Table of Contents

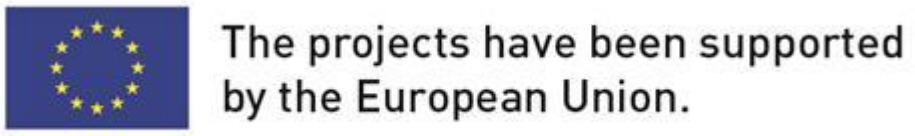
High-Level Programming Languages II.

Zoltán Hernyák

Object-Oriented Programming in practice

2014.03.01

This course is realized as a part of the TÁMOP-4.1.2.A/1-11/1-2011-0038 project.



1. 1. Introduction

The history of computer programing cannot be mapped easily. We often mention Mohamed Ibn Musza (? A.D. 800-850) as an important person, who lived as a mathematician, astronomer and geographer. His book - the Latin translation *Algoritmi (Algoritmi de numero Indorum)*, originally titled as "*On the Calculation with Hindu Numerals"* – related to computer science is interesting. In this book he gave us methods how make calculations in the 10-based numeral system. He formulated his method as a step-by-step procedure, so the very first algorithms were created by him. The name of the whole discipline *Algorithm* got its name from this book.

Furthermore many people formulated algorithms, solving problem in a step by step process. Until the electronic computers appeared these methods were interpreted and executed by humans. Alan Turing (1912-1954), the father of the modern computer science introduced an abstract machine, defined the minimal conditions, commands which are suitable to describe a solution of a problem. In this way he defined a universal algorithm-

descriptive language, containing a small number of different elementary instructions, and the effects of this commands can be described by state transitions. This descriptive language is not so suitable to define computer algorithms, but given by other ways (like in markup languages, flowcharts, etc.) can be transformed into this language. In its form an algorithm can be analysed, examined and studied using mathematical methods.

This Turing-machine served as a basis for the Neumann computers. In these computer a main memory can be found, which contains the data. We might think this as an actual state of our program. This program does nothing else than from the original data (original state) by making computations (elementary instructions, commands) modifies it and finding the final state containing the final values. The main unit who executes the commands of the algorithm is the processor, the CPU. For the CPU the elementary instructions are written in a special programing language named *machine code*.

Describing an algorithm we can choose from many different alternatives. The programing itself means nothing more than describing an algorithm in another way, a special way. Meanwhile the algorithm science tries to give the algorithms platform independently; in the programing languages we choose a concrete programing language to implement them. The algorithm science focuses on small problems to solve: define the problem accurately and describe the solution. Writing a computer program we usually have a complex problem to solve using and combining different algorithms.

A computer has limited resources. When we combine algorithms we usually modify them to exploit the computers limited resources in the most efficient way. It is the essence of the computer programming. However during the modifications we might produce code which contains errors. For this reason we must check the correct operation of our programs, for example with testing.

The machine code programming language is not suitable for writing computer programs, or implementing algorithms. With his low abstraction level one can make errors easily, but finding them is very hard. Another disadvantage is that this language is processor-dependent, which means different processors has different machine codes, heavily differing from each other.

The programming languages with higher abstraction level, like the assembly language or the procedural languages (C, Pascal, etc.) cannot be understood by the processors (so they simply do not exist for the processors). Programs (source codes) written in these languages must be translated into machine code, which is done by compilers. In this form the CPU can execute the instructions. To execute a computer program, not the original, this compiled code will start. It causes another uncertainty as when the compiler works badly, this means our source code might be errorless, but the program might produces errors. It is more common as we might think, mainly when we request code optimization (eq. execution speed, memory size).

The computer programs inside the same computer might cause bad influences, which might disturb each other's work. It is mainly the problem of the operating system as it cannot allow this kind of behaviour. The virtualization, the wide-spread delimitation of the software and hardware put the mark on the computer programming as well. Nowadays it is common that the compiler won't produce CPU's machine code instructions, instead transform the source code to a higher level "machine code", which belongs to a virtual CPU. Programs on this "machine code" cannot be executed directly on a physical CPU, but a virtual machine (a software component) can understand and execute the instructions. This solution has several advantages focusing on security and other similar considerations, but has disadvantages on execution speed and resource utilization.

According to the expectations of the world, the computer programs must meet the serious requirements. While earlier a few lines of code handled and solved the problems, nowadays the computer programs are the production of several programmers, who worked for several months together. For example, the Windows NT v3.1 (1993) was written in 4-5 million lines of source code, the Windows NT v3.5 (1994, in the made by very next year) contained 7-8 million, the Windows NT v4.0 (1996) had already 11-12 million lines of code. The Windows 2000 was built up from more than 29 million, and the Windows XP (presented in 2001) was written more using more than 45 million lines of code[1].

What is a computer program made of? We need data to work with. We store data in variables, which are stored in the main memory. The memory has limited size, so we try to minimize the utilization of it. We try to find a suitable type to store, which can accept the expected values but demands less bytes in the memory. Then we try to optimize the duration of storing data in the memory. Very rarely we try using static lifetime variables (only for the most required cases), and for the other cases we use dynamic variables to store data. When we have several data elements, which describes the same object and in this case they are in connection with each other,

we try to create and destroy them all in the same moment. So we use records, lists, arrays and other composite data structures.

The computer programs contain not only data, but processing instructions as well. The instruction which belongs to each other in a logical way, we arrange into *functions*. The execution of a computer program means calling (executing) the functions in a given order. The functions can work with global data elements and with parameter values.

This is called "traditional" programming style. It has several advantages and disadvantages. The disadvantages appear mainly in bigger projects. The function set made by the programmers cannot be tested easily, and when all the functions are tested and worked properly, we cannot be sure that after combining them they still work without producing errors. The continuous data elements passing and receiving by parameters often burden the processor unnecessary. When data is modified, it is hard to find out which function made that modification. It is true to the global data elements as well. This is why the invariant attributes of the variables' value hard to keep all the time[2]. The type invariant is the only one protection of the variables which are reliable. It means, if we have a variable with the type '*sbyte*', we can be sure that its value is between -128..127 all the time – but nothing else can be sure. If we need smaller interval (stronger invariant) but we have no proper type which holds this stronger invariant, we are at a dead end. In the traditional programming language we usually cannot really define new types (with new and stronger invariants) and so we cannot complement the interpretation of the operators existing in this language. Moreover we cannot define invariant which are formulated on two or more data elements (e.g. "when the value of A is even, then the value of B cannot be greater than 10").

Before we look over the solutions and the possibilities given by the Object-Oriented Programming style (shortly OOP) for these problems, make clear of some facts:

- all the computer programs which can be written in OOP style – can be written as well in the traditional programming style,
- we won't discover new control structures (conditional, loop control structures), in the body of the functions we will still use for, if, foreach, switch, and so on,
- we will still write functions with parameters, still passing and receiving them,
- OOP programs won't run faster (moreover their performance in this field is usually weaker than the programs designed and developer in the traditional way).

Advantages we gain in exchange for:

- our program will be built from better arranged units (logical groups of data and functions),
- these units can be tested together, so the proper work of the program can be better guaranteed,
- in many cases, less number of functions is enough,
- more complex invariants can be maintained on out data,
- new, fully functional types can be created, on which operators and its interpretation can be defined.

2. 2. The history of OOP

The history of programming languages can be characterized by the generation of the programming languages. The machine code is called the first generation. The next generations assumes an existence of a compiler: the programs written in higher generation languages must be translated into machine code.

The assembly language is considered as the second generation. This language is very close to the machine code language. Although there are many and important concepts introduced into the computer programming world, but this is only a more readable form of machine code itself. Its instructions can be mapped to machine code level as one-to-one.

The really big step was the appearance of the third generation, the so-called procedural, high level programming languages. It is also called the modular programming style which was introduced by it. Significant new programming concepts has appeared, but one of the biggest news was that a statement at this level is compiled

and translated into not only one but many machine code level instructions. This fact alone increased the effectiveness and the coding speed of the programmers.

The central element of the modular programming is the *function*. A function focuses on solving a particular problem; it has an identifier and a code block. A function might use previously written functions while solving its problem.

Instead of the low level control structures of the machine code (eq. conditional jump) new control structures were introduced and can be used in the function body (sequence, selection, iteration). When we use only these control structures to implement an algorithm we name it structural programming style. Two researchers, Corrado Böhm and Giuseppe Jacopini formulated the conjecture that every computable function can be developed only using these three control structures. It was an important conjecture because it says that the use of the "goto" statement could be disregarded.

One of the fathers of the Pascal programming language, Edgser Dijkstrawith his article "Go To Statement Considered Harmful" gave momentum to this direction. Today the most of the programming languages contains this kind of statements (e.g. break, continue), since using them we might reduce the complexity and increase the execution speed, and efficiency; but the use of them always must be considered and, if possible – be avoided.

The principles of the high-level programming languages seemed to be adequate, and still seem to be. There are programmers working nowadays who knows and are familiar with only this paradigm, and are developing their well working high performing applications. When they work alone, or in a small group close together this means no disadvantage. The pressure of the software development presented in the "introduction", however, pushed the evolution of the programming languages into new directions.

The principles of the object-oriented programming style were laid by Alan Curtis Kay[3] in his thesis in 1969. He started to work in Xerox Palo Alto Research Center then he went on to finish the development of principles in 1972.

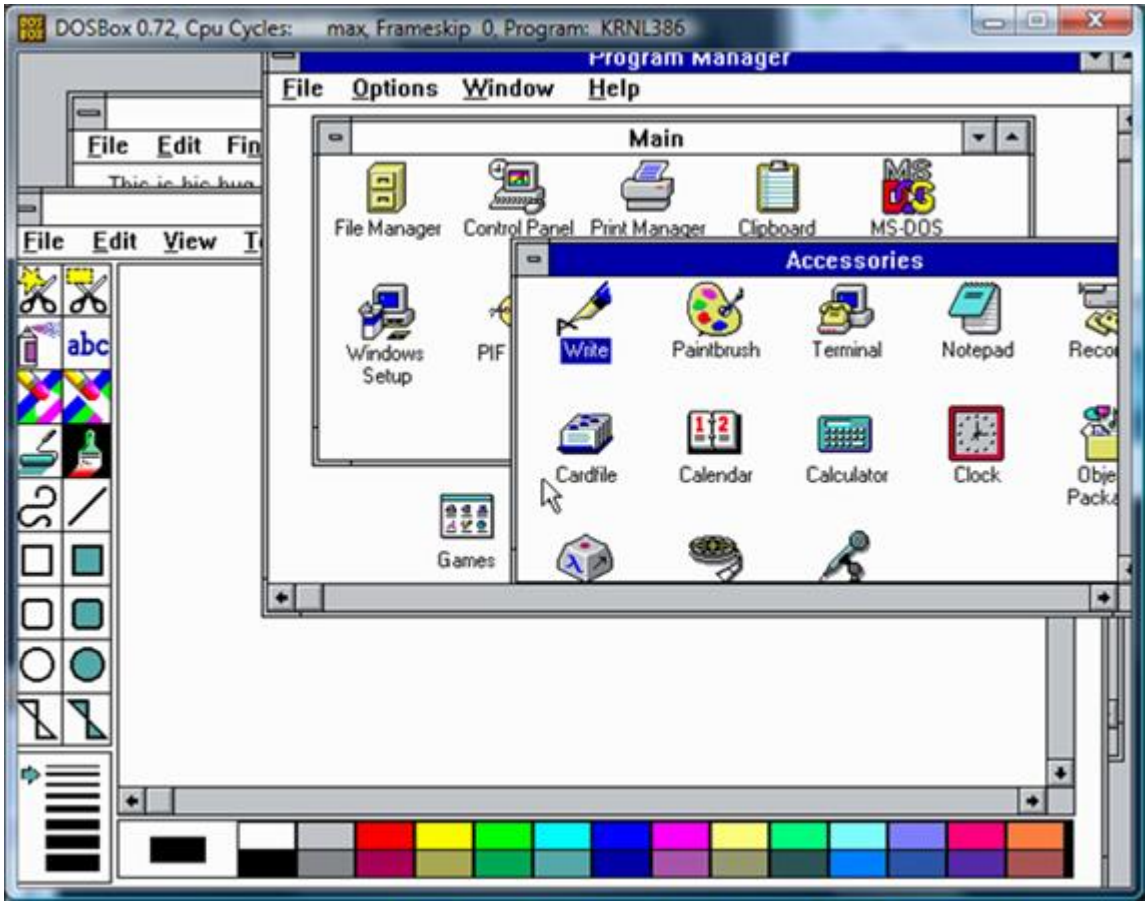
1. picture - Alan Curtis Kay

He designed a programming language called Smalltalk. This is the first but still existing pure object-oriented programming language. Nowadays newer versions of this programming language still being introduced but the main principles remain the same throughout.

He performed a pioneer's work in another field: he suggested using graphical user interface on the computers and using a special input device called the mouse. He also suggested basing this this graphical interface on using icons, menu systems and windows.

Alan Kay dreamed of a portable computer called the Dynabook in 1973. It is a book-sized computer, which consists of wireless network adapter, a good quality colour screen and a very high computing power in it. The

plan remained a plan, but he convinced the Xerox research leaders to work in his ideas. Finally they assembled a computer called Alto using the high technology items which were available at that time. It was actually a mini computer with 256 KiB of memory, mouse, and removable hard disk storage capacity. Its operating system used graphical user interface and it was able to communicate on the network; actually it was the first workstation with a modem. After the ideas on the hardware elements Kay started to design software's which can be the ancestors of the nowadays graphical user interfaces. In the MS Windows v3.1 his ideas can be found.



2. picture - Windows 3.1 desktop

Currently the OOP paradigm is considered as a kind of successful development of the modular (procedural) programming paradigm. As some researcher the OOP is considered as the fourth-generation language, others are placed between the third and fourth generation (and so it is the 3.5 generation). Rationale for the latter, and that the OOP approach, the function body is actually made up of the same control structures as in the procedural programming, but rather the difference between the two lies of grouping of functions, organizing methods.

3. 3. The principles of OOP

According to the original ideas of Alan Kay – three principles must be supported:

- encapsulation,
- inheritance,
- polymorphism.

The principle *encapsulation* says that the data elements in the program which has a cohesive relationship by a logical meaning (eq. the coefficients of an equation) and the functions related to these data (which work with

these data elements some way) must be grouped together. This unit must guarantee that the functions can be called only the data elements are filled with, and the data elements can be modified by these functions only.

This unit is called an object class (shortly class). The data elements related to a class are called fields, not variables, while the functions of a class are named methods not functions. There is such a concept as variable in an OOP language, but only the local variables declared and used inside a method body can be named by it. A field is declared outside any function bodies, usually it has a dynamic lifetime, it is a dynamic storage unit. Since a variable and a field differs from each other in several ways, it is an error if a wrong name is used.

However a function is referred as the traditional (procedural) programming language concept means almost the same as a method, but a function is never part of a class. To call a function one must simply write its name. A method is a function which is part of class, so calling a method is much more complicated in the syntactically and semantically meanings. This is also incorrect if the two terms are not used properly.

Note: at the same time (as we shall see later) for pure OOP languages the concept of a function does not exists (a function simply cannot be created only as a part of a class), so only methods can be written. Strictly speaking it is still an error to name them functions, but it should not be interpreted in a misleading way, so it often happens to use the word 'function' instead of 'method'. Another opinion is that a class level method can be called a 'function', while the instance level methods must be called 'method' certainly.

It is important to note that the object class is an abstract concept (we will see later, it is essentially a type). In other words, the existence of an object class by itself means not necessarily a real working data storage and functionality. An object class is a model, a plan. Similarly, if we have a car that contains data (power, number of seats, number of speeds, braking, acceleration, etc.) and functions (start the engine, stop, braking, cornering) written on a paper. Still we do not have a car. However, using this plan, not only one car can be made, but a lot. The process when one creates a real and working instance of an object class is called *instantiation*. During the instantiation of a class the storage units (fields) actually inserted into the computer's main memory and occupy their space. If we create multiple instances, it happens more than once. The instances are often also referred to as *objects*.

Note: it is often mixed as the object class (class) and the object (instance). It is a common error (incorrectly) using the word '*object*' instead of the word '*class*' (for example, "to design an object").

The principle of *inheritance* says when an object class is already finished (including its fields and methods) and other class must be created with similar data storage capacity and functionality – one might use the finished class as a starting point. In this case one might declare the original class (referred by its name), and it the new class should take over all the fields and methods from this original class without physically copying them in the source code.

The already finished class (the original one) is called as base class (parent class, superclass) furthermore. The new class (which is under construction) is called derived class (child class). So the derived class contains all the fields and methods declared in the base class. It is not a simply copy-paste operation. Since the connection between the two classes is declared in the source code, when a modification is made in the base class (new fields are added, new methods are defined, or modified) the derived class automatically received these changes when we recompile the source code. This is common as we usually make error corrections in the base class or it is simply extended with new features. According to this principle at the compilation process the derived class immediately and automatically take over these changes.

The principle *polymorphism* is the most difficult to understand, but it is a very important one. Basically it says that to the functions, fields, classes multiple meanings can be given in the very same source code. This can be interpreted in the terms of OOP that a descriptive declaration (an interface) can be defined through which the operation of an object can be defined without the actual processes behind the function are known.

We can imagine the situation when there is a central control unit (a general) who can direct his units on the battlefield by simple commands such as "go forward", "turn left", "stop". In other words a 'general' can manage any kind of unit who understand (includes) these three commands, whether they are soldiers, tanks or war planes. It is obvious that the implementation of these commands are completely different in the different kind of units, but the general won't care about it. For him these units are intelligent individuals who know themselves and know how to execute these orders. They accept commands from outside, but nothing else must be known about them for the outsider commander.

The polymorphism allows developing very high level of codes which can cooperate with a wide variety of data types effectively. A sorting algorithm may be able to arrange any type of data set according to the principle that two data items can be asked to compare themselves and calculate which is the bigger one (whatever the meaning of this term). If the order of the two is not correct, the sorting algorithm can ask the collection (array, list) to swap the two elements.

The implementation of this principle causes the most complex and complicated improvements. To fully understand this principle the late binding, the type compatibility, abstract classes, methods and other terms must be introduced. The significant part of this book is about these topics.

The implementation of these principles is not regulated, so syntactical differences arise between the different OOP languages. In addition many OOP languages includes other useful enhancements. C # is one of the most extensive capabilities OOP language which contains very clean syntactical and semantical solutions. When one becomes familiar with this language deeply, using other OOP languages similar or exactly the same solutions can be found. So studying the C# OOP capabilities we can get a very good base knowledge on this subject.

4. 4. The classification of imperative languages

The object-oriented programming supposes the existence of the three principles in the chosen programming language. These principles are compatible with the traditional imperative, procedure-oriented programming language principles. It is very common that an existing version of a traditional programming language extends with the OOP principles, include these into their procedural ones. The resulting programming language carries both procedural and OOP approach.

According to these, we divide the imperative languages into three levels:

Pure procedural programming languages: OOP principles do not exist, they contain only the procedure-oriented approach. Such languages are eq. Pascal and C. In these languages the concept of the “global” variables are interpreted as well as the term function. These global variables are accessible and modifiable by all the functions embedded into its module. Unfortunately, this possibility may encourage programmers to store a substantial part of the data elements in this way, avoiding passing them as function parameter, and the use of the return value of the function. These languages are typically was designed before the OOP principles born.

Pure OOP languages: the design of the language includes all the OOP principles and even some of the traditional concepts are completely thrown out the design. Accordingly, there is no function in these languages, because applying the encapsulation principle forces it to insert all the function into a class, therefore each function is converted into a method. There are no global variables, because each of these variables is also inserted into classes so they become fields. Nevertheless, some difficulties can appear of course, because we will see that the extremes are formed inconvenience. But the small disadvantages can be opposite with serious advantages, these languages have proven to industry challenges as well. Their success demonstrates the benefits of strength. These languages are such as Java, C#.

OOP supportive languages: an existing traditional programming languages are typically known by many programmers, a large amount of source code were developed in this language already. To keep the compatibility and the knowledge it is seemed worthy to modify the syntax of this languages by extending with the OOP principles. To develop computer programs in these languages both "hybrid" approach can be used. In other words, at the same time traditional functions, global variables can be created side by side classes, fields, methods. A competent, experienced programmer’s hands this language is very efficient tool. A novice programmer, however, might be confused by the contradictory syntax, difficult to choose which paradigm is suitable to apply at an appropriate moment. In addition, subsequently inserting the OOP principles into the original syntax made it difficult to use. These languages are eq. Delphi, C++.

With the help of the OOP principles one can cover all the procedure-oriented possibilities, with small compromises. However the clear syntax makes these languages more powerful and easier to use. In these languages less mistakes can be made during developing a computer program.

5. 5. A simple example for an OOP problem

Let suppose our program works with rectangles. One of the edges of any rectangles is always horizontal. The rectangle is characterized by the coordinates of the leftmost lower corner (x,y), and the length of the horizontal and the vertical edges (side a and side b). The program stores these data elements but beyond these it must support the calculation of the area and perimeter values of this rectangle, then from an arbitrary x,y point the program must calculate when it is falls inside the rectangle or not.

On a traditional programming style to store a rectangle we use records:

```
struct rectangle

{

public double x;

public double y;

public double side a;

public double side_b;

}
```

Perhaps we could make a record for the point as well:

```
struct point

{

public double x;

public double y;

}
```

Finally we should make the necessary functions:

```
public static double perimeter(rectangle r)

{

return (r.size_a + r.size_b) * 2;

}


public static double area(rectangle r)

{

return r.size_a * r.size_b;

}


public static bool is_inside(rectangle r, point p)

{

return (r.y <= p.y && p.y <= r.y + r.size_b &&

        r.x <= p.x && p.x <= r.x + r.size_a);

}
```

A possible use of the code, a sample Main function:

```
public static void Main()
```

```
{
  rectangle t = new rectangle ();
  t.x = 10;
  t.y = 12;
  t.side_a = 22;
  t.side b = 4;
  //
  double k = perimeter(t);
  double t = perimeter(t);
  //
  point f = new point();
  f.x = 12;
  f.y = 15;
  bool inner = is_inside (t, f);
}
```

Note that the relationship between the data structure ("struct rectangle") and the functions working with them are very loose. We can find the relationship between them as all the functions have a parameter with the rectangle data type. Let imagine that if these code blocks are scattered in the source code, and the data structure is modified - then further modification requests arise in several locations in the source code. Our next observation is that the structure of the rectangle type must be known by the processing functions. Confusing that we don't know the "perimeter" function will calculate what type of geometrical object – only the parameterization can lightening us (as we examine the type of the parameter we will recognize that this calculates the perimeter of a rectangle).

Let's look at the same example in OOP style. The keyword *class* allows us to encapsulate the data and the functions into one unit:

```
class rectangle
{
   protected double x;
   protected double y;
   protected double side_a;
   protected double side_b;
   //
   public double perimeter()
   {
    return (side_a + side_b)*2;
   }

   public double area()
   {
```

```
    return side_a * side_b;
   }


   public bool is_inside( point p )
   {
    return (y<=p.y && p.y<=y+side_b &&
                x<=p.x && p.x<=x+side_a);
   }
   //
   public rectangle(double pX, double pY, double pA, double pB )
   {
       x = pX;
       y = PY;
       side_a = pA;
       side_b = pB;
   }
}
```

The data fields in the *'class'* are placed close together to the function blocks (encapsulation). The functions become part of the data structure, there is no need to receive a rectangle as a parameter of these functions, as they all can access and work with the data fields directly. If we would have two rectangles, the first would be stored in the fields of the class, the second would be a parameter of a function. The functions (to access the fields) do not use the *'static'* modifier. The last *'rectangle'* function is a special one, it copies the parameter values into the fields, creating the initial state of the instance. This will later be called '*constructor'*. The same *Main()* function written in OOP style is as follows:

```
public static void Main()
{
    rectangle t = new rectangle (10,12,22,4);
    //
    double k = t.perimeter();
    double t = t.perimeter();
    //
    point f = new point();
    f.x = 12;
    f.y = 15;
    bool inner = t.is_inside( f );
}
```

The first line creates a rectangle instance ('t'). The *'new'* operator allocates the necessary memory block for the fields. Behind the memory allocation the calling of the constructor can be seen. The four initial values of the fields are given. The rectangle instance ('t') contains not only fields but also functions. To call them we must

write down the name of the instance (‘t’), then dot operator, and the name of the function (eq. 't.perimeter()'). It means "calculate the perimeter value of the instance 't' using the fields of ’t’". Inside the ‘perimeter()' function the identifiers such as ‘side_a’ and ‘side_b’ means the fields of ‘t’.

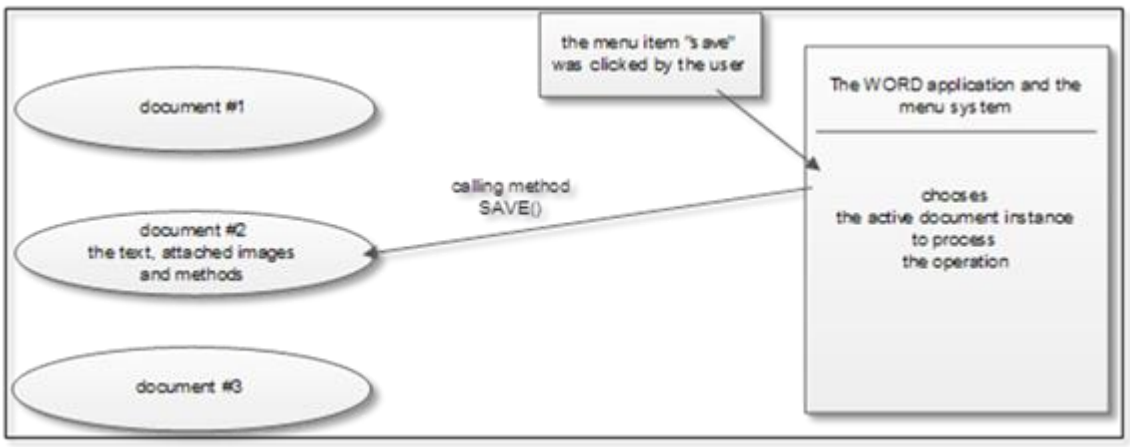
The code written in this style - for those who are familiar with OOP – is much more readable. The scheme "make an instance and see what it can do" is typical. In this case when we want to know what a rectangle can do write 't' and a dot, and the Visual Studio (shortly VS) will list the fields and functions on it owns. The development tool knows that the dot operator means we want to make a reference to a field or a function of this instance. In the traditional (structured) programming style similar assistance could be created, the development tool could gather all the functions with a parameter of this type but obviously it would need much more time and energy. Later we will get to know other ways to organize codes, code snippets into groups[4].

A programming language has a lot of features. It is important that the language would have useful basic types (numbers, letters, text, boolean, etc). To write expressions easily – we need several useful operators. The semantically working of the program control structures must be familiar to get used to using them quickly. The nesting block must be descriptive and well-organized structures. These are the basic requirements, which when in place, programmers can begin to develop their functions and own modules. Additional requirement (prerequisite for the success of a language) is to have rich collection of built-in functions. Thus, programmers can focus on higher-level tasks, with the help of the well documented, handy basic functions. When the collection of functions contain very high amount of functions it may be unfavourable, because the programmers are unable to remember the name of so many functions. If you must search and read documentation or manual before using a function many times – it decreases your performance. Win32 programming environment consists of more than 2000 functions as the base on which the development may start. In the case we have so many functions -- the function names are not helpful enough. Imagine when a programmer wrote a program for the Windows platform, and he wants to change the shape of the mouse cursor from the default to sand clock when the program performs a heavy counting task. What is the name of the appropriate function to do that? SetMousePicture? MouseSetCursor? ChangeMouseIcon? (The correct answer would be the LoadCursor + SetCursor function pair.)

The Microsoft.NET 1.0 Base Class Library contains more than 7,000 classes, many functions per class. To handle a library with this size -- OOP approach is needed.

6. 6. Data hiding

The compilation of a project basing on a large codebase (possibly covering several programmers work) is done by developing classes with separate tasks. The classes contain data items (fields) and a number of functions (methods), with the help of them the instances could process tasks. For example, when editing a Word document an object might store the text, the name of the file and other information (date of the last modification, etc). A function of this object can be 'save()', which can write and store the document text to the disk. The object itself stores the name of the file, so calling the function without any argument is imaginable.



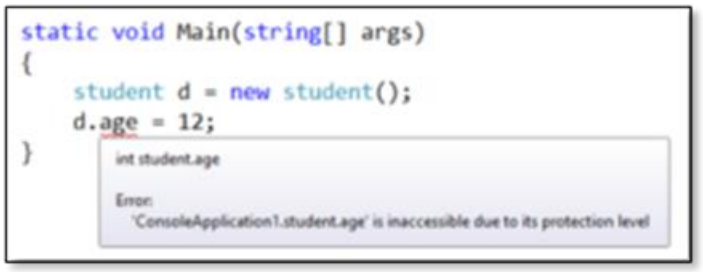
The methods of the object are working with the data fields continuously. In the very first step we must learn how to create object classes with fields, instantiate them in the outside world, and then put data into a data field!

Our first job is to store a primary school student data (name “John Smith”, age 12 years, class 7.C)!

```
class student
{
  int age;
  string name;
  int grade;
  char subclass;
}
```

The code described above is more of a "record" than object, since there are currently only fields. We will need a code to instantiate and fill the fields with data, and generally to make the object do something.

We will write this code inside the function ‘Main()’, into a separate class. (Later this will no longer be detailed, but also later in this document we will put the function Main into separate class.)



Notice that the code is already bugged. The VS shows that the object 'd' has no available field 'age' (“student.age is Inaccessible due to its protection level”). The new concept that we need to learn is the *protection level*.

Three[5] levels of protection available in the world of OOP:

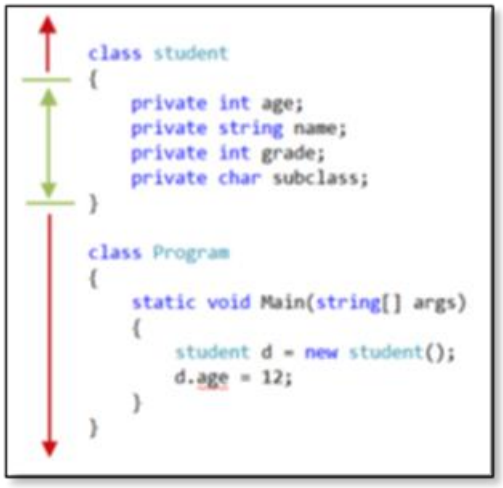
private,

protected,

public.

