number is a perfect number. As the example shows, the perfect numbers grow very fast and I can mention that the next perfect number is

2658455991569831744654692615953842176

In terms of programming, it is a number so large that it cannot be contained in an *ulong*.

9.3 OBJECT INITIALIZING

Consider the following class

```
public class Person
{
  public string Firstname { get; set; }
  public string Lastname { get; set; }
  public string Position { get; set; }

  public Person()
  {
  }

  public Person(string firstname, string lastname)
  {
    Firstname = firstname;
    Lastname = lastname;
  }

  public override string ToString()
  {
    return Firstname + " " + Lastname + (Position != null ? ", " +
    Position : "");
  }
}
```

The class does not require any explanation, but note that set properties are defined for all variables. The class has two constructors and you can create an object as follows:

```
Person p1 = new Person("Svend", "Tveskæg");
```

which means that you initialize two of the instance variables (the last one is null). However, objects can also be created as follows:

```
Person p2 = new Person { Firstname = "Harald", Lastname = "Båtand" };
Person p3 = new Person { Firstname = "Gorm" };
Person p4 = new Person { Firstname = "Erik", Lastname = "Lam",
Position = "Konge" };
```

where to initialize instance variables from a list and where the list specifies the properties to be assigned a value. The syntax is easy to understand and in reality it is just a matter of the compiler generating the following statements:

```
Person p2 = new Person();
p2.Firstname = "Harald";
p2.Lastname = "Båtand";
Person p3 = new Person();
p3.Firstname = "Gorm";
Person p4 = new Person()
p4.Firstname = "Erik";
p4.Lastname = "Lam";
p4.Position = "Konge";
```

The syntax therefore requires two things:

- the class must have a default constructor
- there must be *set* properties to the variables that it should be possible to initialize

Note that the class may have all the constructors needed and that one can combine the usual syntax to instantiate objects with the above.

The syntax is also used by many collection classes. For example you can create a list of integers as follows:

```
List<int> list1 = new List<int> { 2, 3, 5, 7, 11, 13, 17, 19 };
```

Another example is how to create a list of 6 Person objects:

```
List<Person> list2 = new List<Person> {
  p1, p2,
  new Person { Position = "Viking", Firstname = "Regnar", Lastname =
  "Lodbrog"},
  p3, p4,
  new Person { Lastname = "D. 1.", Position = "Dronning", Firstname =
  "Margrethe" }
};
```

The above way to create and initialize objects does not immediately provide the major benefits, but if you need to be able to create objects of a class in many different ways by initializing the objects differently, you get more flexibility and you are free to create a large family of constructors with all possible parameter combinations.

9.4 THE VAR KEYWORD

C# is a type strong programming language meaning that you must assign variables a type. C# however has a keyword *var* which can be used as an alternative to write the type name:

```
static void Test04()
{
  var a = 123;
  var b = Math.Sqrt(2);
  var s = a + b;
  var p = new Perfect();
  Console.WriteLine(s);
  Console.WriteLine(p[5]);
}
```

First of all you must note that this does not means that the variables do not have a type and the type of the four variables are

- *int*
- *double*
- *double*
- *Perfect*

The only thing that matters is that the compiler out of context is able to determine the type of variables and thus the variables get this type once and for all. So it is not a question that the variables can change type at some point.

One can then ask what the benefits are and there is simply no one, at least not in the above example. On the contrary, one should avoid using *var*, as it makes the code more difficult to read and understand, and the only thing you save is to write the specific types. When Microsoft has introduced the possibility, there is of course a reason, and this is called LINQ, which is explained in a later book. In connection with LINQ, among other things, it is the system that creates objects of types that the programmer does not immediately can know, and it is in these contexts that *var* is important and even necessary.

When it is the compiler that determines the type of variable defined as *var* there is some restrictions:

1. only local variables can be defined as *var*, but not instance variables and static variables
2. *var* cannot be return type for a method
3. a local variable defined *var* must be initialized in the declaration
4. a variable defined as *var* cannot be *null*

Note that if you have defined a variable as

```
var p = new Perfect();
```

you can write very well later

```
p = null;
```

for at that time the compiler knows that the type of *p* is *Perfect*.

9.5 EXPRESSION BODIED METHODS

This is a possibility to write a method as an expression, and an example could be:

```
static double My(double x, double y, double z) => (x + y + z) / 3;
```

and the method can be used as all other methods. There are not many advantages and you can decide for yourself whether it is a good idea or not (I do not feel that), but if you need many simple methods, the opportunity is not so crazy. The syntax is naturally associated with lambda expressions, which are dealt with in the next book.

Note that you can also write a class as the following:

```
public class Calc
{
  public double Add(double x, double y) => x + y;
  public double Sub(double x, double y) => x - y;
  public double Mul(double x, double y) => x * y;
  public double Div(double x, double y) => x / y;
}
```

9.6 MORE ABOUT PARAMETERS

I have used parameters with default values in the past, and that just means that parameters have a default value if a method is called without a corresponding actual parameter. For example, you can write a method like the following:

```
static Random random = new Random();
static double Rand(double b = 1, double a = 0)
{
  return random.NextDouble() * (b - a) + a;
}
```

Such a method can be called with two, one or none parameters. The advantage is that in this way you can avoid writing many overrides of the same method. Not all parameters to a method need to have a default value, but if a method has parameters with default values it must be the last parameters.

Another possibility is to transfer parameters by name, and as example I will use the method which calculates the payment for an annuity:

```
static double Payment(double cost, double amount, double rate, int
periods)
{
  return (cost + amount) * rate / (1 - Math.Pow(1 + rate, -periods));
}
```

The method has four parameters and the principal is the sum of the two first. The following test method use the method to calculate the payment for the same annuity three times:

```
static void Test07()
{
  Console.WriteLine(Payment(1000, 10000, 0.01, 40));
  Console.WriteLine(Payment(1000, 10000, periods: 40, rate: 0.01));
  Console.WriteLine(Payment(periods: 40, rate: 0.01, amount: 10000,
  cost: 1000));
}
```

The first call is a normal method call where the order of values for the parameters are determined by the order over the methods parameters. The next call use a mix of a normal method call and named parameters. The first two parameters are values which must be values for *cost* and *amount*, while the last two are named parameters. You should note that the order of the last parameters not is the same as the order of the parameters in the method declaration, but it is not necessary when the actual parameters are named. In the last call of the method all parameters are named. You must note that if you mix normal parameter transfer with named parameters, the named parameters must be written as the last parameters.

Generally there is no need to use named parameters, but to some extent it can be said that it makes the code more readable, but the most important application is when calling methods with many parameters where it can be difficult to remember the order.

In the book C# 1 I described method's parameters, but there is one possibility called *out* parameters which I does not mentioned. It's almost the same as *ref* parameters with the difference that the actual parameters not need to be initialize, while the called method then must assign values to *out* parameters. The use of *out* parameters is as parameters to methods

which must return several values that not is possible as a return value. The following method determines the minimum and maximum value in an array, and the two values are returned as *out* parameters:

```
static void MinMax(int[] arr, out int min, out int max)
{
  min = int.MaxValue;
  max = int.MinValue;
  foreach (int t in arr)
  {
    if (t < min) min = t;
    if (t > max) max = t;
  }
}
```

The method is trivial, but you should note that the *out* parameters are initialized. Below is at test method which use the method *MinMax()*:

```
static void Test08()
{
  int[] arr = { 13, 7, 19, 23, 3, 11, 5, 2, 29, 17 };
  int a, b;
  MinMax(arr, out a, out b);
  Console.WriteLine(a + " " + b);
}
```

Note that the two parameters are not initialized which is not necessary.

9.7 LOCAL METHODS

In general a method is a member in a class and can be called from an instance of this class, and you should take this as a definition of a method, but a method can also be a local method in another method, although there may not be many good uses. The following method calculates the first prime that are greater or equal the parameter, and to test whether a number is a prime number the method use a local method:

```
static ulong NextPrime(ulong u)
{
  ulong m = (ulong)Math.Sqrt(u) + 1;
  if (u <= 2) return 2;
  if (u % 2 == 0) ++u;
  while (!IsPrime()) u += 2;
  return u;

  bool IsPrime()
  {
  if (u < 11) return u == 3 || u == 5 || u == 7;
  for (ulong t = 3; t < m; t += 2) if (u % t == 0) return false;
  return true;
  }
}
```

A local method is private to the enclosing method and can only be called from this method. You should note that the local method can reference local variables and parameters in the enclosing method.