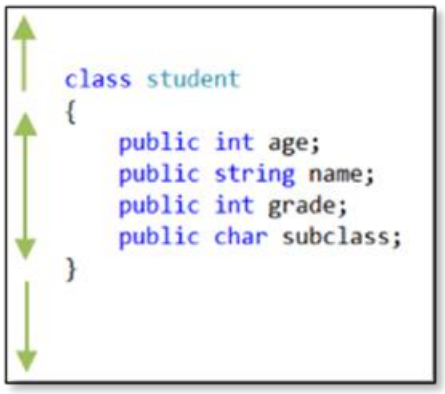
First, it should be understood that the protection level are **scope modifiers**! The scope, as we remember the property of and id that specifies which part of the source code to be used for the id, at which points the compiler recognizes the id.

The default security level is *private*. If you do not select a protection level explicitly, the private will be selected by the compiler. This level of protection means that the fields (and methods) are accessible only inside their classes:



The class “Program”, and its function “Main()” are outside of this area, so the private field cannot be accessed there.

What we need is the protection level “public”, which guarantees the access not for the container class but for any methods in any classes, so the function “Main()” as well:



With this knowledge the main program can be written:

```
public static void Main()

{

  student d = new student();

  d.age = 12;

  d.name = "Kiss Lajos";

  d.grade = 7;

  d.subclass = 'C';

}
```

6.1. 6.1. To protect the value of a field

The values of a field of an object instance are given by the outside world. This outside world can change these values substantially at any time. The methods that work with the data of the field must check the correctness of these values every time they start. Assume that in our school there are only A, B and C subclasses. So the field “subclass” there can be only these three letters. To avoid further problems lowercase letters are not permitted to be in this fields. The use of the “enum” structure can be a solution, but it would be a problem if we would try to use this class definition for two different schools, when one school has only A, B and C subclasses, the other

contains D and E subclasses as well. Back to the original problem: we want to enable only A, B and C values for this field.

Our goal is clearly not achievable; if the field is public and is accessible to the outside world, the world outside can substantially change the value of this field at any time. Where we would have methods to work with this field, they must check the actual value of this field every time. These additional operations are significantly slower than the speed of the code run. It is obvious that if the field is unchanged until the last check, a recheck is unnecessary.

There are no good solutions for this problem in the traditional (procedural languages) approach. A programmer can write into the documentation that "pay attention for the value of this field". In principle, we can provide a function to put new value for this field, which checks whether the new value is suitable or not. But we can evade using this function and write the new value directly into the field.

```
static bool setSubclass(student p, char subClass)

{

  if (subClass != 'A' && subClass != 'B' && subClass != 'C')

      return false;

  else

  {

     p.subclass = subClass;

     return true;

  }

}

student d = new student ();

setSubclass(d, 'C');

student k = new student ();

setSubclass(k, 'X');
```

6.2. 6.2. A solution with methods

In the world of OOP, however, there is a solution to the problem, which is based exactly on the level of protection. The protection level of the field is not public this time, to prohibit the outer world from changing its value directly. The compiler only checks the basic type of the fields, so that any character value is accepted. So the defence can be bypassed. Set the protection of the field to private. Thus, the outside world cannot write incorrect value into the field directly, but unfortunately correct value neither, because the access to the field access is totally disabled. So create function 'setSubclass' OOP equivalent as a method (and do not use the 'static' modifier this time):

```
class student

{

  public int age;

  public string name;

  public int grade;

  // cannot be accessed from outside as it is 'private'

  private  char subclass;

  // can be called from outside as it is 'public'
```

```
  public bool setSubclass(char subClass)
  {
    if (subClass != 'A' && subClass != 'B' && subClass != 'C')
        return false;
    else
    {
       this.subclass = subClass;
       return true;
    }
  }
}
```

The protection level (access level of the method) is ‘public’. The protection levels for methods are the same as described for the fields. For a method the protection level defines the places where the method can be called. A public method can be called from anywhere in the source code, not only from the container class. This method returns a boolean value, which indicates if the write to the field was successful or not. This method needs no student argument, as it is the part of the class of a student object instance. The actual student object is identified by the keyword ‘this’ inside the body of the method. So the ‘this.subclass = subClass’ means: put the value of the argument into the field.

Therefore direct writing into the field is no longer possible from the outside world (as it is ‘private’), but the call of the public method is available.

```
student d = new student();
// cannot write this because of the protection level:
// d.subclass = 'X'
// but this is working:
bool succ = d.setSubclass( 'X');
```

The help of the compiler cover the disable of the direct writing to the non-public field. So the modification of the field can be initiated only by the call of the method. This method, however, will always check the value coming from the outside world, and only the corresponding value is accepted and is put into the field. Therefore, the value of this field is always[6] correct; the other methods must not check it all the time.

While the setting is secured to the outside world (writing), do not forget the possibility of reading! At present, for the private access, not only to the direct writing, but also the direct read is also prohibited.

The solution to this is the writing of methods again. As the methods are parts of the class, the access to the private fields is guaranteed. The method can be public, so the outside world it is callable:

```
class student
{
  public int age;
  public string name;
  public int grade;
  // cannot be accessed from outside as it is 'private'
  private  char subclass;
```

```
  // can be called from outside as it is 'public'
  public bool setSubclass(char subClass)
  {
    if (subClass != 'A' && subClass != 'B' && subClass != 'C')
        return false;
    else
    {
       this.subclass = subClass;
       return true;
    }
  }
  // reading the actual value of the field
  public char getSubclass()
  {
    return this.subclass;
  }
}
```

The main program might use this new function immediately:

```
student d = new student();
//
bool succ = d.setSubclass( 'X');
char actual = d.getSubclass();
```

It is advised for the Hungarian programmers not to use Hungarian names (words) for naming the fields and methods for several reasons. The use of the Hungarian special vowels like ‘é’, ‘ő’, ‘ű’ etc. are possible but also contraindicated. The naming of this kind of methods based on the English word ‘set’ and ‘get’. For their meaning a method which writes new values to the field XXXX is usually named by setXXXX (for example setAge, setSubclass, setName, etc). The methods which reads the values are named getXXXX (for example getAge, getSubclass, getName, etc). Let us see the solution for the age field of the student class. The restriction to this field is that the value (age) must between 6 and 18:

```
class student
{
  private int age;
  // writing
  public bool setAge(int value)
  {
    if (value < 6 || value > 18)
        return false;
    else
```

```
    {
        this.age = value;
        return true;
    }
  }
  // reading
  public int getAge()
  {
    return this.age;
  }
}
```

The part of the main program is:

```
student d = new student();
d.setAge(12);
//
int actAge = d.getAge();
```

The getXXXX and setXXXX though only a naming convection (a tradition), but this helps a lot as knowing the name of the field is enough to guess the two methods name. In the language Java it is more important. There the concept of *property* (see later) is unknown, but the same suffix in the get… and set… method pair can be recognized by the developer tool, and handles them together as a property.

6.3. 6.3. Error indicating

In the world of OOP is not common that the setter (setXXXX) method is defined as a bool function. The problems would arise with that will be discussed later in the chapter exception handling. That will describe the problems, so we must read for the effective and thorough understanding. We just note that in the OOP environment, when a method is called to perform a task, but involuntarily by the method (usually by the fault of the outside world) cannot be performed, it is indicated by the throwing an exception. For now, it is enough for us not to return false, instead we must use the *throw* keyword to indicate the error. Until the deep understanding of the exception handling we suggest two methods to use. The first is:

```
throw new ArgumentException("… reason of the problem …");
```

and the second form:

```
throw new Exception("… reason of the problem …");
```

The 1st form is used when the (bad) value of the argument is the reason of unsuccessful execution of the function; the 2nd form is used for any other reasons. The "reason of the problem" describes the actual and exact reason for failure.

Think of the 'throw' statement as the alternative of 'return'; when it is executed, no further instructions will be executed from the function body, it will return back to the caller code (such as 'return'). The difference is that the caller remains in the execution state after using statement 'return', the program continues. Meanwhile by the statement 'throw' the program recognizes that an error occurred and does not execute any further statements at the caller side as well, it returns immediately to its caller as well (skipping all the remaining instructions). This continues until the execution returns to the function 'Main()' back to the original calling code, which in this case terminates[7]. So executing a 'throw' because of the skipping of the remaining statements finally causes the termination of the whole program. The given description of the problem usually can be seen by the user, who can inform the developer.

```
class student
{
  private int age;
  // writing
  public void setAge(int value)
  {
   if (value < 6 || value > 18)
        throw new ArgumentException("incorrect, 6..18 is accepted");
    else
        this.age = value;
  }
  // reading
  public int getAge()
  {
      return this.age;
  }
}
```

Accordingly, the 'setAge ()' method becomes ‘void’ instead of 'bool'. If the checking detects a problem, then a 'throw' indicates that the operation has not been executed. An assignment statement is not necessary to put into an 'else' branch; as if the 'throw' is performed, then no other statement will be executed. This control can only reach this assignment if no 'throw' was executed, so the value is appropriate.

6.4. 6.4. Protected instead private

There is a 3rd protection level among *public* and *private*, which has not been mentioned yet. The level *protected* is between them, but to understand the concept of the child classes is necessary. The derivation of a base class is called the child class. The child class inherits all the fields and methods of the base class. This will be discussed more thoroughly later, until then let's look at a simple example:

```
class student
{
  private int grade;
  //
  public void setGrade(int newGrade)
  {
    if (newGrade < 1 || newGrade > 8)
       throw new ArgumentException("only 1..8 can be accepted");
    else
       this.grade = newGrade;
  }
```

```
// ... cont ...
```

```
class advStudent : student
{
    public void reachNextGrade()
    {
        if (this.grade == 8) promNight();
        else this.grade++;
    }
    public void pr
    {
        // sing so
    }
}
```

int student.grade

Error:
'ConsoleApplication1.student.grade' is inaccessible due to its protection level

The *advStudent* class is a child class of the *student.* Consequently it contains all the 4 fields (eq. grade and subclass), and the *setGrade* method. It wants to introduce a brand new method which is not derived from the base class: the *reachNextGrade()* function. In this method we want to use the derived *grade* field, but the VS indicates a syntax error, according to a violation of the protection level. The private field is derived by the child class, but its scope still won't extend to this area. It is a paradoxical situation, but later it will be reasonable. For now, we simply note that if in the child class we want to use the derived fields directly, the *protected* level is needed. The protected access extends the scope not only the original (base) class but to the child classes as well, but into foreign code lines, such as foreign classes (the function Main() in a different class) does not.

The direct access to the fields is a confidential matter. The one, who can access the field, can set any value of the given type to it, which is not surely suitable for the object as well. So the OOP world is divided into three areas in terms of reliability.

The 1st (innermost) area of trust contains only the class itself with all the methods defined inside this class. This is the private level. The field's value cannot be read or write by the codes in the other areas, only through the public methods of this class. They carefully check every values coming from the unsafe areas before accept them and put them into the private fields.

The second area is the *protected* level. All the child classes (and their child classes) are here. A protected field defined in the base class is accessible in the methods of the child classes directly. This protection level is the most used, as the read/write operation of the field are fast through direct access. Using public methods the read/write operations are slower. The child classes are independent classes, so they must be responsible for their own rules. When they ruin the value of the field – it is their problem. We will see later that the child classes

have the right and chance to redefine (rewrite) the derived methods, so if a child class puts incorrect values to a derived field, and so a method of our own (originally defined in our base class) makes an error – it is still not our fault. We can say "why you won't redefine this derived method to alter its operation against this originally unaccepted value".

For this reason the use of the *private* protection level is very rare, as we don't want to hide the fields against the child classes unless for a very well-founded reason. The most common level of protection is *protected*, and for the unprotected ones is the *public*.

The *public* means the lack of protection. The public fields can be accessed by everyone, its value can be changed at any time. So the values of the public fields are unreliable and should be checked every time we want to use it.

6.5. 6.5. Why is the private the default protection level?

When not writing protection level explicitly it is equivalent to the protection level *private.* This is the default level. Why is that? Why is not the *protected* or *public*?

Any of the choices would have pros and cons. The first reason to choose *private* is that the most frequent reason for not defining the protection level directly is that the programmer simply forgets about it. Then automatically the default protection level, the highest, the strongest *private* comes up. The developer of the class won't detect this for the very first time, as the methods inside the class can still access these fields directly. However, the developers of the outer world code immediately detects this "forget" as they has no way to read or write this field. Their first step is to alert the "forgetful" programmer about the problem, thus gives an opportunity to think about the protection level, possible ease this strong protection.

Think a moment about another case, when the default level would be the *public*. In this case when a programmer won't define exactly the protection level, he still won't detect this as the methods inside his class still can access these fields, the same as the *private* case. But the developers of the outer world could read/write these fields as well, but they won't alert him about this "problem", instead of this they quietly enjoy the benefits of this forgetfulness.

In the Delphi language there is a phenomenon. In Delphi the fields and methods of protection levels effect only outside the source code of the module. If we put two classes (the base class and a foreign class) into the same source code, the methods of the foreign class can access the private (and protected) fields as well in the other class without any problem. The same is true if this source code contains the function "Main", in this code we can reach all the hidden fields of these classes. However, when we move (refactor) the foreign class (or the function "Main") into a different source code, it stops working and syntax errors start to arise, the compiler start to indicate violation of the protection levels.

The reason is that Delphi assumes that one source code is written by only one developer. In other words any code statements are reliable within the same source code, the protection must not apply. As two source codes might be written by two different developers, so the protection must be enforced. The languages are not good whose syntax contains such an exceptions, but we must say they can increase the effectiveness. The "unreliable" code can gain access not to the protected fields but to the 'get' and 'set' methods only (slow execution speed), but the reliable codes can read/write the fields directly (high speed). Delphi compiler extends the "trusted code" concept to all the codes within the same source code.

6.6. 6.6. Property

The concept of the "property" is in connection with the protection levels. This concept did not comes from the principles of the OOP itself, so there are OOP languages which knows nothing about this concept, in other languages the concept exists but not with the same syntax. We will be getting familiar with the C# language version of "property".

The "property" is a syntax candy. It is for a developer to feel the comfort of a well-defined language. It is for writing a common used code piece in an easier to read or write way, and make that more useable. It is all about, that a hidden (protected or private) field we usually make get and set methods to allow the access for the outer world. We (in the outer world) must use these methods, use brackets for reading and writing the value of that field. We cannot use the assignment operator for writing, instead we must call a method to do the assignment,

and the new value to write must be passed as an argument. It is a very uncommon syntax for these very common operations.

Accepting the "virtual field" concept is the easiest way to understand what a property is. This is "virtual" because it looks like a real field in syntactical way. It has type; we can read its value or assign a new value to it with the common assignment operator.

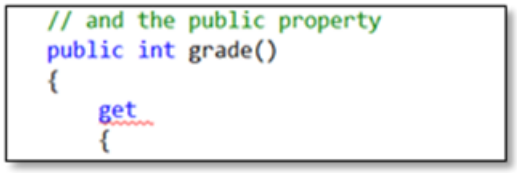
However, the property is not a real (physical) field. The property has no memory storage, is not in the memory space of the object instance, it is not part of this instance in this way. So this point of view it is like a method. In fact, as we will see in the background a property is really method (actually usually two methods). Only the syntax of calling these methods makes it similar to the fields.

```
class student

{

  // the heavily protected field

  protected int _grade;

  // and the public property

  public int grade

  {

    get

    {

      return this._grade;

    }

    set

    {

       if (value < 1 || value > 8)

            throw new ArgumentException("only 1..8 can be accepted");

       else

            this._grade = value;

    }

  }

  // ... cont ...
```

The syntax of the creation of property looks like the syntax of the creation of a simply field. The protection level is typically *public*, because it is created for the outside world (but protected and private properties are reasonable as well, but usually contains the *set* part only). It is followed by the type of the property (int) and its name (grade). If we would insert a semicolon at this point, we would create a real field:

```
  public int grade;
```

In case of creating a property we won't finish this definition with a semicolon at the end of the line. But we won't insert any brackets here as well:



In this case not a property but a method will be constructed with an empty parameter list and a body. But in that case keyword "get" won't be acceptable there, a body of a method cannot be split into two parts (get and set parts). The property therefore is not a field and not a method; it has its own syntax to create.

The body of a property can be divided into two parts, the *get* and the *set* part. The *get* part is responsible for the reading of the virtual field's actual value, and the *set* is for the change of it. In other words: when the outer world tries to read the actual value of a property – the *get* part will be executed; and when it tries to assign a new value to the property, the *set* activates. Inside the body of the *set* part we might refer to this new value with the keyword the *value*.

```
student d = new student();

// writing the virtual field -> set -> (value = 7)

d.grade = 7;

// reading the virtual field -> get -> (returns 7)

int nextYear = d.grade + 1;
```

In this case the property behaves as it would be a real field, the same syntax for reading and writing can be used. When we want to assign a new value to this, after the instantiation the common assignment operator (=) can be used. As the property sits on the left side of this operator, so the compiler will recognize the write operation, so the *set* part will be executed. Inside the *set* part the keyword *value* will represent the integer value 7. First the *set* examines if the value is out of the [1..8] interval. As it is not (in this case), the new value is stored into the protected real (physically exists) field.

Later, when we tries to read the value of the property, the *get* part is executed. It returns the previously set value from the real field (_grade) which still holds and stores the value of 7.

It is typical that the name of the property and the name of the real field are very similar to each other. Of course cannot be the same, since two identifiers cannot be the same inside the same scope. The similarity can be maintained in several ways. Traditionally the name of the real (hidden) field begins with an underscore, and the public property's name is "prettier". Alternatively the name of the real field begins with uppercase, while the property name begins with lowercase. Similar solution is that the name of the field begins with letter "f" (field), and the property's name not.

We must highly take care of the property, to avoid mixing the two identifiers. Inside the *set* or the *get* we must refer to the real field, not to the property! The set writes the value to the field, and the get reads (returns) with the value of the field. The name of the field begins with an underscore. Consider the following (bad) code:

```
class student

{

  // the heavily protected field

  protected int _grade;

  // and the public property

  public int grade

  {

    get

    {
```

```
      return this.grade;
    }
    set
    {
       if (value < 1 || value > 8)
            throw new ArgumentException("only 1..8 can be accepted");
       else
            this.grade = value;
    }
  }
  // ... cont ...
```

According to the code when the outside world tries to read the value of property *grade* (get operation), then we must read and return the value of *grade*. This is actually a property (the *grade*), and its value must be read by its get operation. So again, the get will be executed. This 2nd get will read the value of a mysterious *grade*, which is a property, its get will be executed… and so on. It is called recursion, this case it is an infinite recursion.

The reason of this infinite recursion is that inside the get (the reading) part of the property we read the value of the same property (the underscore is missed).

The similar problem is inside the *set* part. When the *value* is correct, the new value is to be written to "this.grade", which is the property itself, and writing into this causes to execute the *set* again and again. This is also an infinite recursion, and the reason is the same (missing underscore, bad identifier use).

Unfortunately, the compiler cannot detect these two errors, as the code (body of get and set) is syntactically absolutely correct.

6.7. 6.7. When we might think no protection is needed

It is not necessary to apply protection to a field when the value of a field can be changed at any time, but the rules of acceptable values can be fully described by the type of the field. The simplest example is a boolean-type field. This, according to the boolean type itself, only the true or false values can be assigned. It is nonsense to make a property like this:

```
// the protected field
protected bool _is_enrolled;
// the public property
public bool is_enrolled
{
  get
  {
     return this._is_enrolled;
  }
  set
  {
    if (value != true && value != false)
```

```
        throw new ArgumentException("only true/false accepted");

    this._is_enrolled = value;

  }

}
```

Since the compiler always checks if just true/false can be define as the new value of the boolean property, the previous code adds no additional protection. Furthermore the use of the property (methods) significantly slow down the read/write operation speed, as writing to this property uses extra costs such as a method call, then a useless but slow test of a condition (if statement). Reading the value of this property costs an extra method call, which is also slower than reading a field directly.

```
// need no protection

public bool is_enrolled;
```

Another easy example when the type of the field is an enum, so the acceptable values are described in the definition of the enum.

6.8. 6.8. Writeable once fields

It is common that we want to enable to write the value of a field for the outside world for the very first time, but later the change is not accepted. We can name it as a write-once-read-many field.

The write-once fields can be implemented in many different ways – some programming languages provide support for this special issue. The following is not a typical solution in the C # language, as it also includes special constructions for this problem - but the concept of the property can be deeper understanding through this. (The conclusions are interesting, and the method is adaptable to other, similar problems.)

First we must decide on the protection level of the write-once field. It is easy to see that the *public* is useless this time, as a public field can be read/write unlimited times for the outside world. Fortunately both *protected* and *private* is suitable for this time.

On the other hand, let us see an example: consider the value of the student's age. Let it be the field whose value is allowed to be set only once, and then later every year the 'beOlder()' method must be called. This is a good case because we store the actual value of the age in an integer value (e.g. int), but we know that a negative value is meaningless in this field: thus, there exists an initial value, which clearly indicates that the field has already been set or not!

```
class student

{

  // -1 is the special initial value

  protected int _age = -1;

  // the public property

  public int age

  {

    get

    {

      return _age;

    }

    set

    {
```

```
      if ( age == -1) // not set
      {
        if (12 <= value && value <= 90) _age = value;
        else throw new ArgumentException("only 12..90 accepted");
      }
      else throw new Exception("the age is already set");
    }
  }
  // ... cont ...
}
```

In this case a special initial value (-1) exists which is unacceptable according by the actual rules. Inside the *set* we check if the field still has its initial value. When it still has, and the new value is acceptable (12..80) we store this new value to the field. During the next attempt of writing the *set* will detect the improper situation and will throw an ArgumentException.

Unfortunately there is a possible bug in the previous solution. We can read the value of the property *before* we set the value of it. In that case we will receive the initial value (-1). It is easy to correct but it is also easy to forget:

```
// the public property
public int age
{
  get
  {
    if (_age == -1)
         throw new Exception("still not set");
    else
         return _age;
  }
  // ... cont ...
```

The situation is a little bit harder when our field has no special initial value. For example a graphics object has X,Y coordinates, and they can be simple anything during the execution of the program. It is useless to start with the value of -1, as later during moving the object it can be -1 again. In this case the property will believe that it is the first time (again) and will allow the writing of the field directly again.

We can use an additional boolean field to store the set/notset information. Its true value indicates that it still is not set, its false value means that “already set”.

```
class graphicalObject
{
  protected bool _X_set = false;
  protected int _X;
  public int X
```

```
  {
    get
    {
      if (_X_set == false)
            throw new Exception("still not set ");
      return  X;
    }
    set
    {
      if (_X_set == false) _X = value;
      else throw new Exception("already set");
     }
  }
}
```

Obvious that this solution method is more universal, and is still simple. Unfortunately it is so impractical as an extra field is needed and must be handled correctly, and it is also bad according to the memory size. The most appropriate solution for the write-once-read-many fields is presented in the “9.8 The real write-once fields” chapter.

6.9. 6.9. Read only fields

The read-only field is an everyday practice in the world of OOP. A read-only field describes one characteristic value of the object instance without enabling to change that directly. Its value usually still can be changed by an operation of the object (eq. using a method). An easy example is the size of a vector (t.Length), size of a list (l.Count) or the CapsLock boolean field of the Console class which indicates whether the CAPS LOCK key is on or off.

For the read-only fields, the *public* protection level cannot be used as the outside world has an opportunity not only for reading but also writing. The *private* and *protected* alone also does not provide a solution, because they take away the possibility of writing but also reading.

The solution is given by the property concept again. We need to know that it is not required for a property to contain both parts (both 'get' and 'set'). There is a property that has only one of the two parts. Obviously there is no sense of a property which has neither part.

Let the example to demonstrate the read-only field is a petrol tank of a car object. The current amount of petrol in a car can be read, for example, by the computer of the car. The writing of this property, however, is illegal because it would be ambiguous: the 'skoda.benzin = 30' might mean the "let the amount of petrol be 30 litres" or "increase the amount by 30 litres". In the first case it is not clear how much fuel had to pour into to reach 30 litres. In the second case it is not sure whether there was enough free space in the fuel tank to hold another 30 litres. We can continue our philosophical speculation, but let us accept as a fact that the class ‘car’ is designed as the following: the field describes the amount of patrol is a read-only one! Let's see the solution:

```
class car
{
  protected double _petrol;
  public double petrol
  {
```

```
    get
    {
      return _petrol;
    }
  }
  // ... cont ...
}
```

The outside world might read the value of field ‘petrol’ but cannot modify its value:

```
static void Main(string[] args)
{
    car p = new car();
    p.petrol = 12.4;
    Console.WriteLine(p.petrol);
}
```

Changing the field’s value is not possible. The VS also indicates of the property cannot be on the right side the assignment because of the lack of the 'set' part - while the read operation is correct one line below. The "*Property or indexer car.benzin' cannot be assigned to - it is read only*" error message is displayed.

The problem of the read-only field is closely related to the write-once fields. The ‘set’ part of a write-once field works only once, on the second and subsequent calls causes an error. In the “9.6 Write-once fields” chapter we solve the problem without any 'set' part, with a help of another method. In that case we skip the checking of ‘is it already set’ in the *get* section as for another reason it is guaranteed that it is always set when a *get* part could be reached.

6.10. 6.10. A more effective solution

In the programming language *Delphi* there is a better solution than in the C# to handle the property. In C# the main problem is that the *get* part behaves as a method, in the background it is a slow activity - and usually contains only a “return” statement which gives back the value of the non-public field.

If we think about it, the *get* allow a direct access to the non-public field for reading – but slows down the execution speed. In Delphi it is possible to write a *set* part (a method) for a property, but it is also possible to redirect the reading of this property to directly to the field.

```
type yourClass = class
  private
    FCount: Integer;                        { private field }
    procedure SetCount (Value: Integer);    { method for writing }
  public
    property Count: Integer read FCount write SetCount;
  end;
```

In the Delphi example we define a *private* field 'FCount' field and an also private method 'SetCount'. It adds a *public* property 'Count' to this class, which can be read/write by the outside world. When someone wants to read the value of this property, it means the private field 'FCount' must be read directly. The write a new value into this property triggers the calling of the ‘SetCount' method. After the *read* keyword a suitable type of field (as in

our case) or a function with an appropriate type of return value can be added. In the latter case, the reading of the property will trigger the call of that function.

In Delphi according to the proper working of the compiler generates the same machine code for reading the property as to reading the private field directly. So this compiled code executes with a very high (maximum) speed. In Delphi creating a read-only field is much simpler than in C#:

```
type yourClass = class

  private

    FCount: Integer;                        { private field }

  public

    property Count: Integer read FCount;

  end;
```

In the property 'Count' the 'read' part is given only, so no one can assign a new value to this property. Reading this property in the background is the same as reading the private field directly. It is almost as when we could define the field 'FCount' as its reading would be *public* and its writing would be *private* only.

7. 7. Methods

The class which contains (probably public) fields only is named as the **record**. Records are important in computer programming and they are not complicated structures. With the help of records we can group together our data elements, and can make units of data elements. An advantage of a record is when we want to give a group of data items to a function as a parameter, we can give them all in a one simple step:

```
class circle

{

  public double X_coord;

  public double Y_coord;

  public double radius;

}


static double perimeter( circle k )

{

   return 2 * k.radius * Math.PI;

}
```

A function which receives a record as a parameter works fine. Don't forget about that the keyword 'class' creates a type which is a member of the reference type family. However a reference type parameter can be a null value as well. The 1st call of this function in the example below works well, but the 2nd will generate an error:

```
circle k = new circle();

k.radius = 12.5;

double k1 = perimeter(k);      // works fine

double k2 = perimeter( null ); // causes an error
```

In fact a function with a record parameter must check every time if this parameter value is null or not.

```
static double perimeter( circle k )
```

```
{
   if (k==null) throw new ArgumentNullException(”k is null”);
   return 2 * k.radius * Math.PI;
}
```

In OOP we must insert all the functions into classes. Naturally it is possible that the function does not go into the class ‘circle’:

```
class circle
{
  public double X_coord;
  public double Y_coord;
  public double radius;
}

class calculations
{
    public static double perimeter( circle k )
    {
      if (k==null) throw new ArgumentNullException(”A k is null”);
      return 2 * k.radius * Math.PI;
    }
}
```

Because of the static modifier to call this function we must specify the class name as well:

```
circle k = new circle();
k.radius = 12.5;
double k1 = calculations.perimeter( k ); // works fine
```

It is also possible to insert this function into the class ’circle’:

```
class circle
{
  public double X_coord;
  public double Y coord;
  public double radius;
  //
  public static double perimeter( circle k )
  {
     if (k==null) throw new ArgumentNullException(”k is null”);
     return 2 * k.radius * Math.PI;
```

```
  }

}
```

In this case we can call it the following:

```
circle k = new circle();

k.radius = 12.5;

double k1 = circle.perimeter( k ); // works fine
```

When we throw the 'static' modifier away, it effects serious results.

7.1. 7.1. Instance-level methods

The methods with the modifier 'static' in fact are the old traditional functions, they are similar in both the point of view of planning and using. The functions exist in OOP world in this form. When a developer coming from the traditional world goes into the OOP world, the first advice to him "put your functions into a class { … } an that's all dude". The category name of the static methods is **class-level method**.

A method without the 'static' modifier differs in many ways. Their category name is **instance-level methods**.

An instance-level method is a function which has a non-optional parameter, which is a record made of the same class which holds this method, and this record parameter cannot be null.

This rule is very important. We discussed before that the function which receives a record as a parameter should always check if it is null or not. The usual answer to this (from a developer) is "Why? Why would give the caller me, a null value? And if it gives, and it would cause an error, then what? Is it my fault?". It is not obvious if in this case the caller or the developer of the function is guilty for the error. In the 1st look the answer seems easy: "the developer of the function is not the guilty one". However, it is possible that the function executes several steps before the null parameter value would cause the fatal error. In this case it is not so simple that "it is not his fault".

A simple rule of implementing a function is to check all the parameter value; this must be the 1st step of the function. It must be checked before the execution of any task inside the function. Of course it is not a simple thing, usually boring and easy to forget. So it is valuable that the "record parameter cannot be null", which means it is checked by the compiler itself.

Notice the changes of a static function to become an instance-level method:

· primarily, it loses the 'static' modifier,

· secondly, it loses the record parameter,

· as it has no parameter named 'k', we cannot reference to the fields with this prefix, so 'k.radius' won't work,

· an instance level method is a significant part of the instance, so the field of this instance can be accessed and reference directly (not 'k.radius' but simple 'radius').

```
class circle

{

  public double X_coord;

  public double Y_coord;

  public double radius;

  // without !!! STATIC !!! and !!! PARAMETER !!!

  public double perimeter()
```

```
  {
        return 2 * radius * Math.PI;
  }
}
```

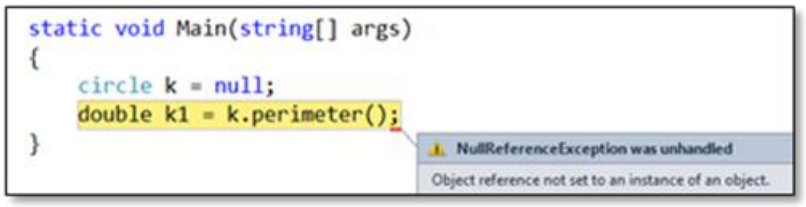
Another syntax must be used in the call of an instance level method:

```
circle k = new circle ();

double k1 = circle.kerulet(k);
```

but from now it is:

```
circle k = new circle ();

double k1 = k.perimeter();
```

Here we can see the answer. The checking of 'null' goes to the point of the method call. If 'k' would be null, then the function call of 'k.perimeter()' throws an error immediately at the call point, and probably stops the program at this line. So the check inside the function would be totally a waste of time and energy.



When we take a closer look to the function 'perimeter()' body, we might have some interesting thoughts. First: what kind of field 'radius' do we use in there? Which instance owns this field – when there is no sign of any instance? What about if there is absolutely no instance (yet) from the class 'circle'? How this function works in this case? Secondly: what about when we have several instances, what field is used in this case in this function body?

7.2. 7.2. Handling the actual instance

The answer to both questions lies at the point of the method call, and the background operation (in the generated code).

First answer (if there is no instance): this function cannot be called! Try to make some attempts, but the instance-level syntax for method call is:

instaneName.methodName(parameters)

In other words: to call an instance level method we need an instance at the point of calling. An instance is needed! It is a consequence of the modified syntax of the method calling – a function with a record parameter can be called without an instance of a record, but the OOP version of this method call cannot be fulfilled without it. Among other things this close relationship between the instance and this kind of methods is the reason why we name it "instance level method".

The 2nd question (multiple instances) can also be answered examining the method call syntax again:

```
circle k1 = new circle();

circle k2 = new circle();



double d1 = k1.perimeter();

double d2 = k2.perimeter ();
```

When we have multiple instances (k1 and k2) during the call we must specify the instance whose method must be executed. So far it is easy and simple. But how will the function know which instance was used at the call point? It is not clear the connection between the calling side and the function body.

It is important to understand that *there is no such a thing**[8]** as OOP language.* The main unit inside the computer which understands and executes the instructions is the CPU. It has no knowledge about such abstract concepts such as instances or methods. It is only a trick of the compiler. We write an instance-level method, and we do not give a record as a parameter to this. The compiler reads our high abstraction level OOP style source code, and translates it into traditional non-OOP programming concepts. An OOP developer must write this (as the compiler expects this):

```
double k1 = k1.perimeter();
```

The compiler thinks this is:

```
double k1 = circle.perimeter( k1 );
```

So we are back again at the very 1st step. Well almost. When we try to write this before, we were backbitten as we are not an OOP developer, and we should not take care of the parameter 'k1', declare this as a parameter, check if it is null or not. In the OOP style this parameter is handled automatically by the compiler, declares this, check if it is null or not. This is something!

But where is the extra parameter? We created the method 'perimeter()', and we know that there is no such an argument!

An instance level method always has an extra parameter, which holds the instance which was given in the calling point. This extra parameter is declared and handled by the compiler automatically, the name of this parameter is "**this"**.

We might see that the instance of the call is to reach our function, but we cannot see the "this" parameter in the function body. Still it cannot be understood that the field 'radius' belongs to which instance?! Well, again, this is a trick of the compiler. The compiler "allow us" to reference to an instance field without any prefix. But it knows well that a field always belongs to an instance, and this instance must be referenced. It thinks we have an instance, given by a parameter called "this" (the actual instance whose method was called). Therefore when we write the following:

```
return 2 * kerulet * Math.PI;
```

the compiler thinks it is:

```
return 2 * this.kerulet * Math.PI;
```

The whole source code written by us:

```
class circle

{

  // ...

  public double perimeter()

  {

       return 2 * radius * Math.PI;

  }

}


class Program

{
```

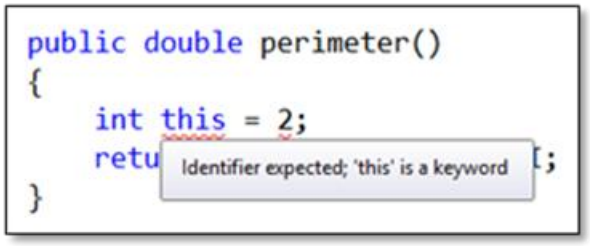
```
    public static void Main()
    {
     circle k = new circle();
     double d1 = k.perimeter();
    }
}
```

And the real source code (read by the compiler):

```
class circle
{
  // ...
  public double perimeter()
  {
       return 2 * this.radius * Math.PI;
  }
}


class Program
{
    public static void Main()
    {
     circle k = new circle();
     double d1 = circle.perimeter( k ); // checked k != null !
    }
}
```

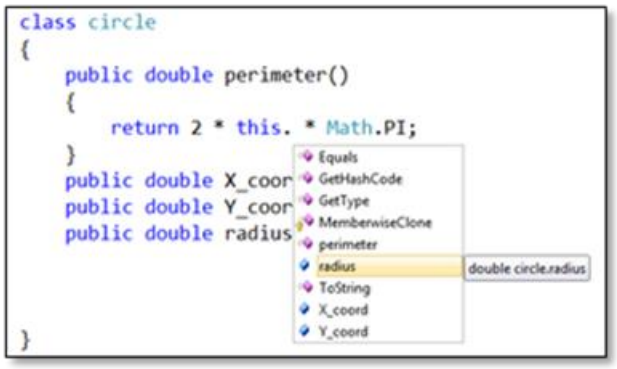
Thus, we may think: it is not a big mistake to write our code in this way. But yes, it is! In fact, the compiler has many other tricks to apply for our comfort, in in order to eliminate errors and spare us from further errors. If we write the code directly this style, the most powerful OOP techniques – eq. use of late binding - is impossible, and we do not allow the compiler to better understand our code and apply the tricks on the code generation.

The 'this' keyword is really presented in the body of an instance-level method. It is the ID of the extra parameter. Which means that the same name cannot be declared (as the ‘this’ keyword is a reserved word):



Since the parameter 'this' inside the method body means the actual instance, so you can use it in the body of the method. The 'radius' field can really be referenced as 'this.sugar'. The 'this.' prefix is useful in many cases, which

will be discussed later, but it is the best to start using it from now. The compiler won't complain against this. The 'this.sugar' form in the source code emphasizes that: the field radius of this instance. An additional benefit if we writes 'this.', after we write the point letter the VS immediately shows a hint list containing the list of the available fields and methods.



The 'this' keyword in the instance-level methods refers to the current instance. It can be used in the source code to refer to the fields. The 'this' is the name of the extra parameter of the method handled automatically by the compiler. The 'this' is therefore a reserved word (a keyword). The type of 'this' matches the type of the class in which the method is included.

8. 8. The function Main

The program consists of instructions. The instructions are executed in a specific order. According to the sequence execution rule the order must be the same as the instructions appear in the source code. We need a starting point, which will define the very first instruction. From then on, it is clear which will be the next … and so on.

The starting point for a C language program usually a special function called the 'main()' function. The language C# is designed on the foundation of C, so the choice is very similar, the name of the start function is 'Main()' (with a capital M in writing).

In an OOP language each function must be placed into a class (encapsulation). Thus, the function 'Main ()' cannot be outside of a class, so it should be placed into a class as well. Into which one? It does not matter! The rule is that the program must have exactly one function 'Main()' somewhere! Somewhere in the precise means that no matter what class contains this function.

The 'Main()' function is not allowed to be instance-level! Why? To execute an instance-level method we need an instance before. To have an instance the instantiation instruction is needed. Where to place this instantiation when the very first instruction must be placed into the function 'Main()' (chicken or the egg problem)? If the 'main ()' function is not accidentally marked the static modifier (class level), then the compiler will not recognize it as the starter function.

What about the protection level of the function 'Main'? It does not matter! Exceptionally it could be private as well, it won't change the fact. According this the simplest Main looks like this:

```
static void Main()

{



}
```

It is interesting to know that the function Main can have 'void' or 'int' return type. The latter case is used when the program is started from a batch program (or a script). This kind of start the program must indicate to the caller script if its run was a successful one or not. As usual the return value of 0 indicates that there was no error, any other value indicates some sort of error happened. When 'void' is the return type is the same case as if 'int' would be, and 0 was returned (this is done by the compiler):

```
static int Main()
{
  // ..
  return 0;
}
```

When we start our program from a command prompt, we can give it command-line parameters.

The parameters written in this command line which are after the name of the program (fileEnrcypt) are actually handled as strings. Their number depends on how many of these are included into this command. To handle these strings parameters theymust be defined for the Main:

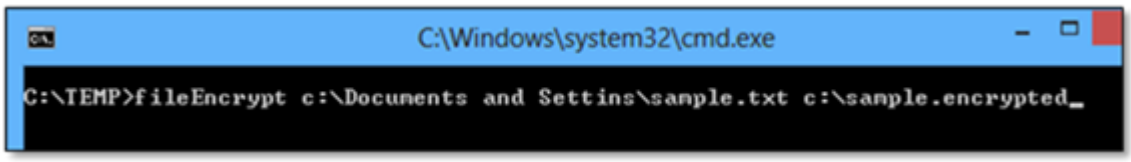
```
static void Main(string[] args)
{


}
```

There will be two elements in the ’args’ vector:

args[0] => ”c:\sample.txt”

args[1] => ”c:\sample.txt.encrypted”

Take care of the spaces in the command line as for the operating system they are word-separators. Take a look into the following example:



In this case the ‘args’ vector will hold 4 items:

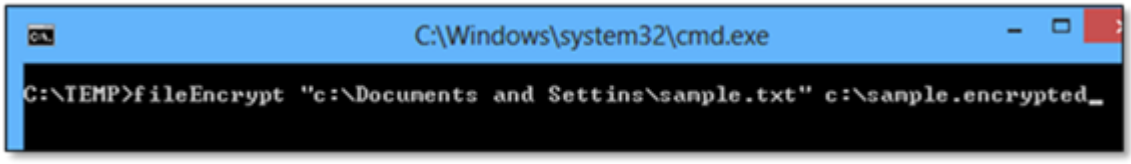
args[0] ð ”c:\Documents”

args[1] ð ”and”

args[2] ð ”Settings\sample.txt”

args[3] ð ”c:\sample.encrypted”

Fortunately, recent versions of the Windows operating system understand the quotes in the command line:

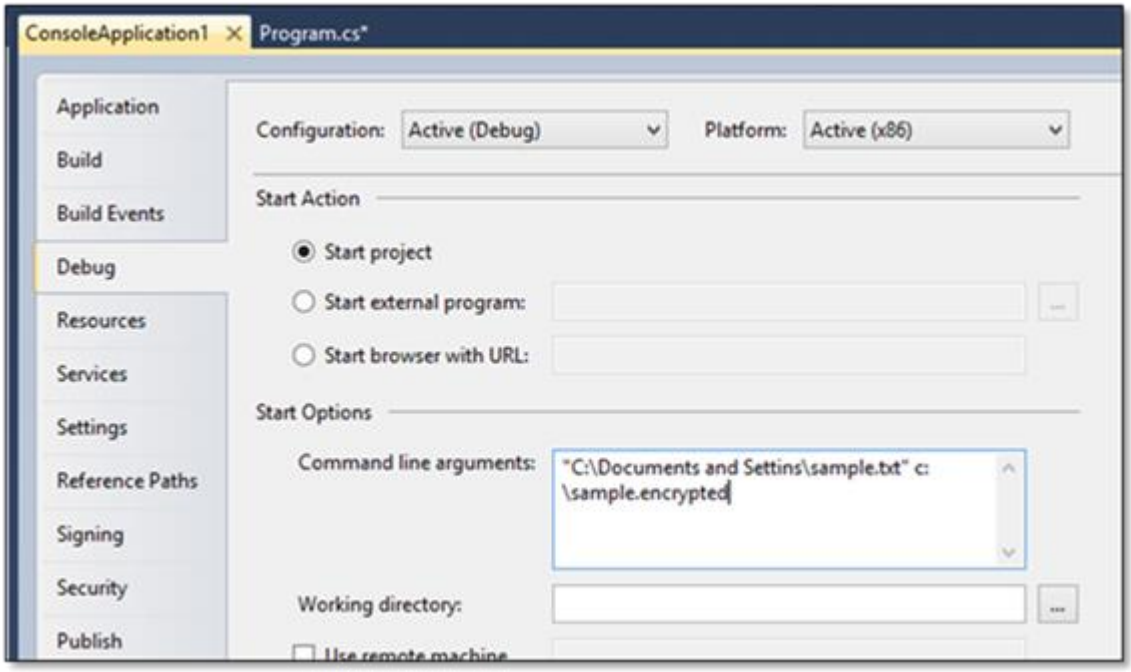


Then the ‘args’ vector will be:

args[0] ð ”c:\Documents and Settings\sample.txt”

args[1] ð ”c:\sample.encrypted”

If the program is not started from the command line, but from the VS, then we still can give command line parameters (for testing purposes). Right click in the Solution Explorer window at the project item, select Properties (the lowest menu item), and then find the Debug configuration tab, and Command Line Arguments section. Here enter the desired parameters for Main:



9. 9. The constructors

Constructors are key elements in the OOP world. Many problems can be easily solved by the constructors. But the most important task of the constructor is about to set the initial value of the instances.

Take the previous example 'student’ class and the field 'age'. The field age rule allows only values between 18..60, and this must be guaranteed. Using all our knowledge we protect the real field and create a public property:

```
class student
{
  protected int _age;
  public int age
  {
    get
    {
      return _age;
    }
    set
```

```
    {
       if (value < 18 || value > 60)
          throw new ArgumentException("only 18..60 accepted");
       else _age = value;
    }
  }
  // ... folytatás ...
```

We might feel a great work has been done, we can relax. If we use the object, it works well:

```
student d = new student();
d.age = 22;
Console.WriteLine("age = {0}", d.age);
```

However, it is easy to generate errors if we forget about a couple of things. According to the rule, the value of the age field must be between 18 .. 60, in all cases! This must be guaranteed by the object, and since the field is accessible for the outside world, they assume the value of the age the same:

```
student d = new student();
// skip: d.age = 22;
Console.WriteLine("age is = {0}", d.age);
```

If you create an instance a *student*, the age field is initially set to zero[9], which does not comply with the rule. At *WriteLine* we will see that the *age* is zero. As it surprises the outside world, it can generate an error, which is not admissible.

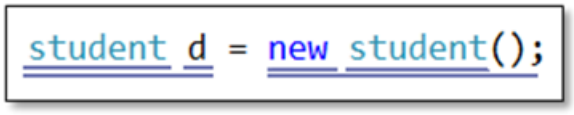
One solution is to define the initial value of the field, such an assignment at the declaration of the fields:

```
class student
{
  protected int _age = 18; // initial value
  public int age
  {
```

One cannot use this method in all cases. What initial value should be given to the ‘name’ field, or the field neptun id[10]? In addition, the outside world can be disturbed by this method, as it does not require setting the values of these fields, it isso much easier to forget to do that. If they want to set them, it is still not an elegant instantiation, we need to use more than one statement. The first statement is the instantiation, followed by the settings of the fields. Not elegant.

9.1. 9.1. Constructors during the instantiation

Approaching the other side: so far, it was not observed deeply why the instantiation looks like as the following:



On the left side of the assignment operator is a declaring of a student type 'd' variable. Since this is a reference type, it needs 4 bytes of memory. The actual data (the fields) are stored in secondary memory area. This second area is reserved by the 'new' keyword. The 'new' is clever enough to reserve the appropriate amount of memory

for a student instance[11]. A more interesting question is what is the 'student()' part after the ‘new’ keyword exactly?

Note that this 'student()' part looks like as a function call. This is confusing that there is a ‘new’ in front of it. In the absence of it would be a function call:

```
string s = inputSg();   // ez valóban függvényhívás

diak d = diak();      // ez is annak néz ki
```

Well, the reason for disturbing similarity: this is really a function call. A call of a special function. Special in many ways. First, the name. The function name is 'student’, which matches the name of the class (class student).

Secondly, the syntax of the call: this function can only be called in this way, after a 'new' keyword. These two steps create a group complementing each other.

Such a function with this characteristics are named *constructor* in the OOP world. A constructor is a method, a function. Its task: the newly created instance set to its initial state. This initial state is very important. An object instance is required at the time of the creation to guarantee right state: according to the rules each field must have a proper value. The attitude is not correct to create an instance of a student with a wrong state at the beginning, then needs a series of assignments to reach the good state:

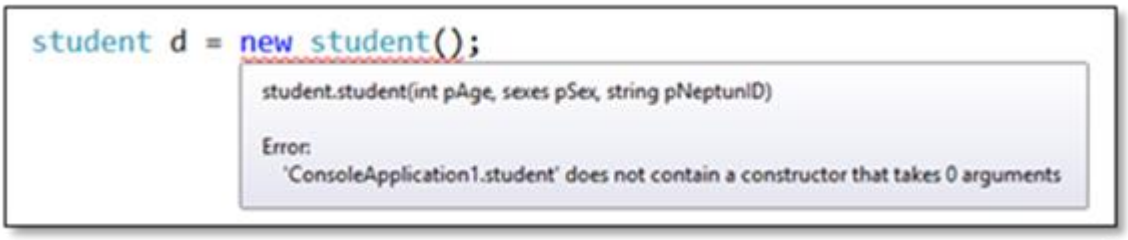
```
student d = new student();

// -- exists from now, but it has a wrong state

d.sex = sex.male;

d.neptun_id = "QQDK23";

d.age = 22;

// --- it has a good state here
```

9.2. 9.2. Creating a constructor

We can create a constructor function, which eases the instantiation process of the outside world, and also could specify the required data as well. When writing the constructor a method with the same name as the class owns has to be prepared. The constructor is not mandatory, usually typically *public*, it has no return type, even void, nothing is available to specify:

```
enum sexex { male, female }

class student

{

  public student(int pAge, sexex pSex, string pNeptunID)

  {

    _age = pAge;

    sex = pSex;

    neptunID = pNeptunID;

  }
```

We created a constructor of the student class. It has three parameters, which are: age, sex, and the neptun id. From this moment to use it is required for the outside world. In other words, the old and simple method for creating a new student instance no longer operates:



The outside world must call a constructor of the class after the 'new' keyword. Because the constructor's name matches the name of the class[12], so it is easy to find out this name. When the 'new' allocates the memory, all the fields gets their initial values, depending on their types (the bool fields are false, numeric fields are zero, all the reference type fields get null, etc). Following all the assignments statements attached to the field declaration are executed. Finally, the call of a constructor execute, which is mandatory to be called after the 'new' keyword. In this example, the constructor has three parameters, so at the call three parameters should be given:

```
student d = new student(22,sexes.male, "QQDK23”);
```

9.3. 9.3. Creating several constructors

We can create not only one but several constructors. Usually we have a lot of constructors for the same class. The same rules apply to each of the constructor: their name is the same as the class name. The rule ‘overloading’ allows us to create functions with the same name as long as they are different in parameters. In this case it will help. The constructors of the same class differ only in this thing, so we must take care about it:

```
enum nutrition { meatEater, plantEater, omnivorous }

enum sexes { male, female }

class animal

{

  public animal(nutrition t, sexes n)

  {

     // ...

  }

  public animal(animal mother, sexes n)

  {

     // ...

  }
```

In the above example we can create animals by defining what it eats and what its sex, or give the parent animals (small animals would eat the same the mother) and the sex.

```
animal m = new animal( nutrition.plantEater, sexes.female );
```

```
animal p = new animal( m, sexes.male );
```

9.4. 9.4. Lack of constructor

In the previous chapters we created several classes, never made constructors, but we still were able to create instances:

```
class circle
```

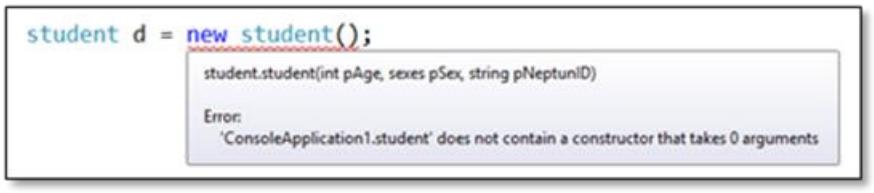
```
{
  public double x;
  public double y;
  public double radius;
}

class Program
{
  public static void Main()
  {
    circle k = new circle();
    k.x = 12.4;
```

The code is equivalent to the case when the circle would have a parameter-less constructor. Since this constructor has no parameters, the fields get their initial values (according to their types) - the constructor body is empty:

```
class circle
{
  public double x;
  public double y;
  public double radius;
  // ...
  public circle()
  {
  }
}
```

It is not allowed for a class to not have any constructor. If the programmer does not create a constructor, the compiler does (like the one above). This constructor is *public*, has no parameter, and does not include any instruction in the body. It has only one important meaning: allow us to create instances of that class. Since syntax rules of the C # language require that during the instantiation a constructor must be called after the 'new' keyword, so constructor must exist. This provides the mechanism described.

Note, however, that the compiler does this favour for us only when we do not creates any constructor. If we create at least one, then the compiler won't create this empty constructor. This is the reason why the error shown below will pop up. If we create a parameterized constructor for the 'student' class, the instantiate with the parameter less one cannot be fulfilled:



9.5. 9.5. Checking the parameters

A field often has a rule. Therefore, a property is created with get and set parts. Inside the set we usually have to check the new value, but the field might get its value through a constructor. Let our example be the student age again, the rule is still 18 .. 60:

```
class student

{

  // hidden protected field

  protected int _age;

  // property

  public int age

  {

    get

    {

       return _age;

    }

    set

    {

       if (value < 18 || value > 60)

             throw new ArgumentException("only 18..60 accepted");

       else _age = value;

     }

  }

  // constructor

  public student(int pAge)

  {

     _age = pAge;

  }
```

This code leave a gap in the protection system of the object. The age field cannot be set to an invalid value by an assignment statement, the *set* will check the rule every time. But through the constructor a bad value can be stored to the field as it copies the parameter value unchecked to the field. In this way a student instance might exists in a bad state, whose age is invalid:

```
public static void Main()

{

  // 12 yrs old student

  student d = new student(12);
```

To avoid this, the constructor should check and validate the value of the parameter:

```
// constuctor
```

```
public student(int pAge)
{
   if (pAge < 18 || pAge > 60)
       throw new ArgumentException("only 18..60 accepted");
   _age = pAge;
}
```

The solution works, but a lot of problems appears. The complete control algorithm (which is this simple case only one 'if', but it could be much more complex) duplicated in the code: inside the *set* and inside the constructor. Not only copying the code is the problem, but also the redundancy. If the rule is modified for some reason, now it must be applied in two places.

Instead, we should use the already created tested *set* part:

```
// constructor
public student(int pAge)
{
  // not into the field, but the property !
  age = pAge;
}
```

In the first (wrong) example the constructor stored the parameter value directly into the field. In the latter case, the *set* part of the property is used, so the parameter values must go through the validation process.

9.6. 9.6. Write-once fields

We discussed previously the problems about the implementation of a write-once fields. With the help of the constructors the problems of the write-once fields can be handled easily. The only one write access, the initial value of the field can be specified via the parameter of the constructor, so the implementation becomes much simpler:

```
// hidden and protected field
protected int _age;
// property
public int age
{
  get
  {
     return _age;
  }
}
// constructor
public student(int pAge)
{
   if (pAge < 18 || pAge > 60)
```

```
        throw new ArgumentException("only 18..60 accepted");

    // store directly into the field, as there is not set part

    _age = pAge;

}
```

In the code above the property ‘age’ is a read-only one. In the *get* part it is not necessary to verify that the physical field, '_age’ has been set previously, since the constructor will set it. The constructor accepts the initial value of the field, check its value according to the rule.

9.7. 9.7. A property with two different protection level

The problems of the write-once fields can be solved by writing a property, a *get* part only, and using a constructor. The latter can be executed only once, so the value of the field can only be set once. This solution is not very elegant. The rule of the field is verified and handled in the constructor, too far away from the property and the field. It would be more logical to handle that in the *set* part. In addition, if there is more than one constructor, which have parameter for this field, the checking algorithm must be copied to each one. We have several reasons to refactor this checking out of the constructors.

However, we cannot create a public *set* as the outside world would be able to use it. If we would create a *set* the best would be to set it protected level. But then both *set* and *get* become *protected*, and the outside world would lose the possibility of reading the value. The true solution would be to set the *get* part public, and the *set* would remain *protected*. This is not a problem. If the both parts of a property have the same protection level, this level can be set outside of the property along the declaration. If we want to use different levels of protection for the two parts, we must set one level outside, and the other level inside of the property:

```
public int age

{

  get

  {

    return _age;

  }

  protected set

  {

     if (value < 18 || value > 60)

          throw new ArgumentException("only 18..60 accepted");

     else _age = value;

   }

}
```

When we have different protection level, we must set the weaker (more public) one outside, and can be further strengthened inside. It will not work in the following way:

```
protected int age

{

  public get

  {

     return _age;
```

```
  }
  set
  {
     if (value < 18 || value > 60)
         throw new ArgumentException("only 18..60 accepted");
     else  age = value;
  }
}
```

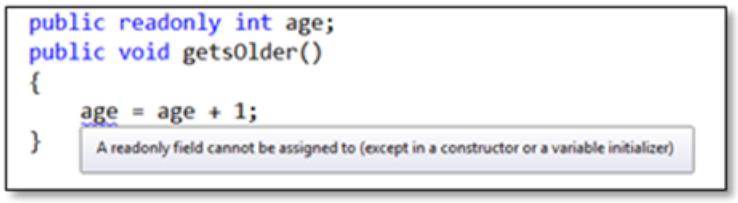
9.8. 9.8. The real write-once fields

Note that a “write-once field” can actually be written more than once. The instance methods can modify its value any time, only the outside world cannot, as it cannot write to the field nor directly nor using a property’s *set* part.

There exists a real write-once field in C#. It must be marked with the 'readonly' keyword, and even has *public* protection level:

```
// public but readonly field
public readonly int age;
// constructor
public student(int pAge)
{
   if (pAge < 18 || pAge > 60)
        throw new ArgumentException("only 18..60 accepted");
   else age = pAge;
}
```

The value of a ‘readonly’ field can only be set in the constructor. Later it may not participate in an assignment operator on the left side, including inside additional class methods:



9.9. 9.9. Constants

The constants can be read and the real write-once fields are very similar to each other. However, there are differences:

```
// constant
public const int maxAge = 80;
// public, but readonly field
public readonly int age;
```

```
// constructor

public student(int pAge)

{

   if (pAge < 18 || pAge > 60)

        throw new ArgumentException("only 18..60 accepted");

   else age = pAge;

}
```

First difference is the declaration. To declare a constant the ‘const’, for the write-once fields the 'readonly' keyword must be used. An initial value for a constant must be insert to the declaration as the initial assignment, since the constant cannot appear in an assignment statement’s left side anymore. The value of a write-once field should be given in the constructor.

The second difference is in the point of view of the outside world. The constant is the part of the class, there is no need for an instance to read the actual value of a constant. The write-once field is an instance level field, however, to read its value instance is needed (because the field’s value can be different for the different instances):

```
student d = new student(22);

int a = d.age; // reading the readonly field => 22

int ma = diak.maxAge; // reading the constant => 80
```

Note: as any field, the initial value of a ‘readonly’ field can be given by an initial assignment operator, as the case of a constant. However, this is actually an error for two reasons. The 1st is that this read-only field become essentially a constant this time (this value could be changed in a constructor but no other way exists), however this ‘constant’ becomes an instance level field therefore to read its value an instance is needed - which is meaningless, because the value is not instance dependent. The 2nd reason, as for every instance level fields, when an instance is created, another copy of this field goes into the memory, occupying another area in the memory with the 'new' operator. This is a waste of memory, since no matters which instance is used to query the value of this field – in every case we would get the same value.

```
class diak

{

   // constant

   public const int maxAge = 80;

   // read only field

   public readonly int maxScholarship = 90000; // HUF
```

If you need a constant, which we want to be read only at instance level, consider the following method:

```
class student

{

  protected const int _maxScholarship = 90000; // HUF

  public int maxScholarship

  {

    get

    {

       return _maxScholarship;
```

```
    }

  }
```

The constant remains constant, but the level of protection is *protected*. For this reason, the outside world does not know anything about 'student._maxScholarship'. The property is at instance level, however, so to execute the *get* part an instance is needed. The get part returns with the value of the constant.

10. 10. The data members

In the OOP world the items which hold concrete values are called data members. There are three types of data members:

instance-level fields,

class-level fields,

constants.

10.1. 10.1. Instance level fields

The instance-level fields store such data values which can be different for the different instances of the class. Instance of a 'circle' class might hold different radius values, andthe x and y coordinates as well. According to the instances of the 'student' class, the different students might have different names, neptunIDs, year of born values, and so on. The instance-level fields are essentially the same as the field concept of the "record" structure, so the "instance level" prefix is often omitted.