A typical application of local methods could be to define an iterator.

9.8 MORE ON OPERATORS

When I explained struct types I mentioned nullable types which is an encapsulation of a struct in a class for the purpose that a struct variable can have the value null. The consequence is that in many cases it is necessary to test whether a variable is zero and, if so, assign it a value. The following method returns a random number or *null*:

```
static int? GetNumber()
{
  if (random.Next(2) == 1) return random.Next(10, 100);
  return null;
}
```

If you use the method you must test where the return value is *null*:

```
static void Test10()
{
  int sum = 0;
  for (int i = 0; i < 10; ++i)
  {
    int? t = GetNumber();
    if (t != null) sum += t.Value;
  }
  Console.WriteLine(sum);
}
```

However, there is an operator that can be used to assign a default value in case a nullable is *null*, and the above method could be written as:

```
static void Test11()
{
  int sum = 0;
  for (int i = 0; i < 10; ++i) sum += GetNumber() ?? 0;
  Console.WriteLine(sum);
}
```

This operator is called the ?? operator or the *null coalescing operator*.

If you have an object of a class type you often need to test if the object is *null*. As an example the following method has a *string* as parameter, and the method should return the value of the parameter as an *int*. Before converting the *string* to an *int* I have to test that the *string* is not *null* and has a positive length:

```
static int? Parse(string text)
{
  if (text != null && text.Length > 0) return int.Parse(text);
  return null;
}
```

This test can also be written in the following way, where the ? after the parameter test for *null*:

```
static int? Parse(string text)
{
  if (text?.Length > 0) return int.Parse(text);
  return null;
}
```

Note that the ? must be placed before the dot.

As the last I will mentioned the $ operator which not in fact is an operator. The program has a test method used to test the method *NextPrime()*:

```
static void Test09()
{
  for (ulong u = 0; u < 100; ++u) Console.WriteLine("{0, 3} {1}", u,
  NextPrime(u));
}
```

The method prints two values on each line and a line is formatted by formatting fields. It is possible to write the same in the following way:

```
static void Test13()
{
  for (ulong u = 0; u < 100; ++u) Console.WriteLine($"{u, 3}
  {NextPrime(u)}");
}
```

and the difference is that using a $ in front of a string you can directly inserts the values in the formatting fields.

The gain is not great but in connection with initialization of strings the syntax is quite appropriate and for example you can write something like the following:

```
static void Test14()
{
  DateTime date = DateTime.Now;
  string text = $"Today: {date.Day}.
    {CultureInfo.CurrentCulture.DateTimeFormat.GetMonthName(date.
    Month)}
    {date.Year}";
  Console.WriteLine(text);
}
```

9.9 INNER TYPES

It is possible to have a class inside another class. As example is below shown a class *Polygon* which has an inner class called *Vertex* that again has an inner *struct* called *Point*:

```
class Polygon
{
  private Vertex[] vertices;

  public Polygon(params Vertex[] v)
  {
    vertices = new Vertex[v.Length];
    for (int i = 0; i < vertices.Length; ++i) vertices[i] = v[i];
  }

  public double Length
  {
    get
    {
      double sum = 0;
      for (int i = 1; i < vertices.Length; ++i)
        sum += vertices[i].Length(vertices[i - 1]);
      return sum;
    }
  }

  public class Vertex
  {
    Point p = new Point();
```

```
    public Vertex(double x, double y)
    {
      p.x = x;
      p.y = y;
    }

    public double Length(Vertex v)
    {
      return p.Length(v.p);
    }

    struct Point
    {
      public double x;
      public double y;

      public double Length(Point p)
      {
        return Math.Sqrt(Sqr(x - p.x) + Sqr(y - p.y));
      }

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

Let me start from the inside. The type *Point* is a struct that has only one method that calculates the distance between two points. It is a very simple type that represents a point in a coordinate system, and seen from the type there is access only to members of the type itself, it does not know members of the outer classes. In this case, the full name of the type is:

SyntaxProgram.Polygon.Vertex.Point

where *SyntaxProgram* is the program's namespace, but in fact there is only access to the type within the class *Vertex*, when it is not defined *public*. That is a *private* inner type can only be referenced from its outer class.

Then there is the class *Vertex* which is an inner class in *Polygon* and has the full name:

SyntaxProgram.Polygon.Vertex

When this class is *public* it can be used from the outside, but note that the name of the class is *Polygon.Vertex*. The class is simple and represents a vertex in a polygon. In addition to a constructor, it alone has a method *Length()*, which determines the distant from the current vertex to another and hence the length of an edge in the polygon.

The *Polygon* class consists of a number of vertices and a property that can determine the perimeter of the polygon as the sum of the lengths of its edges. Below is a test method that creates a triangle:

```
static void Test15()
{
  Polygon.Vertex v1 = new Polygon.Vertex(2, 3);
  Polygon.Vertex v2 = new Polygon.Vertex(10, 12);
  Polygon.Vertex v3 = new Polygon.Vertex(8, 1);
  Polygon t = new Polygon(v1, v2, v3);
  Console.WriteLine(t.Length);
}
```

Inner types are not used as much in practice (or I don't). In most cases, a type is a term that must be used in several places and at least within the current assembly, and there are rarely advantages in defining a type as an inner type, and although when defining the type as public you have the opportunity to refer to it from the outside, you get nothing but a more complex and less transparent syntax. However, there are exceptions. Sometimes you need a type that should only to be used within a particular class, and perhaps to create a stable code you exactly is interested in limiting the type's visibility to the defining class, and in such a case it might make sense, to define an inner type.

9.10 PARTIAL CLASSES AND METHODS

It is possible to work with partial classes, and Visual Studio does this to a great extent in applications that have a graphical user interface. A partial class is a class where the code is distributed across multiple files. As an example is defined below the class *Util*, which has a single method that returns a random bit string of a given length:

```
public partial class Util
{
  public static string BitString(int n)
  {
    StringBuilder builder = new StringBuilder(n);
    while (n-- > 0) builder.Append(True() ? '1' : '0');
    return builder.ToString();
  }
}
```

Note that the class is defined partial. The code for the method *BitString()* is not very mysterious as it simply consists of a loop that fills random characters ('1' or '0') into a *StringBuilder*, but note the method *True()* where the code is not shown above. It is in another file called *UtilCode*:

```
partial class Util
{
  public static bool True()
  {
    return rand.Next(2) == 1;
  }
}
```

It is a simple method that randomly returns true or false. The method uses a random generator, but it is defined in a third file called UtilData:

```
partial class Util
{
  private static readonly Random rand = new Random();
}
```

The sum of it all is that the class's code is divided into three files. In this case, there is no benefit to it, but if you write very large and complex classes, you can increase the readability by distributing the code into multiple files.

Methods can also be partially defined to some extent. I have added the following code to the class *Util* (the file containing the method *BitString()*):

```
public static void Sort(int[] v)
{
  Sort<int>(v);
}

static partial void Sort<T>(T[] t) where T : IComparable<T>;
```

Here you should note the last line, which defines a partial method. It is only a prototype and just tells us that the class *Util* has such a method. You can then try to apply the method from a program:

```
int[] v = { 7, 3, 19, 11, 5, 13, 2, 17 };
Util.Sort(v);
foreach (int t in v) Console.WriteLine(t);
```

And then the strange thing comes: In fact, you can translate and run the program without errors, even if the sorting method does not exist. Of course, the numbers are not sorted!

If, as above, a partial method is defined, but it is not implemented, then the compiler will simply ignore the method and also the statements that apply it. In this case, the program will call and execute the method

```
public static void Sort(int[] v)
{
}
```

since the compiler has simply ignored the call of the partial method.

If you now add the following code to the *UtilCode* file:

```
static partial void Sort<T>(T[] t) where T : IComparable<T>
{
  for (int i = 0; i < t.Length - 1; ++i)
  {
    int k = i;
    for (int j = i + 1; j < t.Length; ++j) if (t[j].CompareTo(t[k]) <
    0) k = j;
    if (k != i) Swap(ref t[i], ref t[k]);
  }
}

private static void Swap<T>(ref T a, ref T b)
{
  T c = a;
  a = b;
  b = c;
}
```

then everything looks different, and if you run the program, the numbers are sorted.

There are several limitations to partial methods:

- A *partial* method can only be defines in partial classes.
- A *partial* method must be *void* and cannot have a return value.
- A *partial* method cannot have *out* parameters, but all other parameters can be used.
- A *partial* method is implicit *private*.

It is because of the last requirement that in the above example, it is necessary with a *public Sort()* method which is calls the partial method.

Note that the above partial method is static. It is not necessary and a partial method can easily an instance method.

9.11 ANONYMOUS TYPES

It is possible to work with anonymous types, and the syntax is similar to the above for object initialization. Consider the following code:

```
static void Test17()
{
  var obj1 = new { Firstname = "Frede", Lastname = "Andersen", Job =
  "Fisherman" };
  var obj2 = new { X = 1.41, Y = 3.14 };
  Console.WriteLine(obj1);
  Console.WriteLine(obj2);
  double sum = obj2.X + obj2.Y;
  Console.WriteLine(sum);
  var obj3 = Create();
  Console.WriteLine(obj3);
  Console.WriteLine(obj1.Equals(obj3));
}

static object Create()
{
  return new { Firstname = "Frede", Lastname = "Andersen", Job =
  "Fisherman" };
}
```

The first statement creates an object that consists of three strings, but no type is explicitly specified. This means that the compiler creates an anonymous type which is derived from object and has three read-only fields. Therefore, since the type does not have a name, you can only use *var* for the type of the variable.

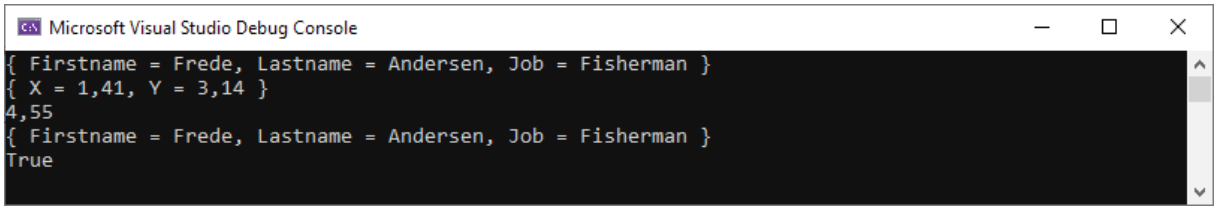
The same applies to the second statement, which also creates an anonymous type, and this time it will consist of two *double* fields.

When the compiler creates the anonymous type, it will also override *ToString()* so that it returns a string consisting of the key / value pairs that define the type.

Notice how to determine for *obj2* the sum of *X* and *Y* and how you can refer to an anonymous type's properties, but do not change them as they are read-only.

Also note the method *Create()*, which returns an object of an anonymous type, and it is perhaps the only immediate use of anonymous types, as one often needs a method to return multiple values, and then an anonymous type might be preferable to having to create a special type for that purpose.

Finally, note the last statement that prints *True*. An anonymous type overrides *Equals()*, so it has value semantics.



9.11 ANONYMOUS METHODS

An anonymous method is a method that does not have a name, and although it has nothing to do with delegates, anonymous methods are often used in this context (see the next book for delegates). Suppose the following delegate is defined:

```
delegate int Calculation(int a, int b);
```

That is a delegate that can reference a method that returns an *int* and has two parameters of the type *int*. To use this type for something, write a method with that signature and create a delegate variable that reference to the method:

```
static void Test18()
{
  Calculation c1 = new Calculation(Add);
  Calculation c2 = new Calculation(Mul);
  Console.WriteLine(c1(2, 3));
  Console.WriteLine(c2(2, 3));
}

static int Add(int a, int b)
{
  return a + b;
}

static int Mul(int a, int b)
{
  return a * b;
}
```

Often the method will be simple and may simply consist of a single statement (as above), and often the method should be called only by the delegate. It is for example the case in connection with an event. It can then seem cumbersome to have to write trivial event handlers, who may do nothing but call another method. This is where anonymous methods can be used. In the above example you could instead write:

```
static void Test2()
{
  Calculation c1 = delegate(int a, int b) { return a + b; };
  Calculation c2 = delegate(int a, int b) { return a * b; };
  Console.WriteLine(c1(2, 3));
  Console.WriteLine(c2(2, 3));
}
```

Here are *c1* and *c2* delegate variables that refer to methods, but instead of writing these methods explicitly in the normal way, one can define them anonymously as part of the variable declaration.

Consider the following type that has a method that can send an event:

```
struct Letters
{
  public event EventHandler<EventArgs> Listener;
  public char ch;

  public void Letter()
  {
    while (true)
    {
      Thread.Sleep(1000);
      ch = (char)Program.rand.Next(65, 91);
      if (Listener != null) Listener(this, new EventArgs());
      if (ch == 'A') break;
    }
  }
}
```

The method *Letter()* determines a random letter every second, and then sends a notification to any listeners, objects that have registered as event listeners for this event.

The following method creates an object of the type *Letters* and registers as a listener with an anonymous event handler that simply prints the letter that is generated. Next, the method *Letter()* is performed, which then calls the anonymous event handler every second.

```
public static void Test20()
{
  Letters b = new Letters ();
  b.Listener += delegate(object sender, EventArgs args) { Console.
  Write(b.ch); };
  b.Letter();
  Console.WriteLine();
}
```

The example above shows how to write an anonymous event handler, but in fact nothing but *Test20()* calls *Letter()* and waits until this method ends, and it is not a typical use of an event.

9.12 TUPLES

Tuples is a possibility to list values in parenthesis and let the compiler generate a type. The syntax is shown in the following method:

```
static void Test21()
{
  (string, double, int) value1 = ("Svend", Math.PI, 13);
  var value2 = ("Knud", Math.E, 17);
  var value3 = (Name: "Valdemar", Number: Math.PI + Math.E, Prime: 19);
  Console.WriteLine(value1);
  Console.WriteLine(value2.Item1);
  Console.WriteLine(value2.Item2);
  Console.WriteLine(value2.Item3);
  Console.WriteLine(value3.Name);
  Console.WriteLine(value3.Number);
  Console.WriteLine(value3.Prime);
}
```

The variable *value1* has three fields of different types, and data type for *value1* is called a tuple. Also *value2* and *value3* are tuples. As default you refer to the items in a tuple as *Item1*,

Item2 and so on, but you can also assign a name for the fields as is the case in *value3*. If the method is performed you get the result:

The most important use of tuples is as a return value for a method, and it can be used as an alternative to *out* parameters, since one can then get a method to return multiple values. For example the method *MinMax()* could be written as:

```
static (int a, int b) MinMax(int[] arr)
{
  int min = int.MaxValue;
  int max = int.MinValue;
  foreach (int t in arr)
  {
    if (t < min) min = t;
    if (t > max) max = t;
  }
  return (min, max);
}
```

and the method could be used as:

```
static void Test22()
{
  int[] arr = { 13, 7, 19, 23, 3, 11, 5, 2, 29, 17 };
  var res = MinMax(arr);
  Console.WriteLine(res.a + " " + res.b);
}
```

9.13 EXTENSION METHODS

As the last thing in this chapter about syntax I will mention extension methods. If you want to extend a class with new methods, the approach is to write a derived class that adds the new methods. It is still the "right" strategy, but it is not always possible, for example if the class is *sealed,* that is a class that you can't inherit. One can however achieve the same thing with an extension method.

Consider the following class that defines three static methods to integers:

```
public static class Integer
{
  public static long DiffSum(this int n)
  {
    return (n + 1L) * n / 2;
  }

  public static int Add(this int n, params int[] t)
  {
    int s = n;
    for (int i = 0; i < t.Length; ++i) s += t[i];
    return s;
  }

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

The first determines the sum of the numbers 1 + 2 + 3 + 4 + ... + N. This can be done with a loop, but you can also use a formula as has been done above. The second method returns the sum of a series of integers, while the latter method tests whether an integer is a prime. Since all the methods are static, they may be performed as follows:

```
Console.WriteLine(Integer.Add(2, 3, 5, 7, 11, 13, 17, 19));
Console.WriteLine(Integer.DiffSum(100));
Console.WriteLine(Integer.IsPrime(97));
```

which is not strange. You should however note that the class is static, and that the first parameter to each of the three methods is of the type *int*, and the declarations of these parameters are prefixed with the word *this*. It is the two thing that make the methods to extension methods. This means that methods can be performed as if they were instance methods defined for type *int*:

```
int a = 2;
Console.WriteLine(a.Add(3, 5, 7, 11, 13, 17, 19));
int b = 100;
Console.WriteLine(b.DiffSum());
int c = 97;
Console.WriteLine(c.IsPrim());
```

and not only that, the methods are known to Intellisense in Visual Studio.