The syntax of declaring an instance level field is:

[protection level] <type> <identifier> [= initial value];

The protection level can be: public, protected, private. If you do not specify neither, the default protection level is private. The initial value must not be set, can be omitted, in this case a default initial value is used which bases on the actual type of the field. Numbers (int, double, etc.) becomes *zero*, bool fields become *false*, char typed fields start with the character whose code is *zero* (UNICODE), and all reference typed fields starts with the *null* value.

In the memory there is no instance level fields at the beginning of the program lifetime. When we instantiate, the 'new' operator is used, it books a proper sized area in the memory for the new instance. The amount of bytes is primarily determined by the number of the instance-level fields and their types. All the private field counts, they need memory the same way as the protected or public fields.

In OOP approach, the level of protection is typically protected for fields, rarely private. The public fields are unprotected, the value can be changed at any time in the outside world, but the rules of the field type never been violated. For this reason, the methods which work with a public field should always validate their values according to the stronger constraint (if there any) before using them.

The **scope** of the instance-level fields depends on their protection level. The private fields can be referenced only in the methods, constructors, properties of the same class. In addition the protected fields are accessible in the child class methods, constructors and properties as well. The public fields can be referenced anywhere in the program text.

The **lifetime** of an instance-level field depends on the lifetime of the instance itself. At the instantiation each fields gets into the memory (new operator + constructor), typically after starting the program, at a later time. The field exists as long as the instance exists. In C# removing and deleting an instance from the memory is an automatic process, the Garbage Collector (GC) detects that an instance is no longer used by the program, has lost its memory address (there is no variable or a field stores the address of the instance nor directly nor indirectly). The instance no longer exists for the program, but it keeps reserving its place in the memory physically, until the GC detects this situation, and frees the reserved memory area. The latest time it will occur when the program finishes.

```
class student
```

```
{
   protected int age;
   protected string name = "-- unknown --";
   protected string nationality = "HUN";
   //
}
class circle
{
  public int X_coord;
  public int Y_coord;
  public double radius;
}
```

To instance-level fields in methods, constructors, properties in the same class can be referenced directly (with the name of the field), or using the keyword 'this' like 'this.fieldName'. If the scope is protected, then the field is accessible in the child class methods as well, and can be referenced the same way. The public fields can reference in anywhere of the program text, but the syntax is different - they need to identify the instance as well, so the syntax is: 'instanceName.fieldName'. In other words, there are three possible types of syntax:

fieldName

this.fieldName

instanceName.fieldName

The value of an instance-level field can be defined as an initial assignment, but it is not really typical. Most often it is given using a constructor (the constructor takes the parameters of the initial value). The value of a field can usually be changed using a property, but any method of the class can also modify it freely.

At the instance level field the 'readonly' modifier can be set. In this case, the value can be set only as an initial assignment or inside a constructor. Furthermore, a property or a method cannot modify it.

10.2. 10.2. Class level fields

The class-level fields stores values which are enough to store only once in a program. For example speed limit is 50 km/h according to the traffic rules, an adult is at least 18 years old, the minimum wage is 78,000 HUF, the VAT rate of books is 5%, the price of petrol is 420 HUF/litre. These values often defined as constants, but we do not want to. These values can be changed at any time. It is better therefore to read the actual values from a configuration setting (.ini file,. xml file, database) at the start of the program, or read from the console, or from any other data source.

The class-level fields are often explained as their values do not depend on any instance, but characterise the class itself, or all the instances of that class. This might help to understand the role of the class-level fields.

The non-OOP environment, in a third-generation language the programmers interpret the class-level fields as global variables. Variables where they can store data items, and where their functions might access these values, might read or modify them. This is true, but in a lot of way it is inaccurate. The word 'global' usually indicates a *scope*, however the scope in the OOP depends on the protection level. The class-level fields much more different from the instance-level fields in their lifetime.

The syntax of declaring a class-level field is:

static [protection level] <type> <fieldName> [= initial value];

This declaration syntax differs from the instance level syntax at only one point: the 'static' modifier. The order of the protection level and the 'static' keyword can be reversed, so the following syntax is also correct:

[protection level] static <type> <fieldName> [= initial value];

The rules of the **scope** of a class-level (static) fields are the same as described for instance-level fields. The private fields can only be accessed inside of the class, the protected fields can be referred in child classes also, and the public fields can be accessed in the entire program text.

The lifetime of a class-level field is **static**, so that the fields are loaded into the memory at the very start of the program, and they remains there until the program terminates. Their existence is not bounded to any other construction or principle. These fields are presented in the memory only one times, even if not a single instance of the class is made, even if there are many instances made of it. The creation or the destruction of a class level field is not influenced by the GC.

```
class student

{

   protected static int minAge = 17;

   public static string university_prefix = "EKF";

   static public int length_of_neptun_id = 6;

}

enum roadRule { leftHanded, rightHanded }

class traffic

{

   static public int maxSpeedLimitDaytime = 60;

   public static double maxAlchoholLevel = 0;

   static public roadRule actualRule = roadRule.rightHanded;

}
```

The class-level fields already exist and are available when there are no instances ever made of the class. Therefore, the class-level fields can be referred from outside world as className.fieldName'. The codes inside the class (constructor, method, property) may also refer as 'className.fieldName', but there we can refer to the field directly (without identifying the name of the class). If the protection level is at least protected, the child classes can also access a field, and might refer to it simply 'fieldName', or the longer form 'className.fieldName'.

Since the 'this' keyword identifies the current instance, but a class-level field does not bind to any instance, the 'this.fieldName' form does not exist, using it causes a syntax error. In other words, there are two types of syntax:

fieldName

className.fieldName

The class-level fields are set by an initial assignment (usually), or very rarely, class-level constructor. This kind of constructor (will be discussed later) cannot receive any parameter, more often it evaluates the expression, or uses fast algorithm to calculate the value. A value of a class level field can also be modified by properties (not common), but more often is modified by the class-level methods.

We can use the 'readonly' modifier to a class-level field. The value of this kind of field can be set by an initial assignment or inside a class-level constructor. Furthermore, the no one (no property, no method) can change it.

10.3. 10.3. Constants

The constants contain data whose value is known at the time of writing the program, and is not expected to change. These kind of data can be inserted into the text of the source code every time it is needed, but more it is interesting to give it a name (an identifier) and use this name in the text, increasing the readability of the source code. When we know that the value is fixed, but we are not sure of its correct value, then be sure to use a constant. We can use the name of the constant referring its value. When we know the proper value, we can insert this into the initial assignment statement of the constant once, then the program will use it at all the places where the constant name is present.

In the OOP world a constant must be placed inside a class as well. The class name is added to the identifier of the constant, so you might want to choose a class with a suitable name. The constant such as PI is placed in the Math class, although it could be placed inside the String class as well, but no one would think to look there. Its full name is Math.PI, which makes everyone understand simply what kind of value it holds. The name String.PI would confuse many people.

The syntax for declaring constants:

const [protection level] <type> <fieldName> = initial value;

Similarly, to the syntax of the class-level field declaration, the order of the protection level and the const keyword can be reversed. The meanings of the protection levels are the same. A private constant can be referred only inside the class.

The **scope** of a constant depends on its protection level. The value of a constant cannot be changed by the outside world nor inside the class. So the constants are typically public, and their values are available anywhere in the source code. There are private and protected constants, when their values are irrelevant or secret to the outside world.

We usually do not talk about the **lifetime** of a constant, as they have no lifetime in the classical meaning. When it comes up, everyone naturally takes it, that at the very first statement the constant can be used, it owns it value, and at the very last statement it still exists and still represents the same value. So its lifetime is considered static. We do not usually say this, because the constants are not stored together with the variables in the memory (are not stored in the data segment, the area allocated for this purpose), but in the code segment instead. It is not true according to some (optimized) translation process, where the value of a constant value might not store anyhow, but is inserted into the code as a literal value would, every time where the constant is referred. (Note that the compiler may even decide that the constant is stored with the variables in the data segment, but this is not usual). Programmers usually think of the constants that its storage and handling is the compiler's job; it is unnecessary to know the details about the storage and handling mechanism.

The constants within the class code can be referred directly by its name. Within the children classes (if accessible) also the name can be used. The outside world can use the 'className.constantName' form. In this way, the constants and the class-level fields have the same syntax. The difference between them is that the constant value cannot be changed, but the class-level field can be any time. A class-level 'readonly' field is almost the same as a constant, but its value can be set by the class-level constructor, and besides this there is no way to change it. However, the class-level field (even 'readonly' or not) is a variable and is stored in the data segment area. So theoretically the value of a read-only field - bypassing the compiler with some tricks - can be changed later, while the constants are stored in a completely different way, so the background handling and storing a constant bases on very different sets of rules.

11. 11. The inheritance

11.1. 11.1. The inheritance of the fields

In OOP the second principle is the *inheritance*. Inheritance provides a new object class development is to be able to use an existing object class. When we develop a new class using the inheritance, the new object receives all the fields and methods of the ancestor object class fields. Further we can add new fields, methods, expanding the functionality of the ancestor class, or in the child class we can modify the inherited functionality.

In other words, the child class can add or change but cannot throw away fields or methods compared to the ancestor class. So the child class knows everything the base class knows, only a few things operate differently in the child class, and the child class might own more things than the base class. (This will be important later!)

The declaration of inheritance is simple: when a child class is declared, the ancestor class must be given:

```
// there is no ancestor class declared

class square

{

  public double a_side;

}


// the ”square” is the ancestor

class rectangle : square

{

  public double b_side;

}
```

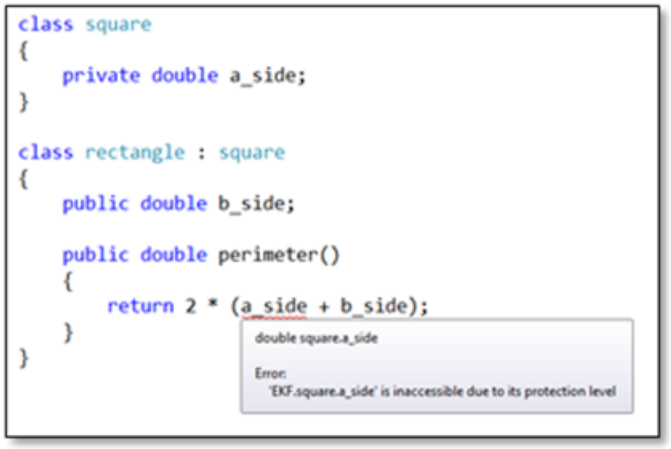
The 'square' class has only one field. When we instantiate (with the ‘new’ keyword), we can calculate the memory needed as it has only one double typed field which needs 8 bytes. The ‘rectangle’ class has two fields; the inherited 'a_side' and he newly added 'b_side', so a rectangle instance needs 2 × 8 bytes of memory[13].

The rectangle class methods reference to the field 'a_side' as if this class was own it, as it does with the field 'b_side':

```
class rectangle : square

{

  public double b_side;

  public double perimeter()

  {

     return 2*(a side+b side);

  }

}
```

We can think of inheritance as in the first step the compiler copies (copy-paste step) of the complete source code of the square class to the rectangle class, and then begin to read the remaining part of the rectangle class. For the first glance it shows quite well how the inheritance works.

Of course this is not so simple, it is not entirely true because in this case (copy-paste) the private fields of the square-class would be accessible inside the rectangle class, if the field declaration would go into the rectangle class, it would mean the following:



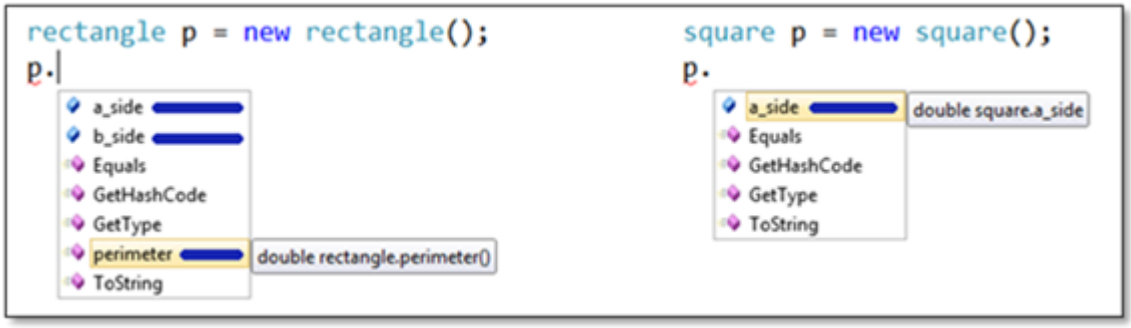
The error message indicates that we are not able to access the private field ‘a_side' due to its protection level. But it does not mean that there is no field 'a_side' of the rectangle class, just means that we are not able to access it directly!

In addition to direct access we have indirect opportunities as well. For example the square class might contain a function or property, which can return the actual value of the field 'a_side':

```
class square

{

  private double _a_side;

  public double a_side

  {

    get { return _a_side; }

  }

}


class teglalap : negyzet

{

  public double b_side;


  public double perimeter()

  {

    return 2*(a_side+b_side);

  }

}
```

In this case, we inherit the private field '_a_side', and a public 'a_side’ property as well. The latter is accessible in the child class due to its protection level. If we create a rectangle instance, it is natural that this instance has an 'a_side' read only property, a 'perimeter()' method, and a field 'b_side'.

When we creates an instance from the square class, there is only the property accessible (in the list we can see other methods Equals(), GetHashCode(), getType() and toString(), they will be discussed later):



11.2. 11.2. Problems around fields inheritance

After becoming familiar with the inheritance rules we must discuss some extreme cases! These cases must be examined as they very rare in real life but with the help of understanding and solving them we can deeply understand the inheritance rules.

Let us suppose that there is a 'first' class with three fields:

```
class first

{

  private int a = 1;

  protected int b = 1;

  public int c = 1;

}
```

An instance from this 'first' requires at least 3 × 4 bytes of memory because of the 3 fields. Let's create a 'second' class, which is the child of 'first':

```
class second : first

{

  public double a = 2.0;

  public double b = 2.0;

  public double c = 2.0;

  public double d = 2.0;

}
```

We can see the following messages in VS:

The 'second' class already has three fields at the beginning inherited from its ancestor, and adds four new fields. Altogether there are seven fields in the instance, so the memory requirement is 3 × 4 + 4 × 8 bytes (total 3 int and 4 double fields).

The problem is, that the 7 fields share only 4 different names. The 'a', 'b' and 'c' field names appear double. The 'd' field name is unique, a name belongs to only one field.

First, we note that this situation must be avoided. We do not use the same name to several fields. The VS warns us also ("second.c hides inherited member 'first.c'). This is usually a design problem. (Why do we name our

fields when so many names exist to choose from). A simple modification on the design: choose different names in the second class, and the conflict is resolved.

Let's suppose we do not want to do that. The situation then becomes really problematic!

First, notice that there is nothing wrong with the field 'a', VS shows no error about it. Why? Because 'a' is a private field, so its scope does not extend to the class 'second', so another 'a' field can be declared there without overlapping each other's scope. The field 'd' is not present in the ‘first’ class, so there is no problem with the scopes.

The scope of the ‘first.a’ and ‘firs.b’ field however extends to the 'second' class, so these identifiers could be used directly in the second class according to the inheritance. As fields such a name are declared, the overlapped scopes causes a "warning" in VS. The warning is a kind of error, we may want to deal with it.

The warning indicates the disturbing of VS. While this is not a strict syntax error, it indicates a problem with the design. The VS therefore asks us to think about the code. If you have accidentally caused the problem, the warning signal is very useful. The "disturbing" does not go away until we indicates to VS that our decision is final and we acknowledge the consequences. Our decision is indicates by the 'new' keyword inserted into the field declarations.

```
class second : first

{

  public double a = 2.0;

  new public double b = 2.0;

  new public double c = 2.0;

  public double d = 2.0;

}
```

First, note that this 'new' keyword is not the same as used in the instantiation, at least it does not do the same semantically. The 'new' used in the instantiation process calculates the size of the memory area needed for the new object instance based on the fields, then calls the constructor. This 'new' here simply means "new", its role is to display our purposes only. The compiler will understand our decision and will no longer bother us with the warning messages.

Now, examine some parts of the code, how to work together with overlapping scope of fields!

```
class first

{

   private int a=1;

   protected int b=1;

   public int c=1;


   public void first try()

   {

      this.a = 11;

      this.b = 11;

      this.c = 11;

   }

}
```

The fields declared in the 'first' class are available as appropriate. The code here suffers from no collision problems as the fields' scope declared in the child class never extends backward to the ancestor class.

The code inside the 'second' class can reflect to six of the seven fields (excluded the private one). However, the inherited 'b' and 'c' are covered by the new fields. Accordingly, in code inside the 'second' class (in the body of methods, properties or constructor) can use primarily the newer (double) fields only:

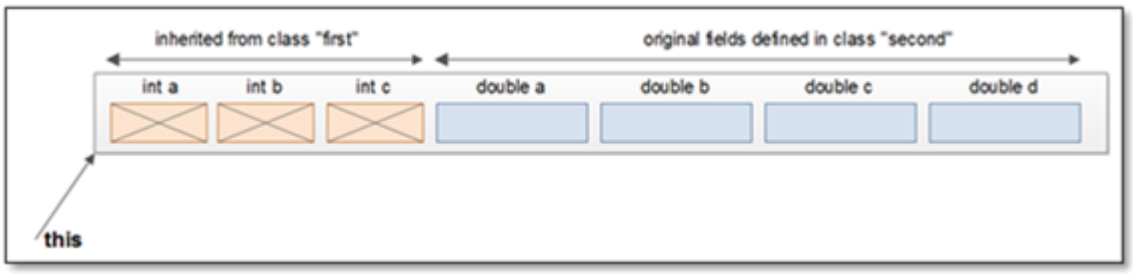
```
class second : first

{

  public double a = 2.0;

  new public double b = 2.0;

  new public double c = 2.0;

  public double d = 2.0;


  public void second_try()

  {

     this.a = 22.2;

     this.b = 22.2;

     this.c = 22.2;

     this.d = 22.2;

  }

}
```

The question is this: if there are existing (inherited) fields which are overlapped by, the int-typed 'b' and 'c' fields, how to reach them?

It does not depend on if we write the keyword 'this' or not. It was mentioned earlier, the 'this.' is added automatically to the field names:

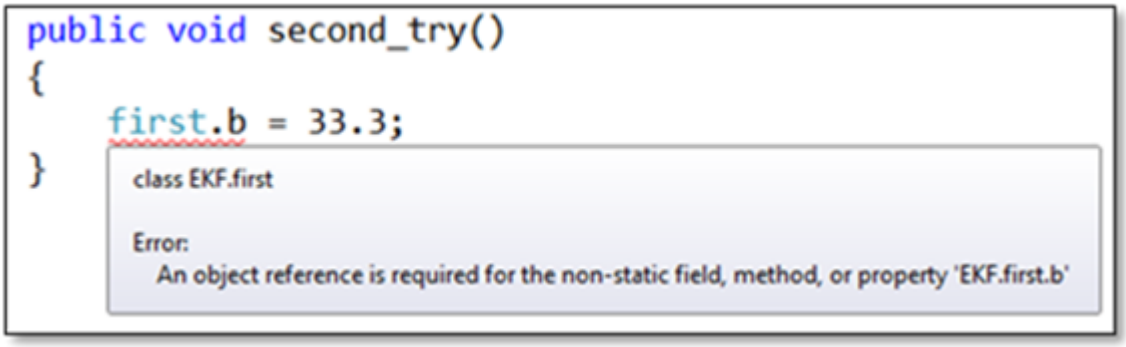
```
public void second_try()

{

   b = 33.3;      // both are the same

   this.b = 22.2; // field reference
```

Now we will examine how the 'this' sees a 'second' class instance:



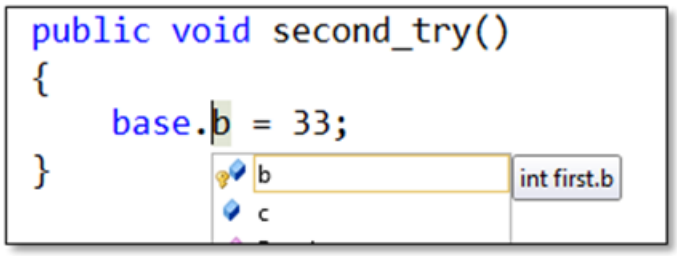
The 'this' cannot reach the field 'int a', because its protection level. Cannot reach nor 'int b' nor 'int c', because they are covered by the new versions of these fields.

Won’t work if we try to inform the compiler that we want to reach the field b declared in the 'first' class with at least with the syntax 'first.b'. This syntax is the same as we used to reach static class-level fields:



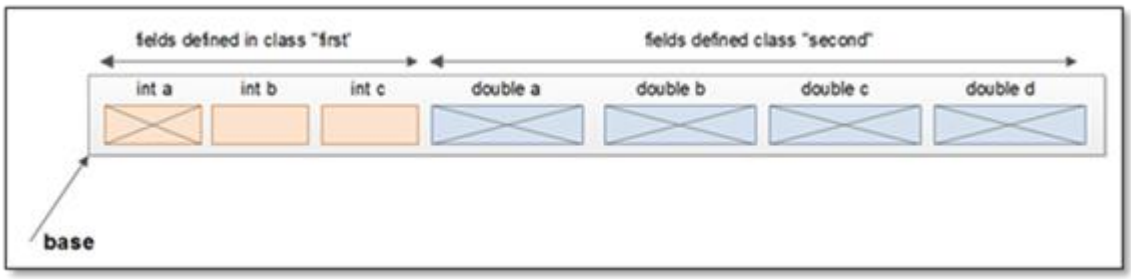
11.3. 11.3. The keyword ‘base’

Basically, there are two methods to reach the covered fields in the ancestor class. The first needs the 'as' operator type modifier, of which we will discuss later. The other is to use the keyword 'base'. The 'this' keyword references to the current instance, the 'base' does as well. Using the ‘this’ inside a method in the 'second' class, the type of ‘this’ is ‘second’. Which means writing the 'this.' (point at the end) the VS will show us all the fields and methods available for a ‘second’ typed instance not listing the covered fields. The 'base' refers to the same actual instance, but the type of ‘base’ equals the ancestor class, in our case it is ‘first’. So using the keyword 'base' we won’t see any newly added fields, but the old ones defined inside the ancestor class. Thus, the 'base.b' will reference the int-typed field ‘b’ defined in the ancestor class:



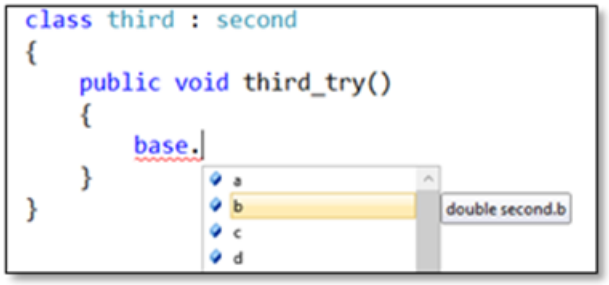
Note that with the 'base' keyword we still cannot access the field ‘a’. The ‘base’ is written into the class ‘second’, where the scope of the field ‘a’ is not extended. (So there is no backdoor!)

As the 'base' sees a 'second' type instance:



According to the 'base' the field 'int a' is unavailable due to its protection level. The double fields cannot be seen as they simply do not exist for a ‘base’. However, the field 'int b' and 'int c' defined in the ancestor class can be referenced.

The 'base' has a serious restriction: it can only access our direct ancestor. If we would create a class 'third' as a child of the class 'second', then this third class would exactly have the same 7 field. In the ‘third’ class we would write ‘base’, which (in this case) would reference back to its ancestor ‘second’, so in that case the ‘base.b’ would be the double field defined in the second class.



So with the help of 'base' we can go backward only one level in our list of ancestors. This does not mean that in the class ‘third’ we are not able to reference to this ‘int b’ field (as eventually it still exists for a third class instance). With the help of the 'as' operator it still can be solved.

11.4. 11.4. Inheritance of the methods

The rules of method inheritance are very similar to the rules for field inheritance. The private methods are inherited, although it cannot be called from the child class. The public and protected methods are inherited as well, but they can be used and called.

```
class first

{

  private int a = 1;

  protected int b = 1;


  public void writeIt()

  {

    Console.WriteLine(" first.writeIt ");

  }

}
```

We create a second class as well:

```
class second : first

{

   public double b = 2.0;


   public void test()

   {

      writeIt();

   }

}
```

Inside the 'test()' method we call the inherited 'writeIt()' function. We can do this as the protection level of this function is public, so the child class may also call it. If we create instances, these functions can be called:

```
first e = new first();

e.writeIt();
```

```
second m = new second();

m.writeIt();

m.test();
```

Through the instance 'e' the function ‘test()’ cannot be called, because of the type of ‘e’ does not contains such a function. This is not due to the fact that creating an 'e' instance a test method is not created along with; do not believe that, as the instantiation has no connection to the methods. During an instantiation only the fields goes into the memory, and the constructor is called. The problem is that the:

```
e.test();
```

would be interpreted (according to the compiler) by the following way:

```
elso.test( e );
```

There we can see two problems. One is that there is no function ‘test()’ inside the class 'first'. Second, although there is a 'test ()' function in the ‘second’ class, but it requires a parameter of 'second' type instance, which 'e' is not.

The inheritance of the properties work the very same. If we have a public or protected property, in the methods or in other properties in the child classes we can call them.

It is very important to repeat again and again: the instance's memory size depends on only the number and type of fields. Methods (either inherited or added as new) does not require additional memory for the instances! It also does not count the class-level constants and fields! During the creation of a new instance a memory area for a new set of instance-level fields is reserved. The fact that before the instantiation no instance level method can be called – the syntax trick of the compiler only. Do not think that as the result of the instantiation we will have new fields and methods as well. The methods were available before, they are loaded into the memory at the start of the program, simple we could not call them! The methods are loaded in the memory only once, even there is not a single instance, and even once when there are dozens of instances.

11.5. 11.5. The problems around method inheritance

Methods inheritance works very similar to the inheritance of fields. The problems begin when we inherit a method and try to develop a new one with the same name. Need to consider the following:

· In the base class is this method *private*? In this case we have no problems at all, since the private method cannot be accessed or called in the child class, so it knows nothing about the presence of such a method. A new method can be created in the child class with the same name, but calling the inherited one directly is impossible.

· The method is not private, we try to create a method with the same name but the new method has *different parameters*? In this case again we have no problems, as it does not count the same name until the parameter list is different, the two names mean two different methods this time, the scopes won’t overlap. When we call one of the methods, in addition to the name we must specify the parameters as well. So the compiler can decide which method is called. This is almost the same as there will be two methods with different names. This rule was called the “overloading rule,” but this works in the OOP world as well.

· It is not private, same name, same parameters? Well, the real problem starts here!

First question we must clarify: why we want to do that in the child class at all? We inherit a method, but we want to write another with the same name and same parameterization into our own class? This situation is very similar to the one discussed previously, but it was the error of the design when we inherited a field and wanted to create another one with the same name.

According to methods this situation almost never indicates an error, this is intentional, and is used in order to avoid the errors. Think about what would happen if our class ‘square’ would holds a method to calculate the perimeter, then the rectangle child class would inherit this:

```
class square

{
```

```
  public double a side;


  public double perimeter()
  {
     return 4 * a_side;
  }
}


class rectangle : square
{
   public double b_side;
   // + inherited field a_side
   // + inherited method perimeter()
}
```

The ‘square’ instances work well. What about the ‘rectangle’ instances?

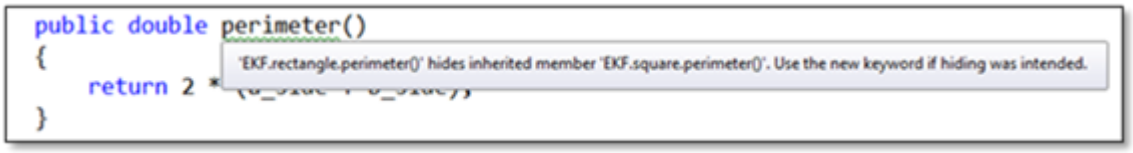
```
square n = new square();
n.a_side = 12;
double nk = n.perimeter(); // 48
// ---
rectangle t = new rectangle();
t.a_side = 10;
t.b_side = 20;
double tk = t.perimeter(); // 40 !?
```

The perimeter() function of the ‘square’ class calculates its value by 'a_side' is multiplied by 4, that's fine. The rectangle method inherits the perimeter() method, but it still calculates the same way, so that the case of a rectangle is not a 2 × (10 + 20) but still 4 × 10 (4 × a_side). What is the solution?

First thoughts: write your own perimeter() method. The idea is good, the implementation is the problem as the VS stresses it and indicates an error:

```
class rectangle : square
{
    public double b_side;
    // + inherited field a_side
    // + inherited method perimeter() -- which is not good for us
    public double perimeter()
    {
        return 2 * (a_side + b_side);
    }
}
```

The error message is "*rectangle.perimeter() hides inherited member square.perimeter()*" - almost the same as we have seen at the case of fields:



The solution is also given in the error message: "*use the 'new' keyword if hiding was intended*". It is the same thinking as was described for the fields. The VS is disturbed again, it thinks we wrote the same method that was inherited. The 'new' calms VS down, indicates this was intentional:

```
new public double perimeter()

{

  return 2 * (a_side + b_side);

}
```

The code in the outside world starts to work fine from now:

```
square n = new square();

n.a_side = 12;

double nk = n.perimeter(); // 48

// ---

rectangle t = new rectangle();

t.a_side = 10;

t.b_side = 20;

double tk = t.perimeter(); // 60 good!
```

When a child class inherits a protected or public method, has the right to develop a method with the same name and same parameterization. This method is an improved version of the inherited one. This new method covers, hides the inherited method, so at the new method declaration the 'new' keyword must be added.

11.6. 11.6. The methods and the 'base'

The 'base' keyword was introduced at the filed inheritance, which allowed us to refer to the ancestor class fields, even if the child class has overwritten them. Very similarly we can call methods or properties created in the ancestor class even if the child class has a newer version of them with the same name and same parameterization.

Consider the following example! Let us create an object class that collects numbers into a list, but only the positive ones. In addition, provides a read-only property to read the sum of the numbers.

```
class positive_numbers

{

  protected int summa = 0;

  protected List<int> collector = new List<int>();

  public void add(int num)

  {

     if (num< 0) throw new ArgumentException("only positive");
```

```
     else
     {
        collector.Add( num );
        summa += num;
     }
   }
   public int total
   {
      get { return summa; }
   }
}
```

As a further development of the class we want to modify the behaviour that a maximum of 30 numbers will be collected. To do this we overwrite the method 'Add()', we have to cover the old one with our new version so the old one cannot be used anymore (with the help of the old one more than 30 number can be added which would violate our new rule):

```
class adv_positive_numbers : positive_numbers
{
  new public void add(int num)
  {
    if (collector.Count > 30) throw new Exception("too much number");
    else base.add( num );
  }
}
```

At the 'else' part the parameter still cannot be added to the collector list, because it should also have to check whether the value is positive or not, and increase the amount of numbers (the summa field). These are done by the inherited method, just need to call with the help of the 'base'.

Let us see a similar problem. In this we want to develop a program which works with fishes. First we want to create a fish class, as we would have several fish instances in our fish tank. Every fish has its own weight (in kilograms, fractions also accepted). Because we would have all kind of fishes, for the weight we accepted any value which is greater than zero and eg. less than 120 kg. Prepare the fish class with the weight field (protected) and a public property:

```
class fish
{
   protected double _weight;
   public double weight
   {
      get { return _weight; }
      set
      {
```

```
        if (value < 0 || value > 120)
          throw new ArgumentException("cannot be accepted");
        else _weight = value;
      }
   }
}
```

In the child class we add a new "alive" filed, which is a boolean type field. If the fish is not alive, then the weight value through the property cannot be changed. However, if it is alive, the weight will work as before:

```
class liveFish : fish
{
   protected bool isAlive = true;
   new public double weight
   {
      get
      {
         if (isAlive) return base.weight;
         else throw new Exception("is not alive");
      }
      set
      {
         if (isAlive) base.weight = value;
         else throw new Exception("is not alive");
      }
   }
}
```

11.7. 11.7. The real problems around the inheritance of the methods

Methods inherence contains additional problems. To illustrate one of them, let us think on if the square class is complemented by a 'writePerimeter()' function as shown below:

```
class square
{
   public double a_side;

   public double perimeter()
   {
      return 4 * a_side;
```

```
   }


   public void writePerimeter()
   {
       Console.WriteLine("perimeter = {0}", perimeter() );
   }
}
```

This method is inherited by the rectangle class, which overrides the perimeter method:

```
class rectangle : square
{
   public double b_side;


   new public double perimeter()
   {
      return 2 * (a_side + b_side);
   }
}
```

Consider what will happen and be written out in the end in the main program:

```
square n = new square();
n.a side = 12;
n.writePerimeter();
// ---
rectangle t = new rectangle();
t.a_side = 10;
t.b  side = 20;
t.writePerimeter();
```

The code is syntactically correct; but does not happen the same as what we would expect. To further enhance the problem, we present another piece of code, which won't work as we expected:

```
static void writeItsPerimeter(square p)
{
  p.writePerimeter();
}


public static void Main()
{
  square n = new square();
```

```
  n.a side = 12;

  writeItsPerimeter( n );

  // ---

  rectangle t = new rectangle();

  t.a_side = 10;

  t.b side = 20;

  writeItsPerimeter( t );
```

12. 12. Type compatibility

Imagine that we have a class, and an instance 'p' made from this class. This 'p' has a number of fields and methods. We create a child class, and from it an instance 'k'. Let us examine the relationship between the two instances:

- If 'p' has a kind of field, then that field is owned by ‘k’ as well! Why? Because ‘k’ has inherited it!
- If 'p' has a property, the same property is presented in 'k' as well! Why? Because ‘k’ has inherited it!
- If the 'p' has a method, the same method can be called using 'k' as well! Why? Because it was inherited!

In summary we can conclude: the child class instance the 'k' will have all the fields, methods, properties which ‘p’ has as well. This is a simple consequence of the rules of inheritance, and the fact that the child cannot decide about what to inherit and what notto. One cannot select fields, methods that "huhh I do not need them thanks a lot, others are welcome". Although it often would be very useful, but then we would lose this opportunity we are discussing now.

At present in OOP we can see that a child class contains at least as many items (methods, fields, properties, etc.) as its ancestor class. So where an instance from the ancestor class ('p') can be used (because of its ability to store data, a method of it is able to perform a task) – there a child class instance ('k') also suitable for the same task! What the 'p' was able to store the 'k' will also be able to store with a field exactly the same name and type. What activities 'p' could perform, the 'k' will also be able to perform with exactly the same named method with the same parameterization.

In such situations in real life we say ‘k’ is a professional substitution of ‘k’. In the IT world the proper world is not the substitution but the compatibility, so it can be said that 'k' is compatible with 'p'. Actually, an instance of the child class can replace at any place any time an instance of the ancestor class, so we might extend the compatibility to the type level: the child class as a type is compatible with its ancestor class as type.

Because of the inheritance a child class inherits all the fields, methods, properties from its ancestor class. For this reason the instances of the child class is able to substitute the instances of the ancestor class. In general, **a child class is compatible with its ancestor class**!

If a class 'A' has a child class 'B', then 'B' is compatible with the 'A' class. If 'B' has a child class 'C', then 'C' is compatible with 'B'. Question: is 'C' compatible with 'A' as well?

```
class A { /* ... */ }

class B : A { /* ... */ }

class C : B { /* ... */ }




public static void Main()

{
```

```
  A a;

  B b;

  C c = new C();

  b = c;

  a = b;
```

Since 'C' is compatible with 'B', so the 'b = c' assignment is correct. Since the 'B' is compatible with 'A', the 'a = b' also correct. What will be the value of variable 'a'? It is the value of 'b'! What is in the 'b'? It is the value of 'c'. That is, ultimately, the 'a' will be equal to 'c'. Remember, any variable in this example is a reference type variable, so only memory addresses are passed!

```
A a;

C c = new C();

a = c;
```

This assignment can be written in a short way like ‘a = c’. Since the 'c' is compatible with the 'b', and ‘b’ is compatible with ‘a' (at the type level), so the ‘a=c’ is correct at the type level as well. We may add more explanations. Consider: a 'c' instance have all the fields, methods, properties, which has a type 'A' instance. How? It inherited from the class 'B' who inherited them from the class 'A'.

The type compatibility is transitive relation, the classes are compatible with their ancestor classes, not only with the direct ancestor, but also by all the ancestors up to the beginning. The classes are compatible with all their direct and indirect ancestors!

12.1. 12.1. The consequences of the type compatibility

The type compatibility is a key concept; the rule of the assignment statement uses this concept mainly to decide if the statement is correct or not. Repeat the rule:

The assignment statement is correct, if the left hand side variable type matches the type of the right hand side of the expression, or the type of the expression is compatible with it.

Accordingly, if the ‘rectangle’ class is the child class of the ‘square’ class, the following statement is correct:

```
rectangle t = new rectangle();

square n = new square();

n = t;
```

More simply:

```
rectangle t = new rectangle();

square n;

n = t;
```

A little bit more simply:

```
rectangle t = new rectangle();

square n = t;
```

The simplest:

```
square n = new rectangle();
```

Let us consider why we work with the above variable assignments. First, notice that in each case the left side 'n' is the type 'square'. Each assignment statement has its right side ultimately a 'rectangle' type of instance - and

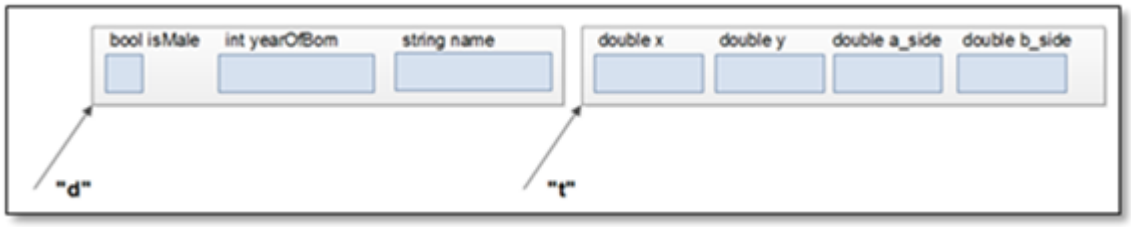
rectangles as described previously are type compatibly with the 'square' type. Thus, each assignment statement is correct!

Remember all the types created by the keyword 'class' are all reference types. The instance variables require only 4 of bytes memory, and store only a memory address. Therefore, technically considering an 'e' and 'f' instances, it does not matter what kind of instances they are, physically either 'e = f' and 'f = e' assignments operations should work; the assignment operator in this case copies only the 4 bytes of memory address from one to the other.

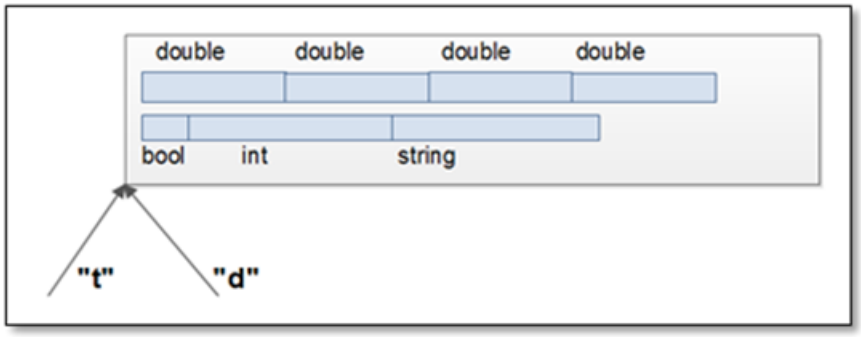
What would happen if the compiler would go without any type checking? Let's look at the following examples! Suppose we have a student and a rectangle class as follows:

```
class student

{

  protected bool isMale;

  protected int yearOfBorn;

  protected string name;

  // ... other fields ...

}


class rectangle

{

  protected double x;

  protected double y;

  protected double a_side;

  protected double b_side;

  // ... other fields ...

}
```

Create an instance 'd' as a student, an instance 't' as a rectangle. Then the memory state looks like this:



If we would have executed a 'd = t' assignment, then we would copy the content of 't' (the memory address) to 'b', so after this operation 'd' would point to the same memory area where 't' was pointed:

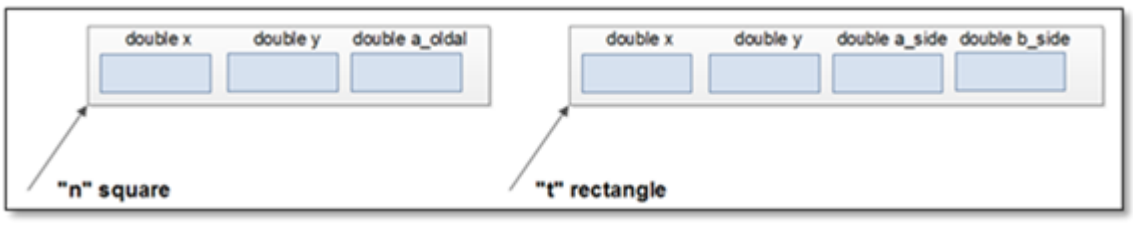


Thus, this memory area could be interpreted in two ways. The 'd.isMale = true' assignment would set the boolean field to true, which is the part of the rectangle 'x' field in the first byte (where the first byte of the double value is stored). After that a 'double x = f' assignment (which reads the double value of the field) would produce a very unattended result. Obviously, this is not allowed.

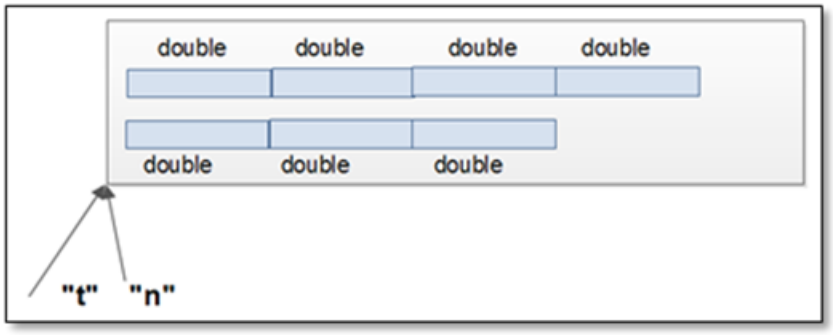
However, the type checking does not allow this:

```
class square

{

  protected double x;

  protected double y;

  protected double a_side;

  // ... other fields ...

}


class rectangle : square

{

  protected double b_side;

  // ... other fields ...

}
```

Then the ‘rectangle’ inherits the fields from the ‘square’, so the memory structure of a ‘t’ instance of ‘rectangle’ looks the same as an ‘n’ instance of the ‘square’ class (at least at their first part):



Due to the type compatibility rules the ‘n = t' assignment is executable. After that variable 'n' will point to the same memory area where ‘t’ is laid, but through ‘n’ only the 1st three fields are available (the' 4th field, the ‘b_side’ cannot be accessed as an instance of a rectangle has no such field):



If we set a new value to 'n.x', then the value of 't.x' also modified since the two fields are stored at the same place in the memory. It is very important that the two 'x' fields are the same type (inheritance), and their position within the structure, in the memory is the same. It is therefore safe.

Let's see the previously presented assignments!

```
rectangle t = new rectangle();

square n = new square();

n = t;
```

First we create an instance of rectangle, and stores its memory address at variable ‘t’. Second, we create an instance of a square class, then stores the memory address to 'n'. In the third step, executes the ‘n = t' assignment, where we copy the value of ‘t’ (the memory address) to ‘n’. With this step we override the previous value of 'n', which was a memory address as well. This previous memory address is lost. We forget the memory address of the square instance; however that instance still allocates its memory area. Later the Garbage Collector will find that area and free up later. However, it is still useless to allocate memory that are for nothing, we won’t work with that area, and the GC is loaded meaninglessly.

```
rectangle t = new rectangle();

square n;

n = t;
```

In this example the 'n' is declared only without the useless memory allocation. In the next statement it receives its 1st value, the memory address from ‘t’. There is only one memory allocation for an instance of rectangle, and two variables, the 'n' and 't' stores the same memory address.

```
rectangle t = new rectangle();

square n = t;
```

In the third experiment, the declaration of ‘n’ is simply merged with the initial assignments.

```
square n = new rectangle();
```

In this latest case we got the same. Only here we eliminated the variable 't', and the memory address of the newly created rectangle is copied directly to the variable 'n'. An assignment in this form of seems meaningless, but works!

It seems useless to create a rectangle instance with 4 double fields, and use the rectangle constructors to initialise this object when we are able to access only three fields through the variable 'n', and only those methods can be called, which are accessible for a square class instance.

12.2. 12.2. The Object class

When we do not declare an ancestor class explicitly to our class, we might think it has no ancestor class at all. But there is, as the compiler uses the default base class as the ancestor, which is the ‘Object’ class.

Since a child class is compatible with its ancestor classes, so the classes without explicitly set base class are compatible with the ‘Object’ class. However, if we select and declare an ancestor class, we are compatible with that selected class, and all of its ancestors (because of the transitivity). One can be sure at the end of this line there is the ‘Object’ class, so basically all the classes declared in the C# language is compatible with the ‘Object’ class.

In the C # language, every object class is compatible with the ‘Object’ class which is inside the ‘System’ namespace (System.Object). The alias name of this class is 'object' –it is often found in the source code with this alias name.

In other words, we write this - and the compiler reads this:

All classes are compatible with the Object class: it has important consequences. An object-typed variable can be used any time on the left-side of an assignment. In this case an expression of any type can be used on the right side – as any type is compatible with the object type (it is guaranteed that the type of the right side is compatible with the left side):

```
object ne = new square();

object donald = new duck();

object number = 2 * Math.Sin(30 * Math.PI / 180);
```

It is in this form it still has no sense still, but if we continue the investigation, well, many possible applications will be found.

12.3. 12.3. The static and the dynamic type

Let us examine this assignment:

```
square n = new rectangle();
```

It's clear that the type of 'n' (as for its declaration) is 'square', while (as noted above) the instance, whose memory address is stored in this variable is a full-featured ‘rectangle’ (it has all the fields of a rectangle, initialized by a constructor of a rectangle).

What is the type of 'n' in real? Is it the ‘square’ type, or rather ‘rectangle'?

We must get familiar with new concepts this time. The type of the declaration is very important, and usually it is the same as the type of the created instance:

```
Random rnd = new Random();
```

When the two types are different, we must name them. The type comes from the declaration named "static type", and the type of instance which is held as a memory address is named “dynamic type” (real type). Thus, in the examples above the static type of ‘n’ is square, the dynamic type is rectangle. Both the static and dynamic type is Random for the variable ‘rnd’.

Obviously there is no problem as long as the two types are the same. Then every function works as expected. However, many interesting questions are raised by the case in which the two types are different.

In the example at the end of chapter “11.7 The real problems around the inheritance” we tried this code which did not work as expected:

```
static void writeItsPerimeter(square p)

{

  p.writePerimeter();

}


public static void Main()

{

  square n = new square();

  n.a_side = 12;

  writeItsPerimeter( n );

  // ---

  rectangle t = new rectangle();

  t.a_side = 10;

  t.b_side = 20;

  writeItsPerimeter( t );
```

In this example above an exciting case can be seen. The function ‘writeItsPerimeter‘ expects a parameter ‘p’ of a type 'square'. At the places of calling this function we can give it a square instance (for example ‘n’). In the background this time the ‘p=n’ assignment statement is executed which we formulate as ‘p picks up the value from the calling side’. This assignment is executable (is good) as this is type-correct (their static type is the same). Furthermore, to this function an instance of ‘rectangle’ is also passed. In that case 'p = t' assignment is executed, and, as we have seen earlier, can be executed due to type compatibility rules. The static type of 'p' is still 'square' (it is declaration type in the parameter list), but the dynamic type is rectangle, since in the second case it holds a memory address of a rectangle instance.

12.4. 12.4. The operator ‘is’

As we have seen, there is a case when we declare a function parameter to a given type, but do not know what is its real (dynamic) type. Of course, we can be sure of one thing: this real type maybe the same as the declaration type, or it is one of its child class. But we might know nothing more.

Consequently a question arises: is there a possibility inside of a function to examine what the actual type is? What is the actual (dynamic) type of the parameter?

Well, with the help of the 'is' operator there is. Using the 'is' operator we can detect the real type:

```
static void writeItsPerimeter(square p)

{

  if (p is rectangle)

  {

    // a rectangle was given

    Console.WriteLine("it is a rectangle");

  }

  else

  {
```

```
    // it is not a rectangle

    Console.WriteLine("it is (probable) a square");

  }

}
```

The 'p is rectangle' is a bool typed expression. Especially, it examines whether the dynamic type of ‘p’ is compatible with the type ‘rectangle’. Who is compatible with the 'rectangle’ type? Even the 'rectangle’ class itself, and all of its the child classes (at any depth).

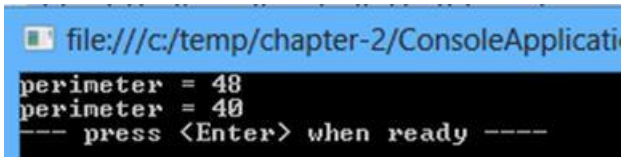
12.5. 12.5. Early binding and its problems

Let us return to the problem described above. There was a method inside the class ‘square’ named ‘writePerimeter’, which called 'perimeter()’ function to calculate the value of the perimeter, and printed the resulted value to the screen. In the 'rectangle’ class we redefine this function, as the perimeter of a rectangle must be calculated differently, but the ‘writePerimeter’ function remained untouched. The main program we instantiate both the square and the rectangle class, these instances are passed to the ‘writeItsPerimeter’ function, which calls the instance function to write the perimeter value.

```
static void writeItsPerimeter(square p)

{

  p.writePerimeter();

}
```

If we start this program, it turns out it is not working properly:

```
square n = new square();

n.a_side = 12;

writeItsPerimeter( n );

// ---

rectangle t = new rectangle();

t.a_side = 10;

t.b_side = 20;

writeItsPerimeter( t );
```



Perimeter of the square is good, 4 × 12, but the rectangle is wrong: it should be 2 x (10 + 20), but instead it is 4 × 10 according to the writings.

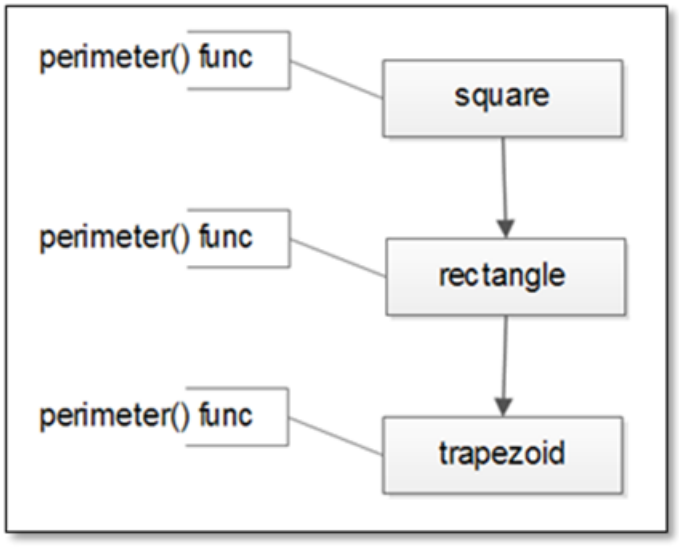
Why? In the 2nd case a rectangle instance is passed to the function, and the perimeter function has been overridden to correct the calculation expression, it is supposed to work well! The problem is due to **Early Binding**.

In the non-OOP programming approach according to the rule “overloading” it may occur that in the in code two or more functions with the same name exists, but obviously the parameterization is different. When we call a function with its name, the compiler must decide which one is it. This process of decision is called **binding**.

During the binding process the compiler connects together a function call and a particular function.

The binding of the OOP environment binding is a much more complex problem. In this case we inherit method, and we create a better version of this method with the same name and with thesame parameterization. So the same method and same parameterization exist several times in the code.

Suppose that there is an class whose ancestors has a 'perimeter()' function without any parameter. We in all the child classes override this method using the ‘new’ keyword as described above:



When we instantiate from the ‘trapezoid’ class, and calls the ‘perimeter’ function, the answer seems clear to the question: which method will be executed?

```
trapezoidoid tr = new trapezoidoid();

double perim_trap = tr.perimeter();
```

For the instance 'tr' actually three different perimeter methods exist, as it has inherited two from the square and rectangle classes, and has developed a new one for its own. We think that it the newest one from the trapezoidoid class will be executed, as it is the latest (the best, the most suitable) version for a 'tr' instance. Of course, the concept of "the best" does not exist in the OOP, so we must formulate the rule rather differently. With the knowledge of the type of 'tr' the compiler must decide which method to be called. It is done by going backward (upward) in the inheritance tree presented above it. There is not such a method in the trapezoidoid class, the compiler goes one level up to find one. If during this search the compiler finds no one in no classes upward (which is suitable by its name, its parameterization and is accessible due to its protection level), the search is over, and a “method not found” error message appears (syntax error).

But which type of the 'tr' is used as a starting point? Is it its static type or the dynamic type?

```
rectangle tx = new trapezoidoid();

double tx_perim = tx.perimeter();
```

A simple test program is written to check this behaviour of the compiler. It soon turns out that the static type is used to determine the starting point of the search. So in this example the perimeter function inside the ‘rectangle’ class is used, which we can see if we use the F11 key to execute (debug) the program step-by-step.

Do not understand why it is done by this way? It is for a very simple reason. Imagine as we move these two statements into a decent distance from each other. It might be they are not in the same function block. Look at the following example:

```
static void writeIt(rectangle p)

{

  double dk = p.perimeter();
```

```
  Console.WriteLine(dk);

}



public static void Main()

{

  trapezoidoid tr = new trapezoidoid();

  writeIt( tr );
```

We can see that to the rectangle "p" holds a memory address of a trapezoidzoid instance, so as in a previous example its static type is rectangle while its dynamic type is trapezoidoid. We call the perimeter function of this ‘p’. Which function is to execute?

Do not expect miracles in the compiler program. The compiler often behaves as if it does not read the program lines one-after-another, but read and interpret them independently. What was read and understood in the previous line and, in the next line is forgotten. Just turn the focus to the actual statement and tries to make the most of it.

```
double tx_perim = tx.perimeter();
```

To understand this line the compiler relies on only what he sees in this line. It knows that the static type of ‘tx’ is rectangle. The compiler already forgot that in the previous statement we assigned ‘tx’ to a trapezoidoid instance. So to decide which ‘perimeter()’ function is needed, in this line knows and sees only the static type, it can rely on only the static type. To start searching in the tree backward the rectangle level is used. As there is one such method in this level (override with the ‘new’ keyword), the search is over, and the binding is complete.

```
double tx_perim = rectangle.perimeter( tx );
```

So we can understand in this example why the rectangle class is the main point:

```
rectangle tx = new trapezoidoid();

double tx_perim = tx.perimeter(); // = rectangle.perimeter( tx );
```

At the case of ‘writeIt()’ function it is easier to see that there is no other way. There is no information about the dynamic type of ‘p’, only the static type is known. So based on this information only the static type (the rectangle) is used to determine the start point. The function call ‘p.perimeter()' means the calling of the method is defined in the rectangle class.

Still there is the question: why the dynamic type does not count? Well, the compiler likes to make decisions. In the compile time (when there is plenty of time) the compiler thinks a lot, searches a lot, examines the static type and makes a decision to select a method to call. It is good for us because this means it is done at the end of the compilation, and needs no extra time during the execution of the program. So the code runs at a very high speed. When the program reaches a point during the execution to call of this method call, it simple jumps to the selected method and continues.

If it would not work this way, then no other decision could be made here about what method to be called (at compile time). The only way isto wait until the program starts and reaches this point. Then (at runtime) check the actual (dynamic) type of ‘p’, and makes this decision then. Since this is a run-time event, it is easy to see that it will slow down the program execution. How many times? Every time when it reaches this point! No matters how much this happens. The dynamic type of 'p' can be different each time, so each time it must be checked again-and-again.

We should know that there is a chance to ask the compiler to behave like this way - but we need to give a sign to the compiler to ask this. The keyword 'new' does not hold this sign.

The binding is the process by which the compiler links method calls the appropriate method. In doing so, the compiler examines the static type of the instance. The decision is taken at compile time, so no extra time is

needed during runtime. Moreover, it provides the maximum running speed. This behaviour is called the **Early Binding**.

In the OOP world the early binding is a heritage essentially coming from the older programming style (or concept). In the modular programming style the ‘dynamic type’ concept is simply unknown, so another binding is not necessary. In the world of OOP the early binding is the source of many errors, as without a deep understand of the binding and how the compiler works we cannot recognize and cannot understand what is happening inside our code, and why it operates in an unwanted way.

13. 13. The virtual methods

In OOP it is easy to recognize and understand the problems of method override. This can be done by using the 'new' keyword - but does not provide the necessary flexibility. To gain a maximum profit from the type compatibility we can use an instance variable, whose static and dynamic types are different. It can be seen that the static type is important and critical during a method call, so it can easily happen that the wrong version (not the latest one) method is called.

It was discussed previously than we have the opportunity to ask the compiler not to use the static type to make a decision. We agree to pay the price for this as a lower running speed, as the compiler must delay this decision to a later time. This will take affect during runtime, and then we will have more information to make a better decision. In this case we can base on the dynamic (actual) type, which is unknown in the compile time. Knowing the dynamic type the latest version of the method can be selected and called. This way of binding is called “**late binding**”.

To indicate our choice (using the late binding feature) we must not use the ‘new’ keyword overriding the methods. It is a very good keyword to “calm down” the compiler, but it will still use early binding to call these methods.

To ask the other (late) binding methodology to our specific methods, we must use special keywords. We must be familiar with two key words: 'virtual' and 'override'.

A 'virtual' modifier must be used in the 1st version of the given method, in the ancestor class (in our example into the ‘square’ class). It must be used when this method (name and parameterization) is not inherited from any other ancestor class, instead we introduce this firstly in the inheritance tree (similar to the using of ‘new’, the ‘virtual’ must be used when we must not use the ‘new’ as it is the 1st appearance of this method).

A 'virtual' indicator holds information to the compiler itself. It shows that this method will be overridden in the child classes with high probability. So it orders the compiler that the binding process targeted to any version of this method (name and parameterization) not the early binding but the late binding should be used.

```
class square

{

   public double a_side;



   virtual public double perimeter()

   {

      return 4 * a_side;

   }
```

The 'override' keyword is used in the child classes (not the *new*!), to override an inherited virtual method.

```
class rectangle : square

{

  public double b_side;
```

```
  override public double perimeter()
  {
     return 2 * (a_side + b_side);
  }
}
```

This means that the 'virtual' keyword is used only once in the base class, and in all the child classes the 'override' keyword is used (every time when they want to override this method):

```
class trapezoidoid : rectangle
{
   public double c_side;
   override public double perimeter()
   {
      return 2 * c_side + a_side + b_side;
   }
}
```

13.1. 13.1. The override and the property

The virtual + override pair can be used with more than just methods. Since a property is a kind of a method, these keywords can be applied to properties as well:

```
class duck
{
   protected double  weight;
   virtual public double weight
   {
      get { return _weight; }
      set
      {
         if (value <= 0) throw new ArgumentException("cannot be accepted");
         else _weight = value;
      }
   }
}
```

In the child class when we want to redefine a property, it can be done only by redefining one of the two parts only, it is not needed to redefine both parts at the same time. So it is acceptable to override only (for example) the 'get' part – which includes that the ‘set’ part remain unchanged:

```
class adv_duck : duck
{
   protected bool _is_alive = true;
```

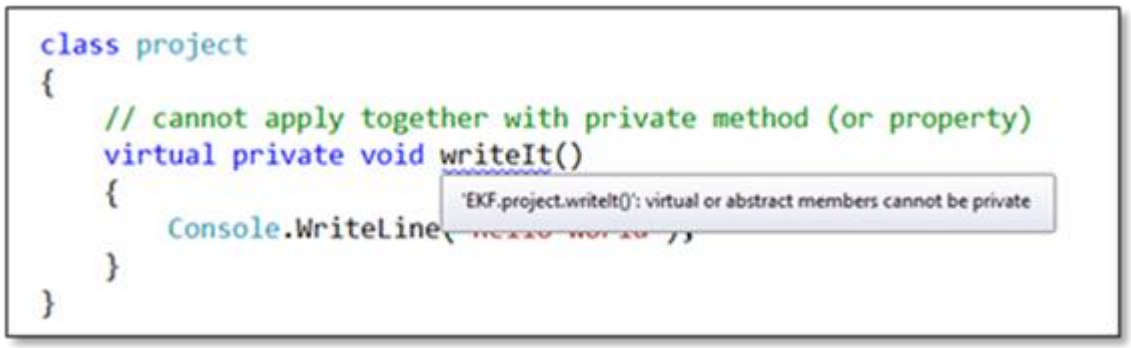
```
   public override double weight
   {
      get
      {
         if (_is_alive == false) throw new Exception("is not alive");
         else return  weight;
      }
   }
}
```

13.2. 13.2. Other rules of override

We must examine the following situations in the aspect of virtual + override:

- private,
- fields,
- class-level methods and properties,
- modified parameter list.