Apparently the type *int* is extended with new methods, but it is obviously not the case. An extension method is a usual static method, and it should be written in the same way as other static methods and can't refer for instance members of the class to which it is an extension. There are only talking about that with the word *this* in front of the first parameter it allows to use a method with same syntax as if it were an instance method. If you compare the above applications of the methods in the class *Integer*, it is clear that it is only a question of how to specify the first parameter, as a normal value or by using dot notation.

Extension methods have their uses, and is as such used by Microsoft a great in relation to LINQ.

10 RECURSION

As a last thing about classes and methods in this chapter I will mention *recursion*. I will deal with recursion later, but here I briefly mentioned what it is, since I will use recursion in the final example.

A method in a class can be seen as an isolated code that performs a specific operation on the basis of parameters and possible returns a value. The method's statements, the commands it executes, can be all possible statements and there are no limitations on what it can be. For example calling another method, and a method may thus especially also calls the method itself. If so, we say that the method is recursive. As an example of a recursive method is shown a method that determines the factorial of a number *n*:

```
static long Factorial(int n)
{
  if (n == 0 || n == 1) return 1;
  return n * Factorial(n - 1);
}
```

At a first glance, recursive methods can be hard to figure out, but once you have been familiar with the principle, it is not particularly difficult. Above is the principle that if *n* is 0 or 1, you can directly determine the result (that is 1). If *n* is greater than 1, one can determine the factorial of *n* as *n* times the factorial of *n-1*. You can think of that in this way, to determine the factorial of *n* it is now reduced to determine the factorial of *n-1*, which is a smaller problem than I started with: To determine the factorial of *n*. If you repeat the above a sufficient number of times, you finally reach the simple case in which *n* is 0 or 1 and where you can directly determine the result.

The principle of a recursive method is that a problem may be divided into two problems: A simple problem which can immediately be solved, and a problem that can be solved as a simple problem and a problem smaller than and of the same kind as the original problem.

Formally, the factorial of *n* is defined as follows:

$$n! = \begin{cases} 1 & for\ n = 0,1 \\ n \cdot (n-1)! & for\ n > 1 \end{cases}$$

and then by a recursive definition, and in such situations, recursion is often a good solution. In this case, the method *Factorial()* simply is just a rewrite of the mathematical formula to a method in C#.

It is clear that in this case the method could be written iteratively by means of a simple loop, and this solution would even be preferable, but in other situations, recursion is a good solution to provide a simple solution and even a code which is easier to read and understand than an equivalent iterative solution. There is a reason to always be aware of recursive methods, as each recursive call creates an activation block on the program stack. There is therefore a danger that the recursive methods use the whole stack, with the result that the program will crash.

11 A FINAL EXAMPLE: A CALCULATOR

As the last example in this book I want to write a program that simulates a mathematical calculator. All computers have such a program installed, and there is a lot of them on the market. The primary goal of the example is to show the development of a program that uses a wide range of topics discussed in this book, and especially a program with many classes and interfaces. The result should be a GUI application that can be installed and used on a regular PC.

The development process will spread over 6 iterations:

1. Analysis, including determination of the requirements of the program.
2. Development of a prototype for the user interface.
3. Design of the model layer.
4. Programming of the main features so it is possible to use the calculator with selected functions.
5. Programming of the program's other features.
6. Code review and the last things.

11.1 ANALYSIS

The program's user interface should look like other calculation programs and must support the following mathematical functions:

- constant functions (functions without parentheses)
  - pi e
- functions in of 1 variable
  - sin cot sqr tan atan
  - asin acot sqrt log alog
  - cos ln abs deg rad
  - acos exp frac floor n!
- functions in 2 variables
  - pow
  - root

A mathematical expression must be entered either by clicking on buttons that represents the machine's keyboard or by typing on the usual PC keyboard. An expression should be entered on infix form, and an example could be:

$$2 * \sin(x0) + \mathsf{sqrt}(5 * x1 + 3)$$

that is an expression that depends on two variables (arguments) *x0* and *x1*, where the values of the two arguments must be either a number (entered as a decimal number) or a reference to an internal register (memory). The machine must be born with a number of registers where a register can contain either a number or an expression.

If there is an error:

- the user has entered an expression that is not syntactically correct
- the user has entered arguments that are not legal compared to the current expression
- an error occurs when the expression is evaluated

the program must show an appropriate error message.

With regard to entering expressions there are following requirements:

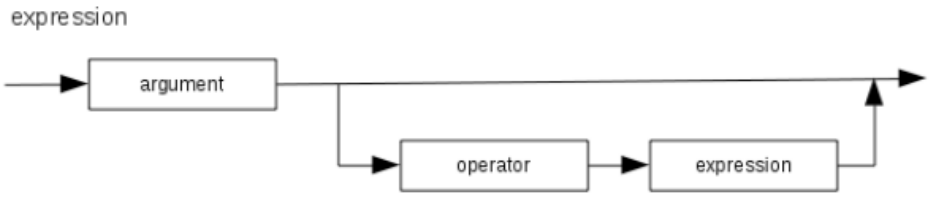
- An expression is not case sensitive, and it should not matter whether you write in lowercase or uppercase.
- An expression may contain a number of arguments, but the syntax for referring to a register has not been decided.
- Numbers is always entered with dot as decimal point.
- An expression must support the four common arithmetic operators +, -, * and /.
- It should be allowed to use parentheses in any number of levels.
- If a mathematical function has several arguments, they must be separated by comma.

The above is a description of the requirements for the program. The description is relative overall, but sufficient in this case, as it is the development of a well-known program.

An expression

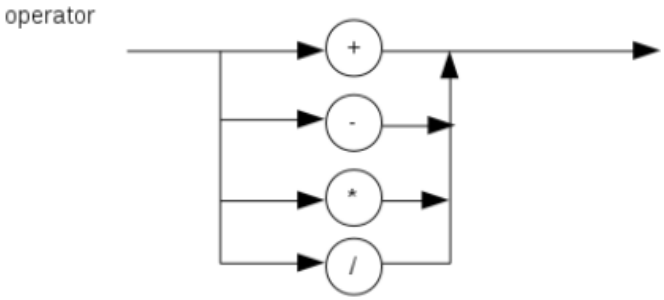
An expression is entered as text, and the program must parse the text to a legal expression. Above I have mentioned which functions an expression can contain, but an expression can

also contain operators, parenthesis and so on. It should also be possible to reference a register, and it is decided that a reference should have the form [999], that is a non-negative integer in brackets. Exactly an expression can be described with the following syntax diagrams:

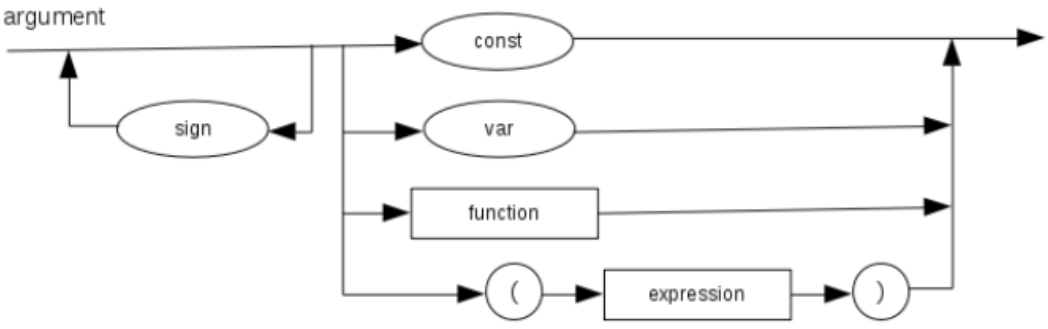


That is an expression can be an argument (a number or a register reference), or it can be an argument followed by an operator and an expression. An expression is then defined by itself that is by recursion. The above diagram shows what an expression is, when I also defines what an argument and an operator is.

It is simple to define an operator, as it is just one of 4 characters:

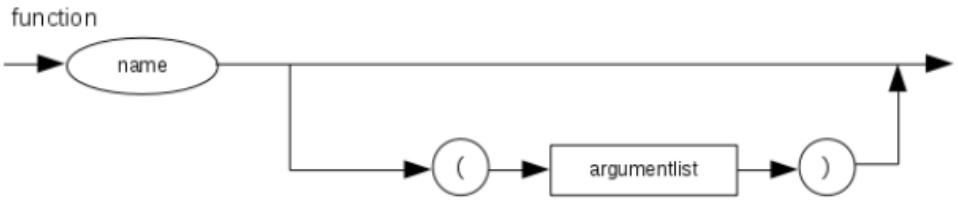


An argument is immediately more complex:



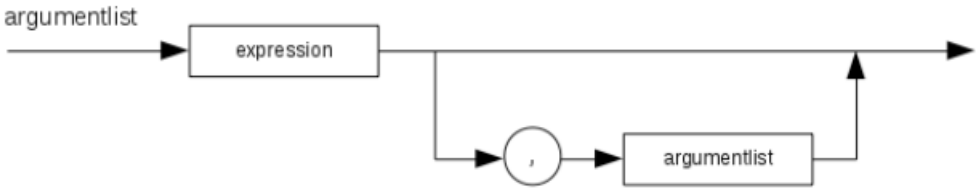
It can start with a sign, but do not have to. After then there are 4 possibilities. It can be a *const*, that is a number or it can be a *var* that is a reference to a register. It can also be a function, and it can be a left parenthesis followed by an expression followed by a right parenthesis.

I then has to define what a function is:



A function must have a name and can be followed by a left parenthesis and an *argumentlist* followed by a right parenthesis.

Last there is an argumentlist:



that is an expression, that can be followed by a comma and an *argumentlist*.

Then the following must be defined:

1. A *sign* that is + or -
2. A *const* that is a none negative decimal number that can be converted to a *double*
3. A *var* that is a reference to a register with the syntax [99], that is a none negative integer in brackets
4. A *name*, that is one of the names for the functions in the function list above

A last remark

As a last thing concerning the analysis one should aim for a solution that makes it easy to expand the program with new functions and without the major changes to the code.

Finally, it should be considered whether it should be possible to save the content of the program's registers, such the registers contains these values when the program is opened again.

11.2 THE PROTOTYPE

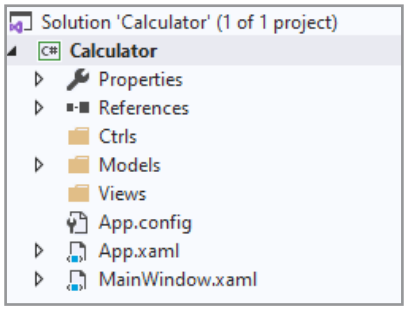
It is a program that is not doing anything. The buttons have no function, but the prototype must clarify two things:

1. The design of the user interface.
2. The overall program architecture.

The prototype will later be further developed for the completed program.

The architecture

Then I have created a WPF project with name *Calculator*. For now I have only the class *MainWindow*, and I have added three folders that should be used later:



The user interface

The design of the user interface is the following window and is the class *MainWindow*, and the meaning of the individual components is

1. The *DataGrid* to the left shows the program's registers, but only registers that are assigned a value. The first column is for the index.
2. The upper field is a read-only *TextBox* and is used for a history for calculations.
3. The next field is a *TextBox* and is the display where to enter an expression.

What the individual buttons means is for most self-explanatory, but at right there are three clear buttons:

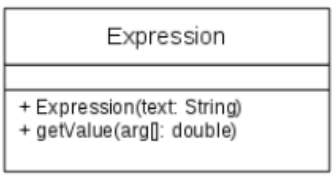
1. *ClH* clear the history
2. *ClR* clear the registries
3. *ClD* clear the display

You should note to make it simple a mathematical expression is entered as text and does not use symbols like the symbol for square root or the symbol for pi. Also note that I do not have chosen a font, size for font or colors and it is all postponed for later.

11.3 DESIGN

I will start the design by make a copy of the project *Calculator* (the copy is called *Calculator1*). In that way I can always, if something goes wrong, go back to the result from the analysis. In this iteration, I want to extend the model with a skeleton to a class *Expression*, which represents an expression.

An expression must be represented by a class named *Expression*. In principle, it is a very simple class that in addition to a constructor simply consists only of a single method that returns from arguments the expression's value:



Here it is the constructor, that is complex since, as parameter it has the expression as a string, and it is accordingly the constructor's task to split the string up into tokens and validate whether these tokens represents a legal expression. A token is an item that can be included

in an expression, and a token is a sub-string such as cos, + and a number. According to the above syntax diagrams and associated comments there are following tokens

+	addition or sign	-	subtraction or sign	*	multiplication
/	division	(	left parenthesis	)	right parenthesis
,	argument separator	sin	sinus	asin	arc sinus
cos	cosinus	acos	arc cosinus	tan	tangens
atan	arc tanges	cot	cotangens	acot	arc cotangens
ln	natural logarithm	exp	exponential function	log	logarithm 10
alog	antilog	sqr	square function	sqrt	square root
pow	power of	root	root of	abs	absolute value
frac	fraction	floor	interger part	pi	the constant pi
e	the constant e	deg	to degree	rad	to radian
!	factorial	[	start of argument	]	end of argument
id	an argument id that is a non-negative integer				
num	a number that is a non-negative decimal number with . as decimal point				

This can result in 35 different tokens, and indeed 36 as plus or minus especially can be a sign. The different tokens has to be treated different and should as such be implemented as different types, but can be arranged into groups with common properties, and then as types with a common base type. It means that the tokens can be designed as a class hierarchy.

When the program instantiates an *Expression* object, the parameter to the constructor is a *String* representing the expression on infix form, there is a format where the operators are written between the arguments and possibly with nested parentheses, as you typically enter expressions on a regular calculator. This format makes it more difficult to evaluate an expression (determine the value of the expression for specific arguments), and therefore an expression usually is converted to post-fix form where operators are written after the arguments and where parenthesis are not required. The constructor must then do three things:

1. scanning, where the string is split into a list of tokens
2. parsing, where to check that it is a legal expression corresponding to the above syntax diagrams
3. converting the list of tokens to a list of tokens on post-fix form

Each of these operations is relatively complex, but the class *Expression* can be designed as follows:

```
namespace Calculator.Models
{
  /// <summary>
  /// Type representing a mathematical expression. An expression is
      specified
  /// as a string at ordinary infix form and must be built up of the
      four
  /// arithmetic operators, mathematical functions and parentheses
      accordance
  /// with normal mathematical notation.
  /// There is only one significant method that returns the
      expression's value
  /// evaluated for the necessary arguments.
  /// </summary>
  public class Expression
  {
    /// <summary>
    /// Creates an expression by parsing a string. If the expression
        is illegal,
    /// the constructor raises an Exception.
    /// </summary>
    /// <param name="text">The string representing the expression</param>
    public Expression(string text)
    {
      try
      {
        List<Token> tokens = Scan(text);
        Parse(tokens);
        Convert(tokens);
      }
      catch (Exception ex)
      {
        throw new Exception("Parse error");
      }
    }

    /// <summary>
    /// Method that evaluates an expression. If the expression can not
        be
    /// evaluated the method raises an Exception.
```

```
    /// </summary>
    /// <param name="arg">The expression's arguments if it needs
        arguments</param>
    /// <returns>Value of the expression</returns>
    public double GetValue(params double[] arg)
    {
      throw new NotSupportedException();
    }

    // Scanning, which divides the string into tokens
    private List<Token> Scan(String text)
    {
      throw new NotSupportedException();
    }

    // Parsing, which controls the expression for the syntax errors
    private void Parse(List<Token> list)
    {
      throw new NotSupportedException();
    }

    // Converts the expression to postfix form
    private void Convert(List<Token> list)
    {
      throw new NotSupportedException();
    }
  }
}
```