Do not need to point out that the 'private' protection level (for a method or property) is not suitable to apply together with the virtual modifier. These methods and properties are inherited by the child classes, but because of this strong level of protection they cannot access them. It is not possible to override (redefine) them. The error message "*project.writeIt(): virtual or abstract members cannot be private*" informs us about this fact:



It is also not an override when in the child class we keep the name of the method but modifies its formal parameter list. The override method can change only the body of the method, neither the name nor the parameterization or the return value can be changed. With different parameterization the rule ‘overloading’ is our guide. The error message "*rectangle.perimeter(bool)': no suitable method found to override*” inform us about this:

```
class rectangle : square
{
    public double b_side;
    override public double perimeter( bool withChecking)
    {
        double perm = 2 * (
        if (withChecking && perm < 0)
            throw new Exception("perimeter is negative");
        else return perm;
    }
}
```

'EKF.rectangle.perimeter(bool)': no suitable method found to override

These keywords (virtual and override) simple cannot be applied to fields. To fields only the 'new' keyword can be applied indicating the overriding, but as we described above we must avoid this situation. The error message "*The modifier 'virtual' is not valid for this item*" warns us about it.

```
class project
{
    // to fields 'virtual' cannot be applied
    public virtual double length;
}
```

The modifier 'virtual' is not valid for this item

At class level the ‘virtual’ and ‘override’ cannot be applied. The main reason is that late binding needs dynamic type, which assumes a presence of an instance. Using (calling) class level methods or properties we have no instance at all, so there is no dynamic type. The error message "*static member project.writeIt() cannot be marked as override, virtual, or abstract*" talks about this:

```
class project
{
    // to static methods (and properties) cannot be applied
    public static virtual void writeIt()
    {
        Console.WriteLine("hell
    }
}
```

A static member 'EKF.project.writeIt()' cannot be marked as override, virtual, or abstract

Let us examine the following situation: a child class inherits a method and wants to redefine it. However, the in the base class the ‘virtual’ was not applied to this method, therefore the ‘override’ is not applicable to the child class, but the 'new' is. However, we know that the 'new' does not indicates the using of late binding, so that the program may not work correctly, as it might use an older method version that can be selected to call. Unfortunately there is nothing we can do in this case. We cannot go back to the base class to add the ‘virtual’ keyword when the original developer did not add it.

To avoid this situation, in the Java programming language and override the virtual keyword simply does not exist, but are used automatically by the compiler. Each method gets the virtual modifier as it appears for the 1st time, and will include the ‘override’ modifier in the child classes when they define (redefine) a method with the same name and same parameterization.

We might think it has only advantages. One thing we must keep in mind: the main role of virtual and override keywords is to instruct the compiler to apply he late binding to all the method calling targeted to these methods. However, the late binding causes slower execution speed. We do not care if we have to choose between the high speed and the proper operation of the program, the decision is clear, we must choose the late binding and the proper operation. In Java according to this automatism, however, we cannot choose the early binding which means we cannot choose its high speed operation, all method call uses the slow late binding mechanism.

13.3. 13.3. Manual late binding – the ‘as’ operator

Staying at the problem, that in the ancestor class the ‘virtual’ modifier was not used, so in the child class we cannot mark our method with the ‘override’ – but we must produce the correct operation in the program.

Let us start from the *square ð rectangle ð trapezoidoid* situation. The square class has a child class (rectangle), which has a child class (the trapezoidoid). Each class has its own 'perimeter()' method, but in the ‘square’ class this method was not marked with the 'virtual' keyword.

We would like to develop a ‘writeIt()' method which receives a square-typed parameter and writes its perimeter value:

```
static void writeIt(square p)

{

  double dk = p.perimeter();

  Console.WriteLine("Perimeter = {0}",dk);

}
```

According to the type compatibility when we have a square-typed parameter, we can pass to this function an instance of the square class, or instances from any of its child classes of:

```
square  ne = new square();

rectangle te = new rectangle();

trapezoidoid tr = new trapezoidoid();

writeIt( ne );

writeIt( te );

writeIt( tr );
```

Because of the 'new' keyword (as we cannot use the override), however, in the ‘writeIt’ function body in every case the ‘rectangle.perimeter()’ will be called (as the static type of ‘p’ is the rectangle, and the early binding is based on this information).

With the help of 'is' operator we can examine the dynamic type, so we can substitute the lack of the late binding. Try to add this:

```
static void writeIt(square p)

{

  double dk = 0;

  if (p is trapezoidoid) dk = p.perimeter();       // case trapezoidoid

  else if (p is rectangle) dk = p.perimeter();  // case rectangle

  else dk = p.perimeter();                      // case square

  Console.WriteLine("The perimeter = {0}",dk);

}
```

Although the code looks good, the ‘dk = p.perimeter()' suspiciously looks the same on all branches. If we try this code, we will see it works badly.

Let us examine one branch closely:

```
  if (p is trapezoidoid) dk = p.perimeter();        // case trapezoidoid
```

We must understand one thing: the compiler knows the 'if' statement, it knows that it contains a conditional expression and a statement (or block of statements). The compiler does not suppose any special mean between the condition and the statement. In this case, the result of the 'is' operator result is a bool value, so it can be a condition. Now the statement: the developer wants to call the ‘perimeter’ method of 'p'. How the compiler thinks about this:

1. Is the ‘perimeter()’ marked with virtual? No? Then early binding must be applied.
2. In early binding bases on the static type.
3. The static type of ‘p’ is the square.
4. So the developer wants to call the ‘perimeter’ method of the ‘square’ class!

As we see in this source code we examine in the conditional expression whether the dynamic type is trapezoidoid or not. We expect that in the statement part of the ‘if’ the compiler according to the result of this examination would call the perimeter method defined in the trapezoidoid, as this is the case when ‘p’ is a trapezoidoid. But the compiler does not work this way, it will not use this information. It understand the statement 'p.periemeter()' call free with any preliminaries, independently of any information coming from the statements or expressions before.

If we want to avoid calling a method of the square class here during the late binding, then we must think these over:

- we cannot order the compiler to use late binding, as we cannot mark virtual or override the perimeter methods,
- the early binding uses the static type in all cases,
- so we must change the static type of ‘p’!

Under ‘change static type’ we mean ‘type cast’. The type cast orders the compiler to believe that a variable belongs to another type, so its static type is different. We can type cast in a way we discussed in the previous semester: we must write a different type name before the value in round brackets:

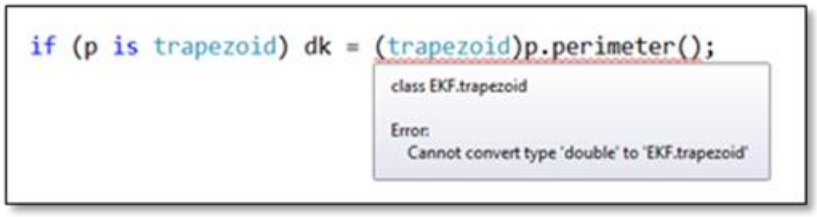
```
if (p is trapezoid) dk = (trapezoid)p.perimeter();
```

This attempt was not very successful. Let's see the reasons why:

- on the right side of the assignment operator there are three operators,
- if more than one operator is presented, the priority of the operators must be examined,
- 1st operator is the type cast,
- 2nd operator is the point operator (select a method of an instance),
- 3rd operator is the method (function) call operator, which are the round brackets at the end of the method name[14],
- the strongest is the point operator, then the function call operator, then the type cast (weakest one of them).

In a statement, the strongest operator is the point (method selection), so it is evaluated first. Second, the function call operator, then third the type cast. According to this the meaning of this statement is “call the p.periemeter

function and the return value must be casted to type trapezoidoid". The perimeter function returns a double value which cannot be casted to trapezoidoid. So the reason of the error is understandable now: "cannot convert type 'double' to 'trapezoidoid'".



When the operators are evaluated in a wrong order because of their priority, we usually change the order using parentheses. The statement above operates as the following parentheses would have been used:

We need another operation, such as the following parentheses we use:

The parentheses around 'trapezoidoid' means type cast, while the parentheses around this and ‘p’ causes to apply this type cast directly to ‘p’. From here the call of the perimeter method still uses the early binding role, but the compiler would think that the static type of ‘p’ is trapezoidoid, so the method defined in the trapezoidoid class will be called.

The double parentheses are uncomfortable, inelegant and its accidental deletion lead to misunderstandings. Fortunately, there is another way to express a type cast. It has the similar semantic, but in a different syntax. The first form of type casting is the traditional form: type name is inserted into parentheses. The second form is based on the keyword 'as':

Syntactically the 'as' operator looks very much the same as an ‘is’ operator, 1st is the name of the instance, then the keyword ‘as’, then the name of the new type to cast:

```
if (p is trapezoidoid)

    dk = (p as trapezoidoid).perimeter();//case trapezoidoid
```

So the help of this the complete code:

```
static void writeIt(square p)

{

  double dk = 0;

  if (p is trapezoidoid) dk = (p as trapezoidoid).perimeter();

  else if (p is rectangle) dk = (p as rectangle).perimeter();

  else dk = p.perimeter();

  Console.WriteLine("The perimeter = {0}",dk);

}
```

In the case of the ‘square’ the type cast is not needed as the static type of ‘p’ is already ‘square’.

13.4. 13.4. When the type cast is the only help

Because the fields cannot be marked with virtual + override, the "early binding" is used when we reference to a field. In order to access the field in a specified class, we might cast the type of the instance to another type (with one of the type cast operator, no matter which one). In chapter *11.3 The keyword ‘base’* we discussed that when a child class overrides a field the inherited one still can be referenced with the help of ‘base’ keyword, but we cannot step backward in the inheritance tree.

The type cast can be apply to the ‘this’ instance as well, in an instance level method or property. In this case we can step backward to our ancestor class level:

```
class trapezoidoid : rectangle

{

  new public double a_side;

  //

  public void trying_the_fields()

  {

     double square_a = (this as square).a_side;

     double rectangle_a = (this as rectangle).a_side;

     double trapezoidoid_a = this.a_side;

     // ...

  }
```

Supposing we redefine the field 'a_side’ in all the inheritance level, we can access the field declared in a specific ancestor. As this method is defined in the trapezoidoid class, in the function body the static type of ‘this’ is trapezoidoid, so no need to typecast ‘this’ to type ‘trapezoidoid’.

13.5. 13.5. Typecast is not an ultimate weapon

We cannot get round the type casting when we want to access fields. The situation is must be avoided (redefine fields with the 'new' keyword). On the other hand, we can define properties to get access to the fields, and properties can be marked virtual + override.

In the case of methods we easily use virtual + override marks. We may be in trouble when the developer of the ancestor class has not marked the methods with virtual, so we cannot mark ours with override. Then we can solve the problem with the help of ‘as’ operator or the other form the type casting:

```
static void writeIt(square p)

{

  double dk = 0;

  if (p is trapezoidoid) dk = (p as trapezoidoid).perimeter();

  else if (p is rectangle) dk = (p as rectangle).perimeter();

  else dk = p.perimeter();

  Console.WriteLine("The perimeter = {0}",dk);

}
```

But notice the following:

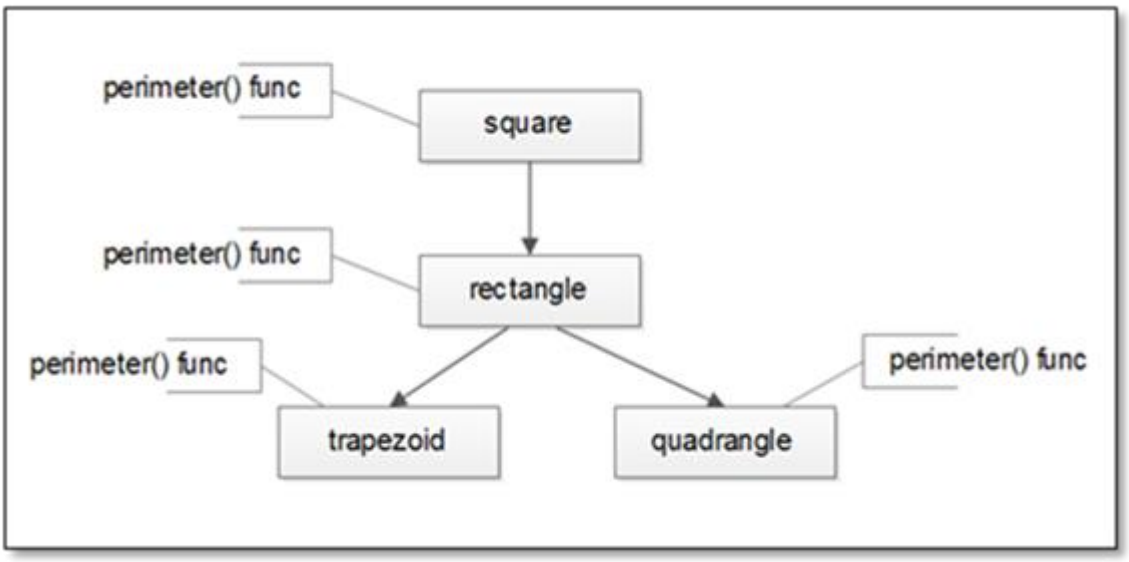
· if the method ‘perimeter()’ is originally marked with ‘virtual’, then we need no ‘if’ to examine the dynamic type (execution speed),

· in this solution we must write as many ‘if’ as many classes own a new version of this method,

· if we wrote all the if-s, but a new class is created, we must go back here to write a new if for the new class,

· only when this new class overrides this method.

Let's see what would happen if, for example, from our rectangle class a general quadrangle class is inherited. Since the quadrangle might have four sides with different length, it is natural that the perimeter method must be overridden.



What would happen if we pass a quadrangle instance to the writeIt() function, but this case is not inserted into the series of if-s?

```
static void writeIt(square p)

{

  double dk = 0;

  if (p is trapezoidoid) dk = (p as trapezoidoid).perimeter();

  else if (p is rectangle) dk = (p as rectangle).perimeter();

  else dk = p.perimeter();

  Console.WriteLine("The perimeter = {0}",dk);

}
```

Remember, the 'is' operator examines not exact type matching but type compatibility. Let's see which types the quadrangle type is compatible with? A class is compatible with its all ancestor classes, so both the rectangle and the square classes.

The first 'if' examines if it is compatible with the trapezoidoid type, but it is not (false). The 2nd examines if it is compatible with the rectangle type. We can say yes (true), so the quadrangle instance is type casted to the rectangle type, and the perimeter() method is called – the perimeter method which is defined in the rectangle class. It is bad for a quadrangle instance, so we must modify our code (as a new class which redefines this method is presented):

```
static void writeIt(square p)

{
```

```
  double dk = 0;


  if (p is trapezoidoid) dk = (p as trapezoidoid).perimeter();
  else if (p is rectangle) dk = (p as rectangle).perimeter();
  else if (p is quadrangle) dk = (p as quadrangle).perimeter();
  else dk = p.kerulet();
  Console.WriteLine("The perimeter = {0}",dk);
}
```

We added the new ‘if’ to the wrong place! As the examination if ‘p’ is compatible with the type rectangle is precedes this new ‘if’, and it evaluates to true, this new ‘if’ can be reached. The order of these ‘if’-s is very important. Because of the whole structure of these examinations, when one is success, the next one is not executed, so it still works badly.

If we have to choose this kind of solution, we must take care of the order in which the conditions are written. We must start with leaf element of the inheritance tree, then going upward to the root: (trapezoidoid | quadrangle) ð rectangle ð square:

```
static void writeIt(square p)
{
  double dk = 0;


  if (p is trapezoidoid) dk = (p as trapezoidoid).perimeter();
  else if (p is quadrangle) dk = (p as quadrangle).perimeter();
  else if (p is rectangle) dk = (p as rectangle).perimeter();
  else dk = p.perimeter();
  Console.WriteLine("The perimeter = {0}",dk);
}
```

So there is a lot ofpossibilities to make mistakes with the help of 'is' and 'as'. In addition, a larger project, we cannot guarantee that we will know each object class, and we can insert them in this order to the proper point. We cannot make a universal solution. However, if a method is marked ‘virtual’, and the compiler use late binding, the code can be written simply:

```
static void writeIt(square p)
{
  double dk = p.perimeter();
  Console.WriteLine("The perimeter = {0}",dk);
}
```

13.6. 13.6. The story of kangaroos[15]

“Mutant Marsupials Take Up Arms against Australian Air Force

The reuse of some object-oriented code has caused tactical headaches for Australia's armed forces. As virtual reality simulators assume larger roles in helicopter combat training, programmers have gone to great lengths to increase the realism of the their scenarios, including detailed landscapes and -- in the case of the Northern

Territory's Operation Phoenix – herds of kangaroos (since groups of disturbed animals might well give away a helicopters position).

The head of the Defence Science and Technology Organization's Land Operations/Simulations division reportedly instructed developers to model the local marsupials' movements and reaction to helicopters.

Being efficient programmers, they just re-appropriated some code originally used to model infantry detachments reactions under the same stimuli, changed the mapped icon from a soldier to a kangaroo, and increased the figures' speed of movement.

Eager to demonstrate their flying skills for some visiting American pilots, the hotshot Aussies "buzzed" the virtual kangaroos in low flight during a simulation. The kangaroos scattered, as predicted, and the Americans nodded appreciatively ... and then did a double-take as the kangaroos reappeared from behind a hill and launched a barrage of stinger missiles at the hapless helicopter. (Apparently the programmers had forgotten the remove "that" part of the infantry coding).

The lesson? Objects are defined with certain attributes, and any new object defined in terms of the old one inherits all the attributes. The embarrassed programmers had learned to be careful when reusing object-oriented code, and the Yanks left with the utmost respect for the Australian wildlife.

Simulator supervisors report that pilots from that point onwards have strictly avoided kangaroos, just as they were meant to."

14. 14. Problems with the constructors

The inheritance indicates a lot of questions, so we must talk about constructors again. For now, all we know about the constructors is to help initialize the fields by the parameters. A class may have multiple constructors as long as their parameterizations are different. In addition, it is important, that a class may have constructor with no parameters, and if we do not create any constructor, the system will generate a parameter less one, which does not contain any statements in its body.

If we have ancestor classes, and we start to develop a child class we must know a lot of facts about the constructors in the base classes.

The first question to answer is: at instantiation only one constructor is executed? Let us create the square ð rectangle ð trapezoidoid inheritance tree, and into the classes create a parameter less constructor which writes a line into the screen:

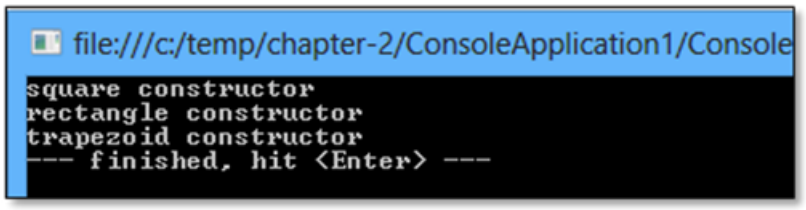
```
class square

{

  public square()

  {

     Console.WriteLine("square constructor");

  }

class rectangle : square

{

  public rectangle()

  {

     Console.WriteLine("rectangle constructor ");

  }

class trapezoid : rectangle
```

```
{
  public trapezoid()
  {
    Console.WriteLine("trapezoid constructor ");
  }
```

Let us create an instance from the trapezoid class:

```
public static void Main()
{
  trapezoid tr = new trapezoid();
  Console.WriteLine("--- finished, hit <Enter> ---");
  Console.ReadLine();
}
```

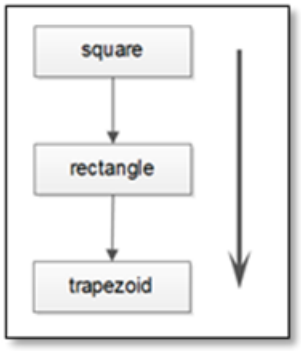
Start this program:



We can detect that not only one constructor has been executed, but many of them. We must examine this phenomenon!

14.1. 14.1. The constructor call chain

As we were able to see that in the instantiation we called only the constructor of the trapezoid class, but all the constructors from the ancestor classes are executed as well. In fact, if we carefully observe the writings, the first started the very first ancestor, then as we walk down the inheritance tree in this order the constructors of the ancestor classes. Finally, the constructor of the child class which was instantiated was executed:

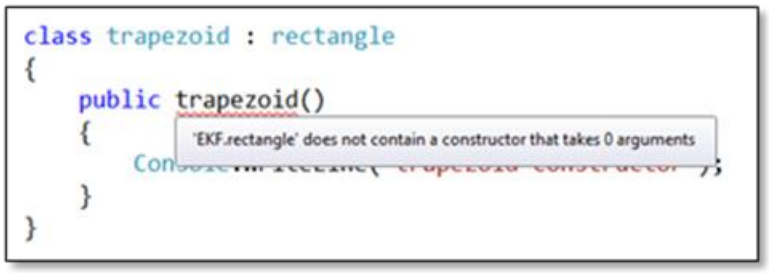


The behaviour above is called a **constructor call chain**, which means to create an instance that the constructors in the ancestor classes are involved as well. Was it accidental or intentional? To find out, for example, modify the rectangle class: add a parameter to the constructor, which allows you to set the ‘b_side’ field value:

```
class rectangle : square
{
```

```
   public double b_side;

   public rectangle(double b_side)

   {

      this.b_side = b_side;

      Console.WriteLine("rectangle constructor ");

   }
```

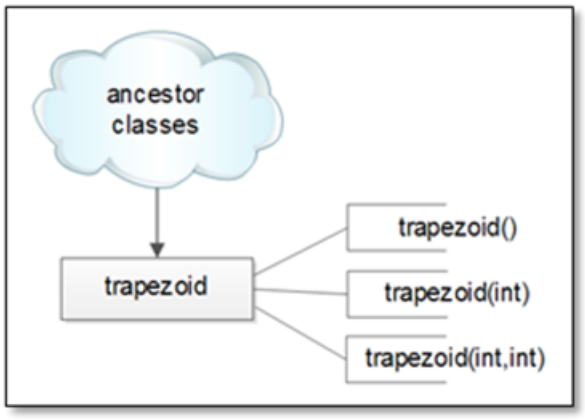
If we were to compile this program an error appears at about the trapezoid constructor:



The problem is that the calling of the base class constructor is not guaranteed, so the definition of the child class is not acceptable. Indicates that the execution of the base class constructor is not optional but mandatory! If for some reason it is not guaranteed, then it means it is a syntax error, and the program code cannot be compiled. This means it is a strong restriction and rule.

14.2. 14.2. Constructor identification chain

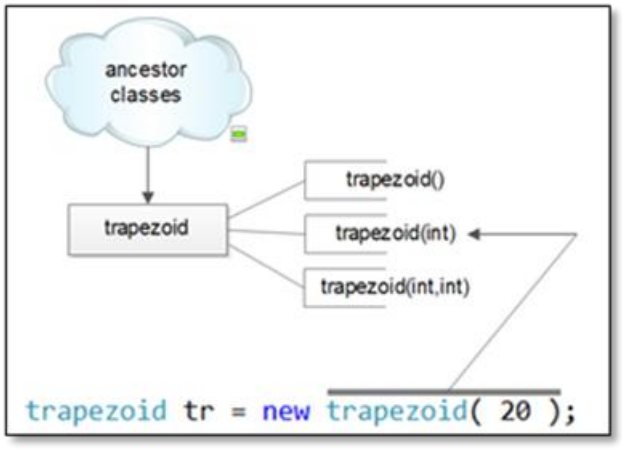
Let's see the real reason for this problem. When we want to create an instance, first we select the class; we allocate memory with ‘new’; then call a constructor. Which constructor is called? There can be many constructors for the class! Consider the class of trapezoid. Let us assume that there are three constructors:



To create an instance we select a specific constructor:

```
trapezoid tr = new trapezoid( 20 );
```

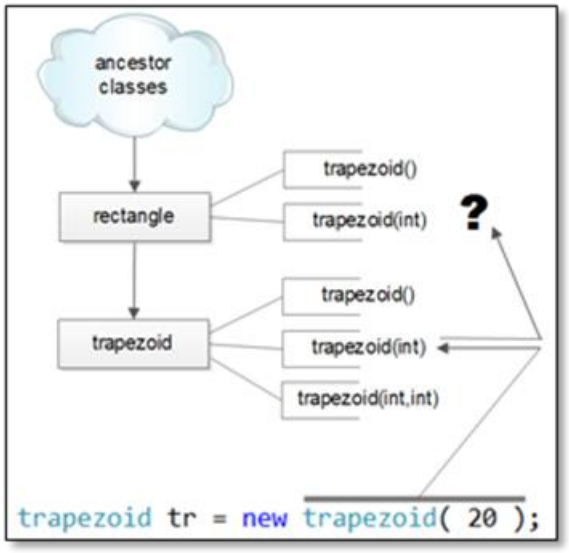
In this case, the selected constructor (according to the actual parameter list) is the one with one 'int' parameter:



The first link is identified by the constructor call. If it fails (there is no constructor with this parameterization, or by its protection level it is not accessible) a syntax error appears:

```
trapezoid tr = new trapezoid( 20, "apple" );
```

Once the compiler has identified the selected constructor, it wants to select a constructor from the ancestor class:



To select a constructor from the ancestor class the rules are:

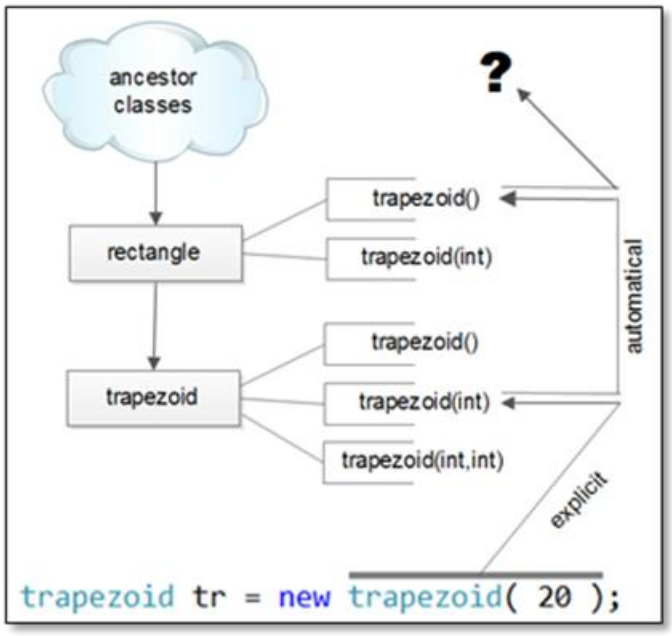
1. only public or protected constructors can be selected,
2. if there is a 'this()' constructor call (into the same class), then this is,
3. if there is a 'base()' direct constructor call (into the ancestor class), then this is,
4. if there is a parameter less (in the ancestor class), then it is selected,
5. syntax error (as for the previous 4 rule nothing is selected, a syntax error will be appeared).

Let's examine the rule #4 first. If the base class has a parameter less constructor which is available (by its protection level), then it will be selected. When we have one such a constructor? If

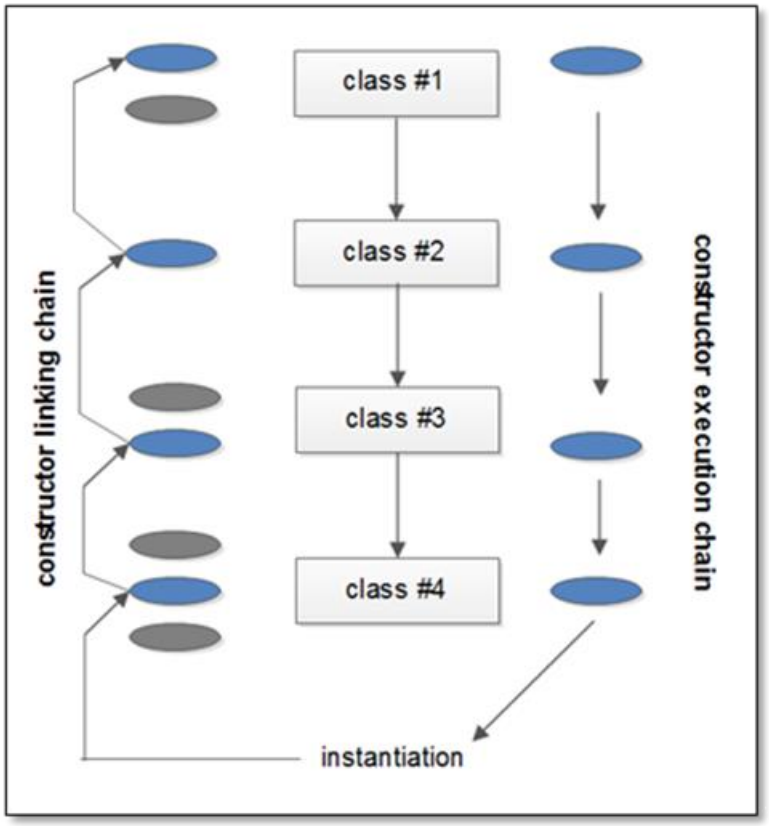
- we creates one, public or protected constructor with no parameters,

· we do not create any constructor at all, and then the system will generate a public constructor with no parameter and with empty body.