Scanning

Scanning consists of splitting the input string into elements, where each element (sub-string) denotes a token. Then, for each of these items, create a *Token* object:

```
namespace Calculator.Models
{
  class Tokens
  {
    // Method that creates a Token from a name. The name that must be
    // converted to a token.
    // The parameter last is the last token scanned (is null if it is
    // the first token).
    public static Token ToToken(String name, Token last)
    {
      throw new NotSupportedException();
    }
  }

  // The fundamental base type for a token.
  // Each token has a priority (precedence) that is used when an
     expression is
  // converted from infix to postfix form:
  //     0 a number (constant), variable
  //     1 negative sign
  //     2 a function
  //     3 multiplication, division
  //     4 addition, subtraction
  //     9 left parenthesis
  //    99 right parenthesis, comma
  interface Token
  {
    int GetPriority();
  }

  // Represents an argument that is either a constant or an argument.
  abstract class ArgToken : Token
  {
    public int GetPriority()
    {
      return 0;
    }
  }

  // Represents an operator, which is one of the four arithmetic operators.
```

```
// The method value() returns the operator's value for two arguments.
interface OprToken : Token
{
  double GetValue(double arg1, double arg2);
}

// Represents a function with a property that specifies the number of
   arguments,
// and a method that returns the function's value.
// The following functions are supported:
//   pi       The constant pi
//   e        The constant e
//   Sin      Sinus
//   Asin     Arc sinus
//   Cos      Cosinus
//   Acos     Arc cosinus
//   Tan      Tanges
//   Atan     Arc tangens
//   Cot      Cotangens
//   Acot     Arc cotangens
//   Ln       Natural logarithm
//   Exp      Exponential functionen
//   Log      Logarithm with base 10
//   Alog     Antilog
//   Sqr      Square function
//   Sqrt     Square root function
//   Abs      Absolut value
//   Frac     Fraction value
//   Floor    Integer part
//   Pow      Power
//   Root     Root
//   Deg      Convert a value in radians to degrees
//   Rad      Convert a value in degrees to radians
// If a function is used with the wrong number of arguments, raise
   an Exception.
abstract class FuncToken : Token
```

```
  {
    public int GetPriority()
    {
      return 2;
    }

    // Returns the number of arguments
    public abstract int GetCount();

    // Returns the value of the function.
    public abstract double GetValue(params double[] x);
  }
}
```

and for each possible token should be written a class, that implements the interface *Token* (or a sub-interface). It is many classes, but all simple classes.

The result of the scanning is a list of *Token* objects.

Parsing

When you have the list of tokens from the scanning, you must check where the tokens constitutes a legal expression. That is, the program should check if the list of tokens is in accordance with the syntax diagrams from the analysis. Therefore, several recursive methods must be written, where there will be a method corresponding to each diagram as well as some other auxiliary methods. The methods must raise an exception, if there is a syntax error.

A stack

In the book C# 1 I described a *List* as an example of a collection class, and I have used a *List* many times since. I have also mentioned the program stack as a data structure, where the run-time system store parameters and local variables. C# also implements a collection class as a stack, and I need that class in the following, and therefore little about what a stack is.

It is as a *List* a container, that can store objects of a particular type, but it is a very simple data structure with only two important operations:

1. *push* that put an object on the stack (store a value at the top of the stack)
2. *pop* that returns an remove the object on the top of the stack

It is as such a LIFO data structure, where the object that is removed from a stack is the last object that is put on the stack. In C# a stack also has other operations, and as an example and operation *peek*, that returns the object on the top of the stack but without remove it. It means that *peek* only look at the top of the stack. As a *List* there is no limit on the number of objects a stack can contain.

In C# the type is called *Stack*, and I use that type in the program, but I will also refer to a stack below, and therefore this remark.

Post-fix form

Usually you write an expression on infix form which means that one writes an operator between two operands, as for example:

```
2 + 3
```

which means the sum of 2 and 3. If there are several operators, we need rules for how the expression must be evaluated. For example means

```
2 * 3 + 4
```

that you first calculate the product of 2 and 3 and then adds this result to 4 - the result is 10. In contrast, means

```
2 + 3 * 4
```

that you first calculates the product of 3 and 4, since multiplication has higher precedence than addition - the result is therefore 14. If you wish to suppress this rule, you has to use parentheses:

```
(2 + 3) * 4
```

and the expression has the value 20.

If in an expression there are several operators of equal priority, the rule is that the operators are evaluated from left. Below is first computed the sum of 2 and 3 and then subtract 4 because addition and subtraction have the same priority:

$$2 + 3 - 4$$

An expression may be more complex, for example

$$(1 + 2 * (3 + 4)) / ((5 + 6) * 7)$$

where there are parentheses within the parentheses. The value of the expression is 0.194805. When an expression contains parentheses, the parentheses are evaluated first, starting with the innermost parentheses. It is certainly possible to write a method that does it, but if the expression becomes more complex with mathematical functions and many parentheses, it is not simple, and therefore one will typically convert the expression to post-fix form. This means that an operator is written after the operands. Thus, the above expression's is written as

```
2 3 +
2 3 * 4 +
2 3 4 * +
2 3 + 4 *
2 3 + 4 -
1 2 3 4 + * + 5 6 + 7 * /
```

If for example you must calculate the value `2 3 * 4 +`, you have a list of five tokens, and you evaluates the expression by traversing the list from left to right, and every time you encounters an operator it acts on the two operands preceding:

```
2 3 * 4 +
  6 4 +
    10
```

The idea is that any expression can be written in post-fix form without using of parentheses. When the expression should be evaluated, it is traversed just from left to right. Every time you come to an operand, put it on a stack. Is it an operator you pop the stack twice (if it is an operator with two arguments) calculates the result and put it on the stack. Finally the stack will contain only one element which is the result. The method may, based on the last of the above expressions, be is illustrated in the following manner:

			4									
		3	3	7				6		7		
	2	2	2	2	14		5	5	11	11	77	
1	1	1	1	1	1	15	15	15	15	15	15	0.1948
1	2	3	4	+	*	+	5	6	+	7	*	/

The conclusion is that it is much easier to evaluate an expression in postfix form than one on infix form and it is therefore worthwhile to seek a strategy (an algorithm) to convert an expression from infix to postfix form. It may be done in the following manner by using a stack:

```
the expression is traversed from left and for every token

     1. if it is a sign push it on the stack
     2. if it is a function push it on the stack
     3. if it is a variable push it on the stack
     4. if it is a number push it on the stack
     5. if it is a left bracket, push it on the stack
     6. if it is a right bracket, then pop the stack and add the top of
        the stack to the result until you get a left bracket
     7. if it is an operator then pop the stack and add the top of the
        stack to the result as long as the priority of the top of the
        stack is less than or equal to the priority of the element,
        push the element on the stack
     8. pop the stack and add the top of stack to the result until the
        stack is empty
```

As you can see, it is crucial in the algorithm that there are assigned the right priorities for the individual tokens. It is the priorities that determines when to move from the stack to the result, which is just a list. The two important points in the algorithm are 6 and 7. If you get to an operator - for example a multiplication - you must first move everything on the stack with a better priority than multiplication to the result list. It will be numbers, variables and functions, and then the multiplication operator is put on the stack.

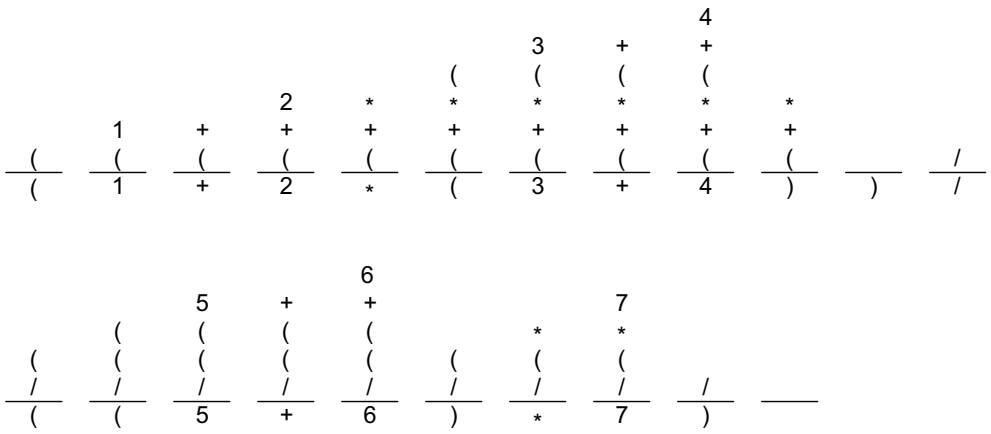
So, if the expression's tokens are converted to post-fix form, it is simple to implement the method *GetValue()* in the class *Expression*.