So if you write a base class constructor with no parameters, or do not write a constructor at all, then this parameter less constructor can be appended to the chain automatically.



If you have any base class constructor with no parameters, the entire process is unnoticed. The selection towards to the base classes can be automatically. When the entire call chain is identified, then the instantiation starts with the top constructor followed downward the chain elements to the bottom:

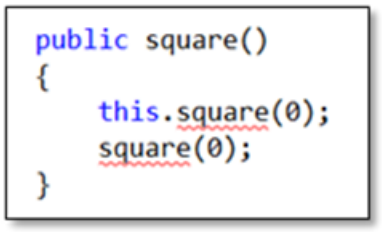


14.3. 14.3. To call an constructor with “this”

When a class has multiple constructors, there is a case when a constructor body can be substituted by another constructor in the very same class. Mainly occurs when the parameter list of the first constructor is a kind of simplification of the second one (some parameters are missing):

```
class square

{

   protected double a_side;

   public square()

   {

      this.a_side = 0;

   }


   public square(double a_side)

   {

      this.a_side = a_side;

   }
```

The first constructor 'new square()' can be substituted for the second constructor as 'new square(0)'. How could this connection be indicated in the code?



Although it seemed logical, since after all, a constructor is a kind of method, but to call another constructor seems should have a special syntax. There are several reasons for this, for example, the constructor does not have a return value (not even void), and the above syntax calls a method with void return type.

The following syntax rule is not a logical one, so do not try to understand, but try to remember: the constructor call has its own special syntax.

```
class square

{

  protected double a_side;

  public square() :this( 0 )

  {

  }


  public square(double a_side)

  {
```

```
    this.a side = a side;
  }
```

A constructor in the same class can be called by the keyword ‘this’. The 'this' must be followed by the function call operator, the two parentheses. Into the parentheses the actual parameters must be inserted. With the information of the actual parameters the 2nd constructor (to be called) can be identified. The call is a special place: it must be added after the constructor formal parameter list, but before the body starts:

```
class duck
{
   public duck(double weight, string name)
          :this(weight,name,true)
   {
     // ...
   }

   public duck(double weight, string name, bool is_alive)
   {
     // ...
   }
```

It is possible, even if the other (to be called) constructor is private (which cannot be called from any other classes directly):

```
enum sexes { ferfi, no }
class student
{
  public student(int age, string neptunID)
         :this(age,neptunID, sexes.no )
  {
  }

  private student(int age, string neptunID, sexes itsSex)
  {
  }
```

It is possible to continue this chain, that is, the 2nd constructor can call another 3rd constructor with ‘this' again:

```
class rectangle : square
{

   public rectangle():this(0)
   {
```

```
    }


    public rectangle(double a_side)
            :this(a_side,a_side)
    {
    }


    public rectangle(double a_side, double b_side)
    {
    }
```

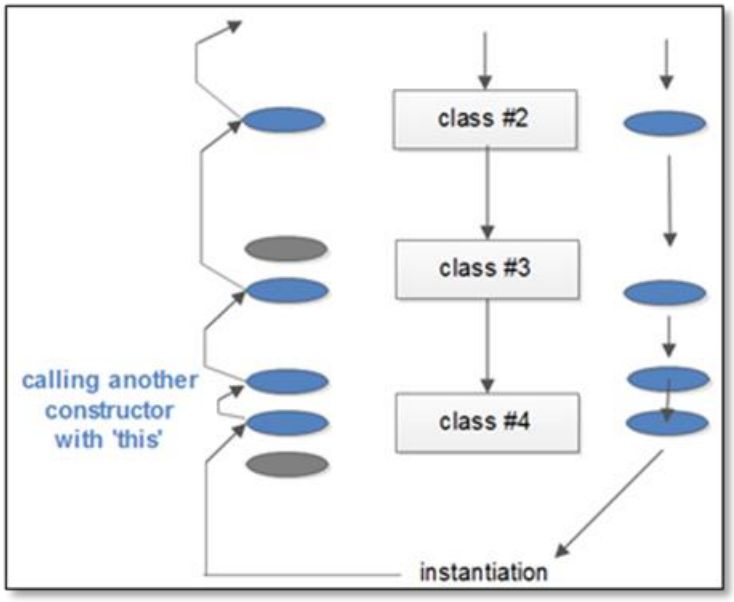
Of course, creating a circle of call is prohibited, so to call back a constructor in the chain which initiate a series of call to go to this constructor again. Unfortunately, the VS cannot detect this kind of syntax error (as each constructor separately is syntactically correct):

```
class rectangle : square
{


    public rectangle():this(0)
    {
    }


    public rectangle(double a side)
            :this(a_side,a_side)
    {
    }


    public rectangle(double a_side, double b_side)
            :this(a_side)
    {
    }
```

If we could create one (a circle of call), then the instantiation will take forever, as in the instantiation we call a constructor, which calls another one, and the instance will namer be finished. There is virtually no infinite recursion, since the computer's memory is finite. Therefore, in this case it will cause a runtime error.

14.4. 14.4. To call an own constructor and the constructor identification chain

When the compiler searches for the identification chain, it recognizes the ‘this’ constructor calls. When meets one, it inserts the (other) called constructor to the chain:



As it is an error to create a circle of calling, there must be a constructor which won't call another one. When the constructor call chain reaches this point, the search continues going upward one level in the inheritance tree.

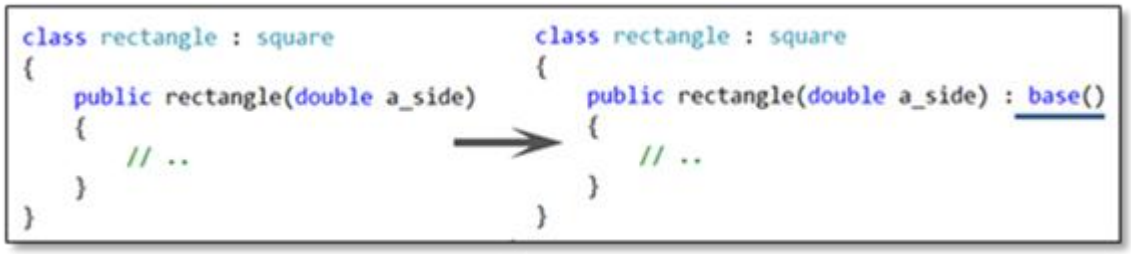
14.5. 14.5. To call an ancestor class constructor explicitly with 'base'

If there is an available constructor in the base class with no parameters, it is selected automatically. In other cases explicit (manual) selection must be initiated. The keyword 'this' cannot be used this time, as it calls another constructor in the very same class. Instead, the 'base' keyword must be used. Furthermore, other syntax rules of using the 'base' to call a constructor are the same as the rules of 'this':

```
class rectangle : square
{
    public rectangle(double a_side)
        :base(a_side)
    {
        // ..
    }
}

class trapezoid : rectangle
{
    public trapezoid(double a_side, double b_side)
        :base(a_side)
    {
        // ..
    }
}
```

When is it necessary to use the keyword 'base'? When there is a constructor with no parameters available in the base class, it is not necessary, as it is selected automatically. If, however, only parameterized constructors are available, the 'base' must be used! The compiler won't be able to pass parameters without our help!

In fact, behind a constructor, where we writes nothing, the 'base()' is included automatically, so the call of the base constructor is there implicitly:



According to this, when we do not write a constructor at all (eg, into the square class), during the compilation process it generates a constructor like this:

```
class teglalap : negyzet

{



   public teglalap():base()

   {

   }
```

If a class provides only constructors with parameters to the child classes, then in the child classes writing a constructor manually is required. To call a parameterised constructor in the base class must be indicates explicitly with the keyword ‘base’. The selection of the base class constructor is done by the actual parameter list.

So the possibilities are as follows:

· We do not write any constructor into our class. In this case a public, parameter less one is generated, which calls the parameter less constructor in the base class. It works, if such is available in the base class. If there is no ancestor class in our class selected explicitly, the Object class is selected automatically. This Object class has one public parameter less constructor, so in the direct child of this class writing a constructor is not required.

· We write a constructor of our own in our class. To this constructor:

a. we add nothing, which is equivalent with added the ':base()',

b. we add ‘:base()’ which is an explicit call to the ancestor class. In this case in the ancestor a constructor with the parameterization match must be present, with protected or public protection level.

c. we add ‘:this()’ which is a call of another constructor in our class. It might be private as well. Take care about not creating a circle of call, as it become a recursive call, which is not detected by the compiler as a syntax error, but will cause a runtime error.

14.6. 14.6. Class-level constructors

Not closely related to the subject (constructors and inheritance), but to close the topic it should be mentioned that there are also class level constructors.

The constructors are called instance level constructors because they are bound to creating constructors. Their role is to set the initial values for fields. The role of the level class constructor is very similar; assign the initial values to the class level (static) fields. It is no accident to use the singular – there cannot be such a (class level) constructor more than the maximum of one.

The name of the class level constructor must match the name of the class (as well), but there are three additional restrictions which are not presented in the instance level:

· must be marked with ‘static’ modifier,

- cannot have any parameters,
- cannot have any protection level added to it.

The present of the 'static' modifier indicates that it is a class level constructor. The lack of the parameters is because (as we will see) it is called automatically, and the system will not find out and pass parameters to it. Because it is prohibited to have any parameters, therefore, we cannot create more than one class level constructor (all of them would have the same name, and the same parameterization, which is not allowed under the rule ‘overloading’ either).

There are two questions to be clarified:

- When is it needed to write any class level constructor?
- When is it executed?

The class level constructor is responsible for setting the initial value to the class level fields primarily. In simple cases the initial values can be handled by initial assignments at the declaration of the class level fields, or often the initial values basing on the type of the fields assigned automatically are suitable. In most cases it is handled this way:

```
class Options

{

   public static int numOfClient = 20;

   public static string serverVersion = "0.9a";

   public static bool debugMode; // = false by default
```

The problem begins when the initial value of the field is not so simple, cannot describe with an expression. Before looking for such a case, let us make an experiment: create a class level field and a constructor in the 'Options' class, which prints a message:

```
class Options

{

   public static int counter = 0;

   static Options()

   {

      Console.WriteLine("Class level constructor of Options class");

   }
```

Prepare an almost useful Main program to test this constructor:

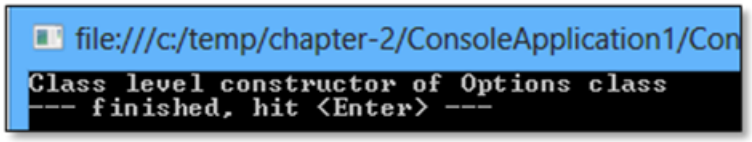
```
public static void Main()

{

   Console.WriteLine("--- finished, hit <Enter> ---");

   Console.ReadLine();
```

When we start the program, we can see that the class level constructor level may have not started at all, as we cannot see the message on the screen:

Modify the main program to increase the 'counter' field with 1:

```
public static void Main()

{

   Options.counter++;

   Console.WriteLine("--- finished, hit <Enter> ---");

   Console.ReadLine();
```

Starting the program again we will see the message:



So the class level constructor has been started, even though we cannot call it explicitly. We should not be able to, as there is no protection level, so the default level private will be used. In this case from the main program we are not able to access it, so not able to call it.

The class level constructor is called automatically, but we cannot give the exact point of time. The system guarantees that it will be executed before the first reference in the code to one of the class (to a class level field, method, instantiation, etc..).

The first attempt it was not executed, because we (in the main program) referenced nothing to the class at all. In the second attempt we referred to the counter field (increased its value), so the constructor is automatically executes before this statement.

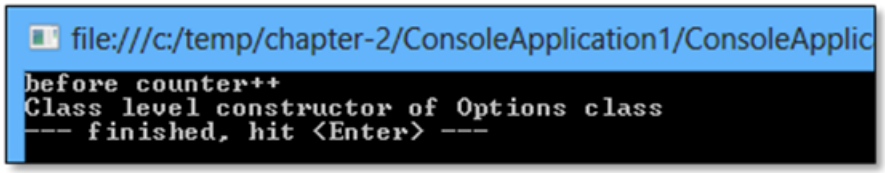
The system tries to optimize the execution of the class level constructors: to postpone the start to the latest point of time as possible (if it can be as it is not required, then be completely skipped).

It would be bad idea all the class level constructors would start at the very beginning as the program starts. Of course it could be. In this case, however, it would happen easily as the start of the program would be greatly delayed, it took a lot of time to reach our first statement written into the Main().

Let's have another try:

```
public static void Main()

{

  Console.WriteLine(" before counter++ ");

  Beallitasok.counter++;

  Console.WriteLine("--- finished, hit <Enter> ---");

  Console.ReadLine();
```

We can see that the programs starts with the statements of the Main(), and the execution of the class level constructor is delayed to a later moment:



So the "when" question was walked around carefully:

- do not know exactly when it starts,
- the only thing that is guaranteed: it starts before any reference is taken to the class,
- it could be that it never starts, as a reference to the class never happens.

The only question remaining: why should we write a class level constructor? Well, there are cases when the initial value of a field cannot be described as a simple expression. For example, because the value of a field must be read from a file or database table, or its value can be calculated from other data values described by an algorithm.

14.7. 14.7. Private constructors

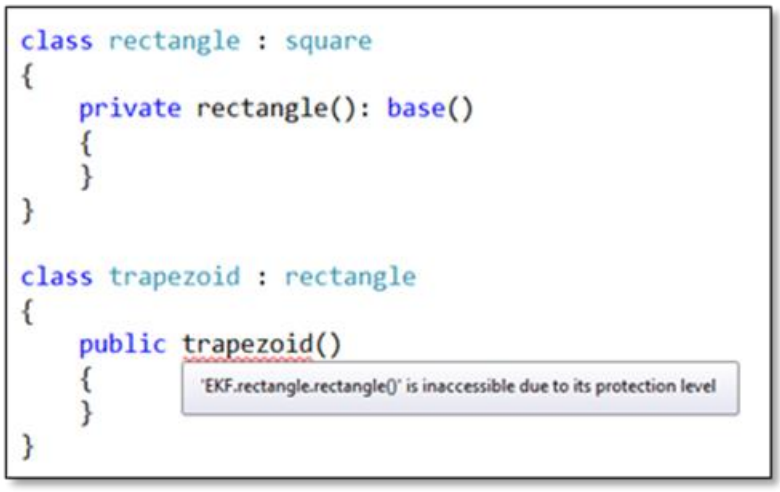
If we do not want to use our class as an ancestor class (do not want to let the creating of a child class), we have two options:

- creating private constructor,
- using the ‘sealed’ keyword.

If we do not write a constructor in our class at all, an empty constructor is generated which is public. Regarding to this a child class can be made, because without using the 'base' explicitly, a constructor in the child class can implicitly select this one. So if we do not provide a constructor at all, nothing will prevent other developers from making child classes from our class.

Obviously, it is the same when we creates constructors, but one of them is public or protected. In this case, the child class can also be created, because it can call this constructor in the chain.

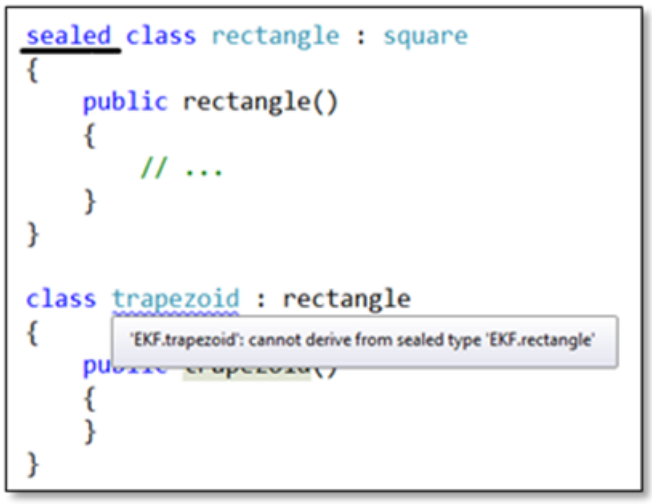
However, if we would create a constructor (constructors), but they are all private, then the child would be in trouble - cannot call them, not even with the help of 'base' explicitly. So a child class cannot be made!



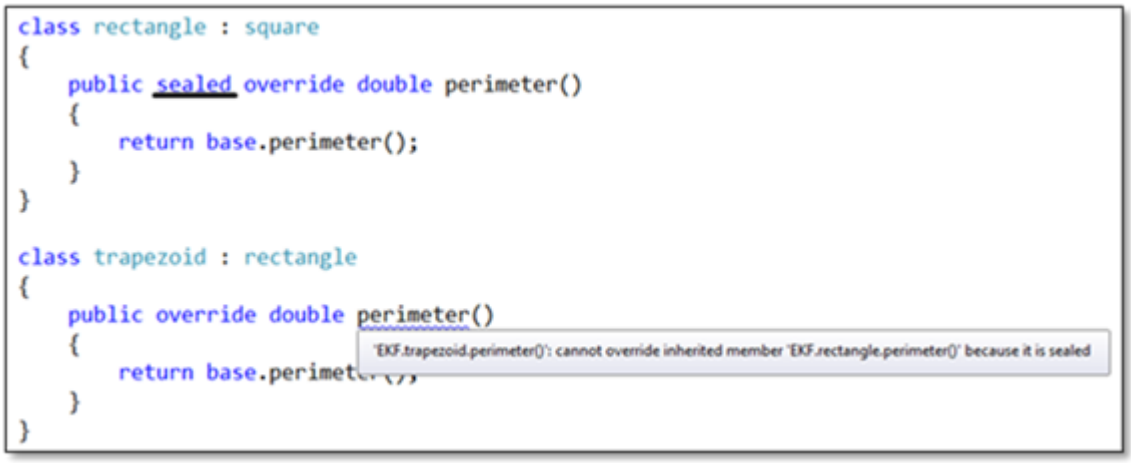
Note: when a class has only private constructors, a child class cannot be made. (But, if an instance can be created from this class is another question.)

14.8. 14.8. The keyword ‘sealed’

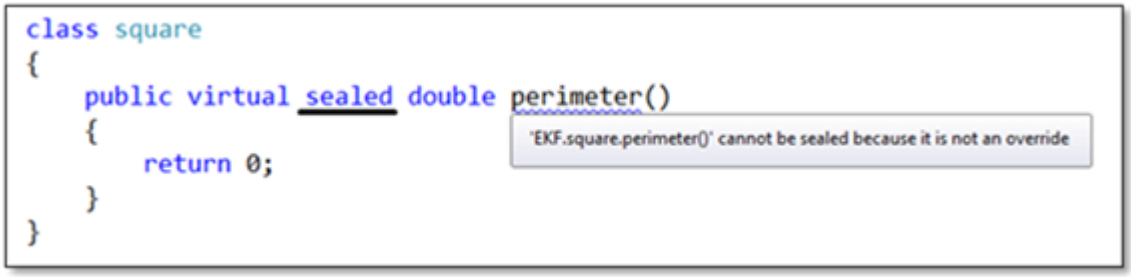
The second method, which can prevent child class to be created, is the use of 'sealed' flag. We can see the error message: "*cannot derive from sealed type rectangle"*.



The 'seal' can be applied not to only classes, but also methods and properties if they are late bounded. Then the child classes cannot override these methods (or properties). The error message indicates this problem in the example below "*cannot override inherited member perimeter() because it is sealed*":



Of course, not to mark the method 'sealed' when it is introduced for the first time (with the virtual keyword). It is meaningless in this case, mark with 'virtual' which enables the child classes to override it and order the compiler to use late binding it when it is called – then immediately disable the chance to be overridden:



14.9. 14.9. The Object Factory

Let us return to the case when our class has only private constructors. In this case, not only creation of a child class is impossible, but it will not be easy to instantiate. In fact, the place where we usually instantiate is outside of the class (e.g., in the function Main). Outside the class the private constructors cannot be accessed, so it cannot be called.

This is difficult, but does not make it completely impossible for instantiating. We must move the instantiation process into the class; the allocation of the memory with the 'new' and the call of the constructor as well. We must move them into a method which is not a problem at the first sight:

```
class rectangle

{

  private rectangle()

  {

  }


  public void createAnInstance()

  {

    // inside this a private constructor can be called

    rectangle n = new rectangle();

  }

 }
```

However, if this method is also instance level (without static mark), then we do not advanced step, because to call an instance level method an instance is needed. The solution is to have this method to be class level, make an instance (and return with the memory address of this instance):

```
class rectangle

{

  private rectangle()

  {

  }


  public static rectangle createAnInstance()

  {

    // inside this a private constructor can be called

    rectangle n = new rectangle();

    // the new instance can be returned

    return n;

  }

 }
```

Then the instantiation syntax is different, we do not use the usual 'new + constructor call' this time, but we can call the class level method:

```
static void Main(string[] args)

{

   rectangle t = rectangle.createAnInstance();
```

This class level method (whose task is no more than creating and returning new instances for the outside word) is called **Object Factory**.

The Object Factory method of course might have parameters, and might pass parameters to the constructor:

```
class rectangle

{

  protected double a_side;

  private rectangle(double a_side)

  {

    this.a_side = a_side;

  }


  public static rectangle createAnInstance(double d)

  {

    // inside this a private constructor can be called

    rectangle n = new rectangle( d );

    // the new instance can be returned

    return n;

  }
```

During the instantiation the parameters must be passed:

```
static void Main(string[] args)

{

   rectangle t = rectangle.createAnInstance(12.5);
```

There are many reasons why we create Object Factory methods, sometimes even when a constructor is available as well. In this example, we will show how to implement Object Factory for the case when only a single instance of a class can be made[16], and everyone must use the same instance. Be this class of a log file writing class. The program writes events to a log file. To write to the same file from all the points of the program (and the file can be opened only once), it can be implemented in the following way:

```
class fileLog

{

 public static fileLog instnce = null;

 static string[] dows = new string[] { "7-sunday",

           "1-monday", "2-tuesday", "3-wednesday", "4-thursday",

           "5-friday", "6-saturday" };

 // ..............................................................

 public static fileLog createAnInstance()

 {

     // if there is an instance we just return it
```

```
    if (instnce != null) return instnce;
    //
    var now = DateTime.Now;
    var basePath = Path.GetDirectoryName(
                      Assembly.GetExecutingAssembly().Location );
    var date = now.ToString("yyyy-MM-dd");
    var dow =
        (int)CultureInfo.InvariantCulture.Calendar.GetDayOfWeek(now);
    var dowstr = dows[dow];
    var fn = String.Format("{0}-{1}.log", date, dowstr);
    fn = Path.Combine(basePath, fn);
    instnce = new fileLog( fn );
    instnce.w.WriteLine(":: ---- {0} {1} LOG --- :: ", date, dowstr);
    instnce.w.WriteLine(":: log file opened {0} {1}",
                  now.ToString("yyyy-MM-dd HH:mm:ss.ff"),
                  dowstr.Substring(2));
    instnce.w.Flush();
    return instnce;
}
// ..............................................................
protected StreamWriter w = null;
public fileLog(string filenev)
{
    this.w = new StreamWriter(filenev,true, Encoding.UTF8);
}
// ..............................................................
public void writeln(string strEvent, params object[] parms)
{
    this.w.WriteLine(strEvent, parms);
}
```

15. 15. The indexer

When we plain and develop a class, we can often choose between writing a property or a method. The latter is usually chosen as it has a simpler and so has a better syntax, as a method call needs at least an empty pair of brackets (when it has no parameter like a getXXX() method), and instead of a setXXX(..) method an assignment statement is much more readable form.

Special case, when there in a field is which can store not one, but more than one value (the field can be a vector or a list). The fields are usually not worth sharing with the outside world (public) as these are reference type fields. When the outside world may access it, it can ruin its value (maybe by override it with ‘null’).

How can we share the values held by a field like that to the outside world? Let us examine an example about a fisher man, whose bag can store a given amount weight of fish. The fish is an object class; a field contains the weight of a fish. The bag is represented by a list, into which the fish are added. It must be guaranteed that no more fish can be added by the outside world than the weight capacity of the bag.

```
class fish
{
 protected double _weight;
 public double weight
 {
  get { return _weight; }
  set
  {
   if (value < 0 || value > 50.0)
            throw new ArgumentException("not acceptable fish weight");
   else _weight = value;
  }
 }

 public fish(double weight)
 {
   this.weight = weight;
 }
}
```

After ‘fish’ class let us see the fisherman class:

```
class fisherman
{
    protected List<fish> bag = new List<fish>();
    protected double bag_weight = 0;
    //
    public void bag_add(fish h)
    {
        if (h == null)
            throw new NullReferenceException("a fish cannot be null ");
        if (h.weight + bag_weight > 20.0)
```

```
            throw new ArgumentException("a fish is too big ");

        bag.Add(h);

        bag_weight = bag_weight + h.weight;

    }

    //

    public fish bag element(int index)

    {

        if (index < 0 || index >= bag.Count)

            throw new IndexOutOfRangeException();

        return bag[index];

    }

}
```

The ‘bag_add()' function is used to add a fish to the bag. In our case, the maximum load of the bag is 20 kg. The ‘bag_weight’ field keeps the actual weight of the fishes in the bag. When a fish is added to the bag, we increase the value of this field. The 'bag_element()' function allows us to retrieve a given (identified with its index) fish.

It can be used as the following:

```
fisherman john = new fisherman();

fish f1 = new fish(2.5);

john.bag_add( f1 );

fish f2 = new fish(4.2);

john.bag add(f2);

fish h = john.bag_element(0);
```

This code is not elegant, not pretty. Below, you'll learn how to write an indexer, and we can write the same program with a prettier syntax.

The Indexer is a property which a parameter might have. The syntax of an indexer must be different to the methods, so the formal parameters must not be closed into round brackets but square brackets instead. In addition, the name of an indexer is given, it must be the keyword 'this'. An example:

```
class fisherman

{

    protected List<fish> bag = new List<fish>();

    protected double bag_weight = 0;

    //

    public fish this[int index]

    {

        get

        {

            if (index < 0 || index >= bag.Count)                throw new
IndexOutOfRangeException();
```

```
            return bag[index];
        }
    }
```

In this case, the read a fish from the bag can be initiated by reading its index. One must write the name of the indexer ('this') as it is added automatically by the compiler:

```
fish f = john[0];
// in real it is
// f = john.this[0]
```

If we want to replace an already added fish in the bag into another one, we might write the following (without an indexer, the fish to replace is identified with its index):

```
public void bag_replace(int index, fish f)
{
  if (f == null)
     throw new NullReferenceException("a fish cannot be null");
  if (index < 0 || index >= bag.Count)      throw new IndexOutOfRangeException();
  fish repl = bag[index];
  if (f.weight-repl.weight + bag_weight > 20.0)
     throw new ArgumentException("this fish too big to bag ");
  bag[index] = h;
  bag_weight = bag_weight - repl.weight + f.weight;
}
```

The example to use:

```
fish f3 = new fish(3.5);
john.bag_replace(0, f3);
```

With the help of the indexer syntax we can write it:

```
public fish this[int index]
{
  set
  {
     if (value == null)
         throw new NullReferenceException("a fish cannot be null");
     if (index < 0 || index >= bag.Count)
         throw new IndexOutOfRangeException();
     fish repl = bag[index];
     if (value.weight - repl.weight + bag_weight > 20.0)
         throw new ArgumentException("this fish too big to bag");
```

```
      bag[index] = value;

      bag_weight = bag_weight - repl.weight + value.weight;

   }
```

Use:

```
fish f3 = new fish(3.5);

john[0] = f3;
```

Or simpler (but it is not due to the indexer):

```
john[0] = new fish(3.5);
```

So in our case the structure of an indexer is the following:

```
public fish this[int index]

{

   set { /* ... body ... */ }

   get { /* ... body ... */ }

}
```

This indexer has an 'int' type parameter, which identifies a fish in the bag. Suppose we want to store another kind of values (for example dates of when we went fishing). As we often go to fish, a list of DateTime elements are stored in a field:

```
public DateTime this[int index]

{

   set { /* ... body ... */ }

   get { /* ... body ... */ }

}
```

It would be difficult to use, since in both cases the number of list elements throughout their index is referenced (reading and writing as well):

```
fish h     = john[0];

DateTime m = john[0];

//

john[0] = new fish(3.5);

john[0] = DateTime.Now;
```

The rule is: a class might have more than one indexer, but their parameterization must be different. So when the types of the indexers (DateTime, and fish) are different it is not enough, the critical is the parameterization:

In addition, it is possible for an indexer to have more than one parameter:

```
public DatTime this[int a, int b]

{

    set { /* ... tartalom ... */ }

    get { /* ... tartalom ... */ }

}
```

The use of this indexer (supposing ‘p’ is an instance):

```
p[0, 1] = DateTime.Now;

DateTime n = p[2, 3];
```

The vectors have an indexer:

```
double[] t = new double[20];

t[2] = 32.4;

double x = t[1];
```

The lists have an indexer:

```
List<bool> l = new List<bool>();

// ... lista feltöltése

l[2] = true;

bool x = l[1];
```

There is also a special "list" of what we call *Dictionary*. While a traditional list (which for example stores student instances) is indexed (elements are identified by) with integer numbers, in a dictionary we can choose what type of identifier we want to use, it may be a string. In this case, it can be imagined as the student’s neptun id (the main point is to be a unique identifier). To have a dictionary instance we must give the type of the identifier, and the type of the items that are stored in the dictionary:

```
Dictionary<string, student> l = new Dictionary<string, student>();

//

l["BZQQC3"] = new student("smith");

l["N67ERO"] = new student("borg");

//

l.Add("SEB8IR", new diak("anna") );

//

diak d = l["BZQQC3"];
```

New elements can be added in one of two ways to an initially empty Dictionary:

- with the Add() method, which receives the index of the element (the key), and the element (value) itself,
- with the help of the indexer, simply assign a new value to a key, so store it into the list.