If you look at the expression

$$(1 + 2 * (3 + 4)) / ((5 + 6) * 7)$$

it can be converted to postfix form in the following manner:

```
1
1 2
1 2 3
1 2 3 4 +
1 2 3 4 + * +
1 2 3 4 + * + 5
1 2 3 4 + * + 5 6 +
1 2 3 4 + * + 5 6 + 7 *
1 2 3 4 + * + 5 6 + 7 * /
```



In this particular task, I will apply the following priorities:

- numbers, variables 0
- sign 1
- function 2
- multiplication, division 3
- addition, subtraction 4
- left bracket 9
- right bracket, comma 99

and from this table and the above algorithm it is simple to write a method to convert a list of tokens to post-fix form.

The model

With the class *Expression* in place, the model and thus the program's data presentation can be defined as follows:

```
class Model
{
  private String history;
  private Expression display;
  private List<Register> registeries = new List<Register>();
}
```

where

```
class Register
{
  private int id;
  private Expression expression;
}
```

11.4 PROGRAMMING, FIRST ITERATION

I am now ready to write the first version of the program. At first I have created a copy of the project from the design, a copy that I have called *Calculator2*. In this iteration I want to implement all functions, except the machine's history and the ability to use the machine's registers. That is that all buttons except

- Clh
- Clr
- Sto
- Rcl

must be implemented, and then you can use the machine for calculations.

The first is to implement the classes for all tokens. The classes are basically simple and are collected in the file *Tokens.cs*, but there are many tokens - 40 types in total, and as an example I have shown the class, that represents multiplication:

```
class MulToken : OprToken
{
  public int GetPriority()
  {
    return 3;
  }

  public double GetValue(double arg1, double arg2)
  {
    return arg1 * arg2;
  }

  public override string ToString()
  {
    return "*";
  }
}
```

The most important class is *Tokens*, which, based on a string (a name), creates a token object. It is that class that is used while scanning an expression, and the class is most of all a big *switch* statement.

Then here is the class *Expression* from the design. The class fills a part, and I will not show the code here, and compared to what is said under the design there is not much new. However, there is a single change. If you want to use a factorial in an expression, the format is

const !

and this situation is not described in the syntax diagrams. A factorial is in principle a function in one variable, but it is entered in another way than the other functions. Therefore, it is handled specifically in the class *Expression*. There are two places. When parsing an expression in the method *IsConst()* and in conjunction with the conversion to post-fix in the method *Convert()*.

Then there is the class *MainWindow* which must be updated, and that means to assign event handlers to all the buttons except the three clear buttons. The button *Enter* must have its own event handler, but the others all use the same event handler which has to insert a text in the display:

```
private void Cmd_Click(object sender, RoutedEventArgs e)
{
  try
  {
    string text = ((Button)sender).Content.ToString();
    int n = txtDisplay.SelectionLength;
    if (n > 0)
    {
      int pos = txtDisplay.SelectionStart;
      txtDisplay.Text = txtDisplay.Text.Substring(0, pos) + text +
      txtDisplay.Text.Substring(pos + n);
      txtDisplay.CaretIndex = pos + text.Length;
    }
    else
    {
      int pos = txtDisplay.CaretIndex;
      txtDisplay.Text = txtDisplay.Text.Insert(pos, text);
      txtDisplay.CaretIndex = pos + text.Length;
    }
    txtDisplay.Focus();
  }
  catch
  {
  }
}
```

The event handler for the *Enter* button should calculate the result and replace the text in the display with the result:

```
private void Enter_Click(object sender, RoutedEventArgs e)
{
  try
  {
    EX expression = new EX(txtDisplay.Text.Trim());
    string text = "" + expression.GetValue();
    txtDisplay.Text = text;
    txtDisplay.CaretIndex = text.Length;
    txtDisplay.SelectAll();
    txtDisplay.Focus();
  }
  catch (Exception ex)
  {
    MessageBox.Show("Illegal expression: " ...);
  }
}
```

When the project is translated, you have in principle a complete calculator that can be used for practical calculations.

11.5 PROGRAMMING, SECOND ITERATION

As the last iteration I starts this iteration to make a copy of the project from the previous iteration. The copy is called *Calculator3*. It has been decided that it should be possible to save the content of the program's registries in a file, and then I in this iteration have to solve three tasks:

1. The program's history
2. The program's registers
3. The ability to save the contents of the registers

The history

A line in the history is an element of the type:

```
namespace Calculator.Models
{
  class Calculation
  {
    public Expression Expression { get; private set; }
    public double Value { get; private set; }

    public Calculation(Expression expression, double value)
    {
      Expression = expression;
      Value = value;
    }

    public override string ToString()
    {
      return Expression + " = " + string.Format("{0:F12}", Value);
    }
  }
}
```

The reason for this type is that it should be possible to insert an expression from the history list by double-clicking on a line. Among other things, the design is changed, so the component for the program's history has been changed from a *TextBox* to a *ListBox*. Next the *Model* is changed:

```
class Model
{
  public event EventHandler CalculationsUpdated;

  private String history;
  private Expression display;
  private List<Register> registeries = new List<Register>();
  private List<Calculation> calculations = new List<Calculation>();

  public List<Calculation> GetCalculations()
  {
    return calculations;
  }

  public void Add(Calculation calculation)
  {
    calculations.Add(calculation);
    if (CalculationsUpdated != null) CalculationsUpdated(this, new
    EventArgs());
  }
}
```

The model now has a list with *Calculation* objects which represents the history and a method *Add()* to add an object to the list. When it happens the method raises an event to notify *MainWindow* that the history must be updated.

The *MainWindow* must now have an instance of the class *Model* and when the event handler for the *Enter* button is performed it must call the method *Add()* in the model. It is all needed to update the history.

The *ListBox* control in the user interface has assigned an event handler for *MouseDoubleClick* that inserts the expression for the *Calculation* object which the user double clicks in the display.

The registers

The use of the registers should works as follows. When the user enter an expression in the display and click on *Enter* a corresponding *Expression* object is stored in the model. This object in the model is always the result of the last expression which is parsed an evaluated. When the user click on the button *Sto* the program opens a dialog box where the user must enter a key as a positive integer, and the expression from the model is stored in a register with the entered number as key. If the key already exists the value is overwritten. If the user wants to use the content of at register there is to possibilities. If the user click on the button *Clr* the program opens a dialog box that look likes the dialog box for *Sto* and the user can enter the key for the value to be used. The user can then remove the register by clicking a button *Remove* or the user can inserts the content of the register in the display at the caret position. What is inserted is the key in square brackets. As an alternative the user can double-click at register in the *DateGrid* and a reference to the register is similarly inserted in the display. There is also a third option since the user can manually enter a reference in the display.

To implements the registers I must:

1. Update the class *Register*
2. Update the class *Model*
3. Add a dialog box to store a value in the registers
4. Add a dialog box to retrieve a value from the registers
5. Add an event handler for the *Sto* button
6. Add an event handler for the *Rcl* button
7. Some other adjustments in *MainWindow*

The class *Register* is modified as:

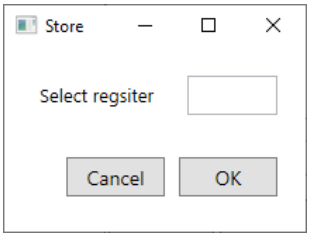
```
public class Register
{
  public Expression Expression { get; set; }
  public int Id { get; private set; }

  public Register(Expression expression, int id)
  {
    Expression = expression;
    Id = id;
  }
}
```

Then the model is updated where I have only shown what is changed:

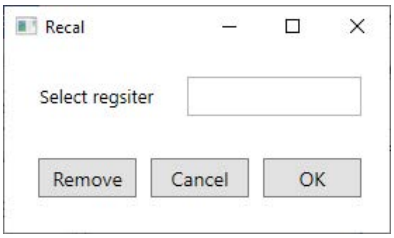
```
public class Model
{
  ...
  public event EventHandler RegisteriesUpdated;

  public Expression Display { get; set; }

  private List<Register> registeries = new List<Register>();
  ...

  public Register GetRegister(int id)
  {
    foreach (Register reg in registeries) if (reg.Id == id) return reg;
    return null;
  }

  public List<Register> GetRegisteries()
  {
    return registeries;
  }

  public void Add(Register register)
  {
    foreach (Register reg in registeries)
      if (reg.Id == register.Id)
      {
        reg.Expression = register.Expression;
        if (RegisteriesUpdated != null) RegisteriesUpdated(this, new
        EventArgs());
        return;
      }
    registeries.Add(register);
    if (RegisteriesUpdated != null) RegisteriesUpdated(this, new
    EventArgs());
  }

  public void Remove(Register register)
  {
    foreach (Register reg in registeries)
      if (reg.Id == register.Id)
```

```
        {
          registeries.Remove(register);
          if (RegisteriesUpdated != null) RegisteriesUpdated(this, new
          EventArgs());
          return;
        }
    }
}
```

If the user click on the *Sto* button the program opens a dialog box:



If the user click on the button *Rcl* the following dialog box opens:



The two dialog boxes are both simple, but it are these dialog boxes that updates the model, and to do that they must know the model as well as the display and these objects are parameters to the constructor.

The code for *MainWindow* must also be updates. When you look through the code you should pay special attention to

1. *MainWindow* is in the constructor registered as listener for events regarding updates to the registers.
2. The event handlers for the *Sto* and *Rcl* buttons which opens the above dialog boxes.
3. The event handler for double click on the *DataGrid* which should insert the line doubled click on in the display.

4. An expression must be able to parse a register reference, and to solve this problem the event handler for the *Enter* button substitutes any register reference with the value which is an expression. This substitution is performed using simple string operations.

You must also pay attention to the XML for *MainWindow* and here the definition of the *DataGrid*:

1. How to define an event handler for double click which is defined as a style.
2. How to bind the columns to properties in the class *Register* which is something that first is explained in the book C# 9.

Load and save registers

It must be possible to save the content of the machine's registers in a file. Similarly, it should be possible to load a stored register content, and for this it is necessary to decide where and when. To make it simple I will store the registers using simple object serializing as a file in the program directory. This solution raises two problems as it means that all users of the program uses the same stored registers and since it requires the program to be installed where the program has access to write a file. In this case, I will ignore both issues. With regard to when the contents of the registers are to be saved, it has been decided that this should be done automatically every time the model has changed the registers. The model must then have a designer who deserializes the registers.

To implements this solution the following must happen:

1. All token classes must be defined *Serializable*.
2. The class *Expression* must be defined *Serializable*.
3. The class *Register* must be defined *Serializabe*.
4. The model must be expanded with two methods *Serialize()* and *Deserialize()* to write and read the file.
5. This methods must be called in the right places.

As the last thing in this iteration I have assign event handler for the three clear buttons which are all trivial.

11.6 THE LAST ITERATION

Above, I have explained in detail the development of the application, although I have only shown a limited part of the code. You are naturally invited to try the program and study the code. Regarding the latter, primarily in the code for the user interface there will be things that I have not yet used and explained. It is generally a part of writing programs with a graphical user interface, where there are a lot of details about solving some very specific issues. Many of that kind of details are not explained at all in either this or the following books. If so, the page numbers would swell up to something that is totally unimaginable and yet even inadequate, but when you have such a problem, there is typically help on the Internet where there are lots of examples of how to solve specific problems.

When studying the code, be aware that in several places have been made choices that are not absolutely appropriate. This is because I still need to explain some concepts, and in particular I could have used a different collection class rather than a *List*. Another place where everything is not as it should be is the use of the MVC pattern. Here are the problems, who should create which objects (view or controller and the program has no controller classes) and how the communication between the three components model, view and controller should be, and where there is a goal that the three components should be as loosely coupled as possible. The program has made efforts to solve both issues, but there is more to tell which of the reasons for space consideration has been postponed to later.

Once you have written a program as above and in principle is done, there will always be several things outstanding. Firstly, the program is the result of a process, where code has been added and adjusted, and therefore the code will contain a number of inconveniences such as code no longer used or code that should be written in another way. Therefore, before completing the project, you must carry out a code review, reviewing the code, cleaning up and removing inconvenience and possibly adding missing comments. It can be an extensive job, but it is well-done. Firstly, it is a good opportunity to find and correct errors, and secondly, the result is a code that is far easier to understand and thus maintain.

To make a review I has in the same way as in the previous iterations stated with a copy of the project called *Calendar4*. The result of the review is:

1. For the classes in the model layer there is no changes, and the only thing I have done is remove all unnecessary using statements. There is no particular reason for that, but for future maintenance, they may cause confusion.
2. To the dialog box for the *Sto* button I have added an Activated event handler with the only purpose to assign focus to the *TextBox*. I have also defined the *OK* as button as a default button. In the same way I have defined the *Cancel* button as a cancel button.

3. The dialog box for the *Rcl* button is modified in exactly the same way.
4. To the *MainWindow* I have added an *Activation* event handler to assign focus to the display when the program starts. I have also defined the *Enter* button as a default button and the button *ClD* as a cancel button.
5. In the code for *MainWindow* two event handles performs almost the same thing (inserts text in the display). This code has been moved out in its own method.
6. The folder *Ctrls* which is empty is removed.

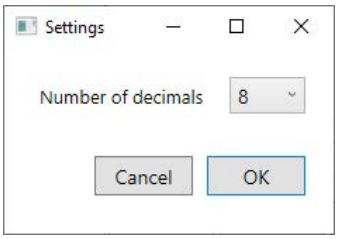
Finally, the program must be tested. The code has been tested for the individual iterations, and as part of the above code review, but before the application can be used, it must be done by a careful test, which may be performed by another than the developer. This work can also be extensive, and the result of this test will often be errors, which must obviously be corrected, but there are also often inaccuracies about the application's use and also even missing functions. As an example, in this case it has been found that it should be possible to specify how many decimals to use for the result. Therefore, the program is expanded with a function that opens a dialog box for selecting the number of decimals. It sounds simple, but a something has to be changed.

First I have added the following class to the model layer:

```
namespace Calculator.Models
{
  [Serializable]
  public class Options
  {
    public int Decimals { get; set; }

    public Options()
    {
      Decimals = 4;
    }
  }
}
```

and the class *Model* is expanded with a property for this type. The class *Options* is nothing more than a thin encapsulation of an *int* and that may seem a bit superfluous, but the reason is that in the future one must expect that the program needs some more of these kinds of settings.

Then I have added a dialog box that look like the two other dialog boxes just with a *ComboBox* instead of a *TextBox*:



To open the dialog box the button for factorial is replaced with two buttons, one for the factorial and one to open the above dialog box.

After the user to select the number of decimals is implemented the next step is to use setting in the program. There are two places

When the display is updated in the event handler for the *Enter* button.
When the history is updated.

To solve the first problem I have added the following method to format a decimal number:

```
public string Format(double value)
{
  string field = string.Format("0:F{0}", model.Settings.Decimals);
  return string.Format("{" + field + "}", value).Replace(",", ".");
}
```

and this method is called in the event handler when the display. To solve the use of the number of decimals in the history the class *Calculation* is changed where the type of the property *Value* is changed from *double* to *string* and the value used when creating a new *Calculation* object is then the result of the above *Format()* method.

The number of decimals and more general the settings should be saved so the program uses the same value the next time it is executed. I have decided in the same way as for the registers to serialize the program's settings to a file in the program folder and the model must therefore have methods to serialize and deserialize the settings. Since these methods are essentially identical to the same method regarding registers, I have instead changed these methods to generic methods:

```
private void Serialize<T>(string filename, T obj)
{
  try
  {
    FileStream file = File.Create(filename);
    BinaryFormatter bf = new BinaryFormatter();
    bf.Serialize(file, obj);
    file.Close();
  }
  catch
  {
  }
}

private T Deserialize<T>(string filename) where T : new()
{
  try
  {
    FileStream file = File.OpenRead(filename);
    BinaryFormatter bf = new BinaryFormatter();
    object obj = bf.Deserialize(file);
    file.Close();
    return (T)obj;
  }
  catch (Exception ex)
  {
    return new T();
  }
}
```

Back there is only to add a method to the *Model* class that can serialize the *Options* object and which is called from the dialog box to set the number of decimal places.