The Add() will throw an error (exception) if an item already exists with the given key, because the id must be unique. With the indexer it is safer, because it is an item with the given key already exists, it replaces (overrides) the old value with the new one. If the key does not exists, then it will be added to the list.

16. 16. Namespaces

The important aspect of the OOP world is "grouping". The use of encapsulation principle compels us to gather together the fields and operations necessary for the programming task. To cover a topic several classes can be made and a developer can select the best for him. So child classes are created to specialize the main classes towards some important aspect.

If we work hard, a handful of classes and enums will be possessed. How can we group them together?

The namespaces are the best to do that. Namespaces allow us to group not fields and classes, but in a higher level we can create groups of classes and enums and other type definitions:

```
namespace organisms

{

    class animals { /* ... */ }

    class pets : animals { /* ... */ }

    class dogs : animals { /* ... */ }

    class cats : animals { /* ... */ }

    class siamese: cats { /* ... */ }

    /* and so on */

}
```

Each namespace has a name, which is followed by the 'namespace' keyword. This is an identifier, so the naming rules of the identifier must be applied (letter, digit, or underscore, do not start with a digit, etc). The namespace name will complement the class name, in this example 'cats' full name becomes ‘organisms.cats’. This syntax can be used, as well, in instantiation:

```
static void Main(string[] args)

{

   organisms.cats m = new organisms.cats();
```

This form, when you specify the name of the class and the container namespace is the **fully qualified name**. According to the rule, it must be completely unique, so in the same namespace each class must have different name.

As we don’t want to use the fully qualified name all the time, we might use the keyword ‘using’. Generally we place it at the beginning of the source code. We must add a name of an existing namespace after the keyword, and this means the in the case of classes and other type names are put into this namespace we don’t need to write the fully qualified name but the class name only:

```
using organisms;

class Program

{

  static void Main(string[] args)

  {

     cats m = new cats();
```

This keyword has no other meanings. So if you delete this line from the source code, the only thing that is changed is when we want to refer to a class inside this namespace we must use the fully qualified name. This is the case of the Console class inside the System namespace as well:

```
// using System; <- deleted

static void Main(string[] args)

{

  System.Console.Write("Enter a number between 1..10:");

  int a = int.Parse( System.Console.ReadLine() );
```

Therefore we warn the Pascal/Delphi developers that this C# keyword seems tobe very similar to keyword ‘uses’ in the Pascal language. When we forget to write a ‘uses <unitname>’ in Pascal, we are not able to call any function or procedure from that unit. The 'uses' is actually a message to the linker. If a unit name is not specified in the uses list, the linker won’t link the given unit to the executable program, and thus the functions are not included into the final program. The keyword ‘using’ in C# has no strong meanings. Without the ‘using’ the classes are still available, just in an uncomfortable syntax. In fact, the same as the Pascal ‘uses’ is the same the *Project Managementð add reference* menu point (will be discussed later at the assembly chapter).

It is important to know that after a ‘uses’ only a namespace can be specified, a name of a class cannot. Otherwise, it would be nice to call the methods of Console class such as:

```
using System.Console; // <- in fact this is an error


static void Main(string[] args)
{
  // call the Console.Write as
  Write("Enter a number between 1..10:");
```

Next you should know that namespaces is a hierarchical system, in other words: namespaces can be nested. In this case a name of the internal namespace is added to the container namespace name:

```
namespace organisms
{
    namespace pets
    {
        class animal { /* ... */ }
        class dog : animal { /* ... */ }
        class cat : animal { /* ... */ }
    }
    namespace wild
    {
        class animal { /* ... */ }
        class lion : animal { /* ... */ }
    }
}
```

In this case the fully qualified name of the ‘dog’ is:

```
organisms.pets.dog
```

While the name of the ‘lion’ class is:

```
organisms.wild.lion
```

The namespaces can be nested to any depth, so there are no obstacles of the following class name:

```
System.Runtime.Serialization.Formatters.Soap.SoapFormatter
```

We can make a complex namespace in the following way as well:

```
namespace organisms.pets

{

    class animal { /* ... */ }

    class kutya : animal { /* ... */ }

    class cica : animal { /* ... */ }

}

namespace organisms.wild

{

    class animal { /* ... */ }

    class lion : animal { /* ... */ }

}
```

Namespaces help grouping classes. If we know the task to fulfil, we might easily find the namespace which will contain a class to help. Classes deal with the same particular problem typically included into one namespace is. The Microsoft. NET Framework is designed consistently that the classes of the Base Class Library (BCL) are in the System namespaces, or in sub namespace of it:

- **System.Collections**: complex data structures (arrays, lists, stack, array, ...)
- **System.Collections.Generic**: general types of complex data structures (List <int> ...)
- **System.Drawing**: to draw to a canvas (brushes, pens, picture formats, canvas, ...)
- **System.IO**: directories, files management, and so on,
- **System.Reflection**: run-time type information management, attributes, DLL (assembly) management, …
- **System.Runtime.Remoting**: remote method call , and so on,
- **System.Threading**: Writing multithreaded programs, critical sections management, locks, and so on,
- **System.Web**: web-based communication management.

In the VS2010, when you create a new console application, the following namespaces are added automatically:

```
using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Runtime.InteropServices;

using System.Reflection;

using System.IO;
```

Most of them are not needed, so might as well get deleted. It is worth to keep the System (mainly due to the console), and the next one with the Generic ending (because of the List types). Perhaps the IO namespace, when we want to manage the file system or files themselves (text file read or write). In most cases the removing of the other usings won't cause any problems.

```
using System;

using System.Collections.Generic;
```

```
using System.IO;
```

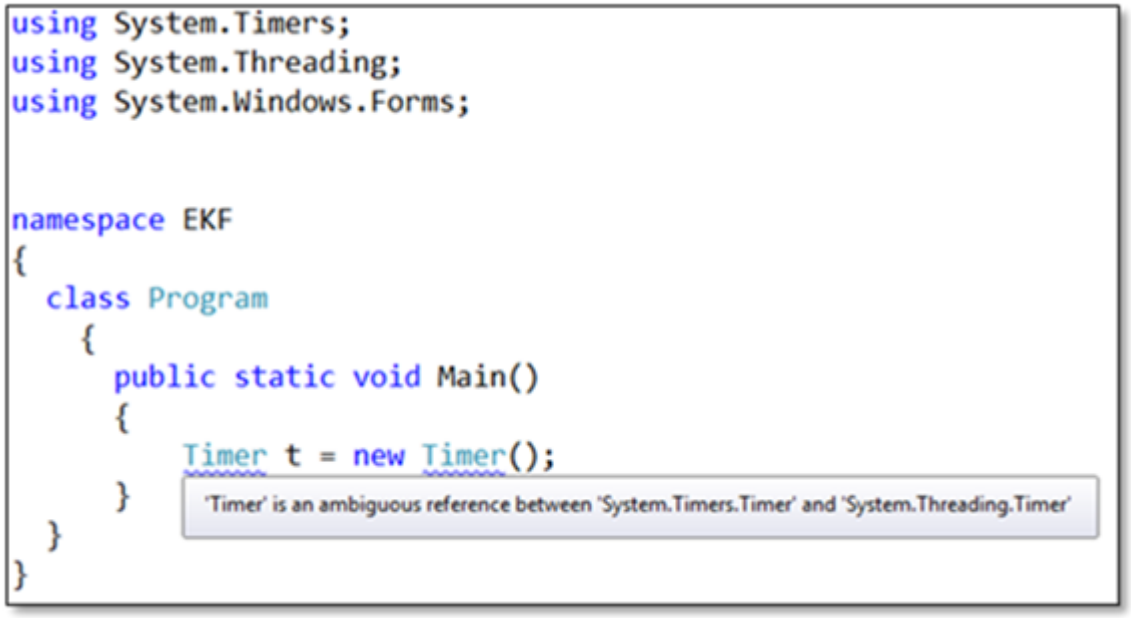
Does it worth to deal with these issues at all, remove the unnecessary ones? How does it work actually ‘using’?

When we refer to a class in the source code, but do not specify the namespace name, the compiler must figure it out. How does it think? It checks all the namespaces for the given class name (based on the ‘using’ list). If none of them does it exist, the class is unknown, so a syntax error emerges. If it find the exact same class name in a namespace, then it is good, the class is identified. If the class name can be found in more than one namespace (opened by a ‘using’), it is also wrong as the compiler won’t choose the one we thought.

Such a case may arise easily. There is a class name, 'Timer', and classes with this name can be found in different namespaces:

- System.Timers.Timer
- System.Threading.Timer
- System.Windows.Forms.Timer

If - for example- all these namespaces are opened[17] with ‘using’, a reference for the Timer class would be a problem:



As we can see in the picture, the VS displays an error. The reason is that it is not clear which Timer class it is trying to refer to. This error may occur when we adds a lot of ‘using’ at the beginning of the program. What should we do? In a case like this we must use the fully qualified name to make clear which Timer class we think of:

```
using System.Timers;

using System.Threading;

using System.Windows.Forms;


namespace EKF

{

   class Program

   {
```

```
        static void Main(string[] args)

        {

            System.Timers.Timer t = new System.Timers.Timer();

        }

   }

}
```

This way, we can handle the situation if we have a lot of open namespaces containing classes with the same class name.

However, considered as a disadvantage of several ‘using’ items: it can slow down the compilation process. The compiler must check a lot of namespaces in order to identify the fully qualified names of the classes. Obviously, the more namespaces it looks at, the process becomes slower to run. (But really, in a modern computer, the difference can be measured as not 1.2 seconds but instead of 1.3 seconds is needed to finish the compilation. We might think we do not really need to deal with this issue.)

Items can be added to a namespace at any time. A namespace cannot be closed, anytime it can be continued, in the same source code or in another one. We can add new classes to the System namespace as well, although Microsoft was asked not to do so because it preserves the option to add classes later, and then it might collide with the our ones.

```
namespace organisms.pets

{

    class animals { /* ... */ }

}



/* .... */

// we close the namespace //

/* .... */



// but we reopen it to add new classes

namespace organisms.pets

{

    class dogs : animals { /* ... */ }

    class cats : animals { /* ... */ }

}
```

It is an interesting fact in connection with namespaces, that this keyword can be used in a different syntax to create alias name for an existing namespace:

```
using gen = System.Collections.Generic;
```

Into this new alias 'gen' a longer namespace name is assigned. Later in the code when we must use a fully qualified name, we can use this alias name instead of the namespace name:

```
// System.Collection.Generic.List<double szamok = new ...

gen.List<double> szamok = new gen.List<double>();
```

For this we will need this short form, because this kind of ‘using’ won’t open the namespace, so to refer a class in this namespace we must use the fully qualified name (or the shorter form with the alias).

If a class is not inserted into any namespace, but is inserted into outside of all the namespaces, then we say that this class is in the **global namespace**.

```
using System;

using System.Collections.Generic;



class animal

{

  /* ... */

}
```

This is possible, but should be avoided. In fact, there is no particular reason why we do not insert this class into any namespace. A namespace name can be – for example - our own name, our monogram, the name of the company (or school), whatever. If we do not use a namespace, then later if a class name collision happens, there will be no chance of unlocking this situation in the code with the fully qualified name.

17. 17. The Object class as the ancestor

If we begin the development of an object class, and it does not indicate the ancestor - it automatically becomes the Object class. The Object class is inserted in the System namespace, so its fully qualified name is System.Object, its short alias is "object". In other words, all of the following is the same:

```
using System;

class myOwn:Object

{

class myOwn:System.Object

{

class myOwn:object

{

class myOwn

{
```

The Object class contains some basic methods, which therefore is inherited by all other classes:

· getType() method, which specifies the type of the class name (the class name and the namespace in which the class was inserted),

· toString() method, which creates a string representing the class instance),

· Equals(x) method, which determines whether the instance ‘x’ is equal to the actual instance,

· GetHashCode() gives you a numeric value (int) representing the actual instance.

17.1. 17.1. GetType()

Let's look at an example. The special function *getType* is usable in debugging, logging. Create a function that prints the name of the type parameter is received:

```
static void itsType( Object p )

{

  Console.WriteLine(p.GetType());

}
```

Let's try this function on some simple types:

```
int a=2;

string b="hello";

List<int> l = new List<int>();

itsType(a);

itsType(b);

itsType(l);
```

And the writings are:

Other examples of how to use the GetType() method will be seen at chapter ”29.3 Finding a class inside an assembly”.

17.2. 17.2. ToString()

The toString() basically for the purpose to convert the instance to string. It is very useful when we write it to the screen. The Console.Write() and WriteLine() functions uses this method for all parameters passed to them – they call the toString() method, and the resulted string is printed to the screen. The numbers are converted to its decimal system form (digits). The logical type values are converted to ‘true’ or ‘false’ words.

```
int a = 12;

Console.WriteLine(a);
```

What it really is:

```
Console.WriteLine( a.ToString() );
```

Same happens when it is inserted into the format string: it means to call the ToString() methods as well. The expressions are calculated, and to the values the type specified ToString() is called:

```
int a = 12;

int b = 20;

Console.WriteLine("{0}+{1}={2}",a,b,a+b));
```

It is really the same as:

```
Console.WriteLine("{0}+{1}={2}", a.ToString(), b.ToString(),

                                 (a + b).ToString());
```

In our own class the toString() method can be overridden, because the toString() is a virtual method:

```
class duck
```

```
{
  public string name;
  public duck(string name)
  {
     this.name = name;
  }

  public override string ToString()
  {
     return String.Format("{0} duck", name);
  }
}
static void Main(string[] args)
{
   duck d = new duck("donald");
   Console.WriteLine("the [{0}] goes to swim", d);
```

In this case when we write 'd' to the screen, in fact we print 'd.ToString()':

17.3. 17.3. Equals()

There are two ways to test the equality in the program. First, with the equal operator (==), on the other hand, with the Equals() method. In most cases, they operate in the same way as the equality operator is often led back to the Equals() method:

```
if (a == b) Console.Write("they are equals");
else Console.Write("they are not equals");

if (a.Equals(b)) Console.Write("they are equals");
else Console.Write("they are not equals");
```

The Equals() method is virtual and can be overridden for individual classes. For the reference types the == operator to test that the two memory address are the same[18], meanwhile the Equals() function is can base on other relations:

```
duck d = new duck("donald");
duck p = new duck("donald");
if (d == p) Console.Write("they are the same");
else Console.Write("different ducks");
```

```
if (d.Equals(p)) Console.Write("they are the same");

else Console.Write("different ducks");
```

In both cases, the ducks will be "different", because the Equals () is the same as operator '==' by default. However, when we override in a class (eg, we want two ducks to be the same, when they have the same name):

```
class duck

{

  public string name;

  // ....


  public override bool Equals(object obj)

  {

    if (this == obj) return true;


    if (obj is duck && (obj as duck).name == this.name)

                return true;

    else return false;

  }

}
```

Then the check 'p == d' shows the two ducks are different, but by the check 'd.Equals(p)' we will find they are the same. Thus, in our program we have the opportunity to define an alternative test of equality to our classes.

17.4. 17.4. GetHashCode()

The GetHashCode() generates a hash value for a given instance. The hash value is a thumbprint of the instance, produced from the key properties of the instance. The production must be fast, as it is used in a speed-oriented checking of equality. For example, if we have two copies of data, the test of the equality we can compare the hash value of two instances. If they are not equal, then the two copies are certainly not the same. If the two hash values are equal, *it is possible* that the two copies are equal. Since the hash values can also be equal if the two copies are not identical; therefore in a large amount of instances the hash value is suitable to find the few instances which are possible the be equal very fast, then in the next step we can check the remained ones in a slower but exact test of equality.

In our example we can produce a hash value by multiplying the age of duck (in month) by its weight. Because of the weight is a double value (in kilograms), so we convert it into decagram and so it is rounded to a whole number:

```
class duck

{

  protected double weight;

  protected int ageInMonths;


  public override int GetHashCode()

  {
```

```
    return (int)(weight * 100 * ageInMonths);

  }
```

It is possible that for different duck instances the hash value are the same. But in addition it is also possible that if we want to find a 20-month 3.5 kg duck in a large list with about 10,000-elements, based on this comparison only few instances match, and in the time-consuming comparison only few instances shall be checked in detail. Thus, we can achieve much faster searching.

17.5. 17.5. Boxing – Unboxing

The Object is the ancestor of all classes created by the 'class' keyword. If it is not direct ancestor, then it is an indirect ancestor. These types are reference types, which means an instance variable costs 4 bytes as it stores a memory address at first level. The secondary memory area is allocated by a ‘new’ keyword during the instantiation. Let us examine the following assignments:

```
duck d = new duck("donald");

object o = d;
```

The instance 'o' needs 4 bytes of memory as well as the variable 'd'. The 'o = d' means the copying of the 4 byte memory address from ‘d’ to ‘o’. In fact, between any instance variables this copy could be executed physically, since they are 4 bytes, and stores memory addresses:

```
duck d = new duck("donald");

F15Tomcat x = d;
```

Obviously, to copy a memory address of a duck to an F15Tomcat instance variable is physically possible, but it is very unlikely the method call 'x.flyHigh()' would work as expected. The compiler will check these assignments whether they match the type compatibility rules.

We know that any of the classes the Object is an ancestor, however the Object is compatible with all the types. But what about the value types (bool, double, int, etc.)?

The value types are not created by the 'class' but with the use of the 'struct' keyword. For an instance of a 'struct' (record) usually costs more than 4 byte in memory, as they stores not a memory address but the values directly.

When we create a struct type, we cannot indicate any ancestor, as the OOP rule inheritance is not extended to this keyword. The default ancestor in this case is System.ValueType class, whose ancestor is in the Object. For this reason, struct types have the methods mentioned above (toString, getType etc). From the class ValueType a child class cannot be created (this is disabled by the compiler), but with the help of the struct keyword it can.

Because all the value types are child classes of the Object class, the following assignment statement is correct:

```
double d = 13.4;

object o = d;

Console.WriteLine("o = {0}", o);
```

In the assignment 'o = d' the type of the right-hand side expression (whatever it is) is compatible with the type of the left-hand side (object). For it is the double typed variable 'd' can appear on the right side of the assignment. Try to imagine what happens during this assignment, since the memory requirement of ‘d’ is 8 bytes, the variable ‘o’ has only 4 bytes in memory[19]. It is clear that it is not possible to copy the 8 byte number to a 4 byte area. In addition the mantissa and the characteristic of a double cannot be interpreted as a memory address. Against these, the code above is correct, it works, and at the end you see the "o = 13.4" writing on the screen. The main reason is that during the WriteLine the overrided version of the 'o.ToString ()' is called. As the dynamic type of ‘o’ is double, it is the ‘ToString()’ defined in the double struct, which will generates this string ’13.4’.

Now let us go back to assignment itself. There are two interpretation is possible. The first is the more natural idea: variable 'o' stores the memory address of the variable 'd'. If this is so, then what should the following code write to the screen?

```
double d = 13.2;

object o = d;

d = 16.5;

Console.WriteLine("o = {0}", o);
```

If 'o' would store the memory address of 'd', and after the storing the actual value of 'd' would have changed, the writings would be the last (actual) value of 'd' which is 16.5:

It appears it is not what has happened here, since original value appeared. Find a new theory!

The statement when we assign a value type value to a reference type (this can be only the Object type) variable is called **boxing**. During the boxing we make a copy if the value type value in the memory and its memory address is stored in the reference type variable. In other words, 'o = d' means copying the double value of 'd' (13.2, which costs another 8 bytes), and finally we stores its memory address to ‘o’. Then the value of 'd' could be changed, it will not affect the value of ‘o’ anyway.

The Write will call the 'o.ToString()', but it is possible to extract the original double value from 'o', however it is not so easy:

```
double d = 13.2;

object o = d;

// ...

double x = o;
```

The latest assignment ‘x = o’ is not type correct. Consider: the 'o' can store almost any type of value, why the compiler should think that it is a double? Since the type compatibility is a one way thing, and the left-handed type is double, which is not compatible with the type of Object, so the assignment is syntactically incorrect. That is not a big problem, just we must apply a type cast. The 'as' operator is not an option this time, because it cannot be used for value types. Only the traditional method remains:

```
double d = 13.2;

object o = d;

// ...

double x = (double)o;
```

This action, when he recover a value type value from an Object type variable is called the **unboxing**, the opposite of boxing. In doing so, the value is copying back from the memory address stored in 'o' to the double variable 'x'.

17.6. 17.6. The list of Object

Let us examine the List<Object> construction from this point of view. If we create such a list we can add any type of values to it, as any type is compatible with the Object type (which is the base type of this list). We can add strings, ducks, doubles to the same list. When we add a reference type value, we store the memory address into the list. When we add a value type value, a boxing statement is executed, and the memory address of the copied value is added.

However, the List<double> stores the double values themselves. Adding a new double value also means copying the value, but at least directly to the list, not into somewhere to the memory, and stores the extra and unnecessary 4 byte memory address. That is we can add a new value without the boxing steps.

```
List<Object> lo = new List<object>();

List<double> ld = new List<double>();

double d = 13.4;



lo.Add( d ); // boxing, slow, it costs 12 bytes

ld.Add( d ); // no boxing, fast, 8 bytes only
```

When we recover an item from the list, there are other problems that arise with the Object typed list arise. Similarly to the above, type casting and unboxing is needed:

```
double x = (double)lo[0]; // needs typecast, unboxing, slow

double z = ld[0]; // no need typecast, no unboxing, fast
```

The System namespace contains an ArrayList class, which is essentially the same as the List<Object>. In the earlier version of C # language and the. NET Framework the List<double> and the concept of list with a given type simply did not exist, so it was the only possibility to create a list. In the more modern version of C# the ArrayList is not useful anymore.

There are two comments to the list we must discuss. If a universal type (eq. Object) is the base type of the list, make sure that always the type cast is to use to recover the values from the list, never the converting method. Compare the following three ways:

```
double h = (double)lo[0];

double k = Convert.ToDouble(lo[0]);

double m = double.Parse(lo[0].ToString());
```

If we have an Object type list, and we know that the 0th element is a double, then use the type cast method presented at the variable 'h'. Whereas an unboxing happens, but it is still the fastest way. The case of 'k', an Object is given to 'ToDouble()', which immediately calls the 'toString()', then the resulting string is converted back to double. The same happens at 'm', but it seems better in this case, ad 'Parse()' won't accept an Object as a parameter, so we must call 'toString()' manually before.

We present an approximate speed measurement method. A steps above is inserted into an iteration with a big amount of repetition (about 100000000 steps), and measure the time. To make sense to read the 0th element of the list, we calculate the sum of the numbers. Although this means a little extra time, but because it is in all three cases, it will increases the total running time withthe same amount of time):

```
DateTime start = DateTime.Now;

double sum = 0;

for (int i = 0; i < 100000; i++)

{

  double h = (double)lo[0];

  sum = sum + h;

}

DateTime stop = DateTime.Now;

Console.WriteLine("total time: {0}",stop - start);
```

```
Console.ReadLine();
```

According to the measurements[20]:

'h' method	2,28 sec	100%
'k' method	3,77 sec	165%
'm' method	71,77 sec	3147%
List<double>	1,67 sec	60%

The results show that the Convert.ToDouble is fast, probably uses the 'is' operator to examine whether it receives a double, and converts the same way as in 'h'. As an extra 'is' operator is used, and an extra function call, an extra parameter passing, it causes the increased execution time. The third method appears to be the slowest solution, converting to string and back seems extremely time consuming. However, the List<double> is the fastest method.

When we want to summarize only the double elements from the list, we might use a 'foreach' iteration, but not in the following way:

```
List<Object> lo = new List<object>();

// ... fill the list with elements ...

//

double ossz = 0;

foreach (double d in lo)

{

  ossz = ossz + d;

}
```

The foreach processes all the elements from the list. When it reaches a non-double element in the list, then we will get a runtime error when the foreach tries to assign that value into the double 'd' variable. Of course, if you we are sure that all items are double in the list, like in the example above, we can write this code. Now the question may arise: why is List<double> not what it was originally? Run another measurement:

```
List<Object> lo = new List<object>();

for (int i = 0; i < 1000; i++) lo.Add(1.2);

//

DateTime start = DateTime.Now;

double sum = 0;

for (int i = 0; i < 1000000; i++)

{

  foreach (double d in lo)

    sum = sum + d;

}

DateTime stop = DateTime.Now;

Console.WriteLine("total time: {0}", stop - start);
```

```
Console.ReadLine();
```

The measurement, however, produces an interesting result. The total time in the case of the List<object> was 14.48 seconds, while the List<double> unexpectedly took 16.69 seconds during 1000000 iterations. At the same time, remember that the list of all the items in a list <object> plus 4 bytes of storage required compared to using the List <double> list! But we do not forget that in a case of List<Object> we need an extra 4 bytes for all items.

If not all the items in the list is double, the 'is' operator is useful to select the required items from the list. In this example, we show a way to count the number of 'duck' items in the 'lo' list:

```
List<Object> lo = new List<Object>();

// ... fill the list with elements ...

//

int counter = 0;

// foreach(duck k in lo) cannot be used !!!

foreach (object o in lo)

{

  if (o is duck)

       counter++;

}
```

If we want to call a method with the duck instances, we need not forget that the type of the loop variable ‘o’ is an Object, so after the check of whether it is duck or not, we need a type cast (as the point operator bases on the static type not on the dynamic type):

```
foreach (object o in lo)

{

  if (o is duck)

        (o as duck).flyHigh();

}
```

Of course, this is true even if the list type is not Object, but any "ancestor" class type (in this example, an ancestor of the duck class).

17.7. 17.7. An object parameter

It could happen that we develop a method which receives one or more parameters, which can be "anything". Such as the Console class Write and WriteLine class level methods. Try to develop a similar method:

```
static void writeIt(Object p)

{

  // ...

}
```

Then, of course, we can pass to it a string, a double, a duck, anything:

```
writeIt( "some text" );

writeIt( 12.5 );

duck d = new duck( "donald" );
```

```
writeIt( d );
```

As our parameter type is Object, we can do these, but when we pass a double value, a boxing will be executed in the background. We can avoided it when we develop a special version of this method:

```
static void writeIt(Double p)

{

  // ...

}
```

If we look at the Console.Write and WriteLine functions we can detect that there are special parameterized versions of the most common value type parameter.

Think about how we can pass any number of any types of parameter. In this case it also means that the types of parameters may be different from each other (for example, a 1 string, a double, 2 ducks, 1 int, then a duck again, etc). When we do not know the number of parameters to receive, we can specify the keyword 'params' in the formal parameter list:

```
static void writeIt(params Object[] t)

{

  // ...

}
```

Then inside the function, the 't' is actually a vector of type Object, the length of the vector depends on the number of parameters passed:

```
kacsa d = new kacsa("donald");

writeIt("some text", 12.5, true, d);
```

The ‘params’ keyword means that the call will not pass a vector from Object type, but the elements of the vector, an the ‘t’ vector must be created only at the moment of calling from the given elements. In the example above the vector inside the function will be:

t.Length ð 4

t[0] ð string “some text”

t[1] ð double 12.5

t[2] ð bool true

t[3] ð duck d

If we want to create a formatted string from the elements of 't' vector, there are several options. The first is the use of StringBuilder. This is essentially a list of string to which items can be added. Its great advantage is that, when we finish adding the elements, then it is easy to create a single string by concatenating elements:

```
static void writeIt(params Object[] t)

{

  StringBuilder b = new StringBuilder();

  foreach (object x in t)

     b.Append(x);

  string reslt = b.ToString();
```

```
  // ...
}
```

The StringBuilder instance ‘b’ with the ‘Append()’ method is used to add items. It is similar to the lists ‘Add()’ method. Finally it can be converted to a string by the ‘toString()’ method. Obviously, in the StringBuilder class the ‘toString()’ method is overrode, the result is generated with the concatenation.

Alternate way to solve the conversion to string can be used when we know somehow in advance the number of elements of 't'. The String class Format() method can be used in this case. This method behaves like the Console.WriteLine, but its result is not written to the screen, but the string will be returned:

```
static void writeIt(params Object[] t)
{
  string reslt = String.Format("{0} {1} {2} {3}",t[0],t[1],t[2],t[3]);
  // ...
}
```

Here, of course, we must know the number of elements in the vector. But it helps us that the String.Format can also receive the values as ‘params’ vector values, so as one item we can pass the whole vector:

```
static void writeIt(params Object[] t)
{
  string reslt = String.Format("{0} {1} {2} {3}", t);
  // ...
}
```

Of course, we might also ask for the format string as a parameter. Practically the Console.WriteLine ask us as well as the first parameter before the additional ones. In the first parameter we can refer to the items in the ‘params’ vector by their index. In this case the first parameter of the ‘writeIt()’ function must be string, the others can be values of any type. Then it's easier to use the String.Format:

```
static void writeIt(string formatStr, params Object[] t)
{
  string reslt = String.Format(formatStr, t);
  // ...
}
```

If we build up this function this way, the call should be easy:

```
duck d = new duck("donald");
writeIt("{0} {1} {2}", 12.5, true, d);
```

In this case the first parameter is a function of the format string, then the 't' vector will be three element length, containing 12.5, true, and the 'd' duck.

With the help of 'params' keyword not only an Object vector can be built!

```
static int Maximum(params int[] t)
{
  if (t == null || t.Length == 0) return int.MaxValue;
  var m = t[0];
```

```
  foreach (int x in t)

     if (x > m) m = x;

  return m;

}
```

This function can receive any number of parameters, but each of them must be int. From the vector we select the largest number, and return it as a result. Only int-typed parameters can be accepted, because it is given after the keyword 'params' as the base type of the vector.

18. 18. The abstract classes

The idea seems clear that when a developer starts creating a class, he knows all the information about it, and all the methods can be written. This is often true.

However, there are situations when the developer knows that a method is/will be needed, but the method cannot be written. A sub-tasks arises, to solve that a method must be developed. We can design the name of the method, we can determine the parameters of the method, but we cannot write the body! Why do we know this? Most of the time we develop a method body and we reach a sub-task we cannot solve. We want to continue the developing of the method body, after the sub-task. We refactor this task from the function into a separate method, and call it.

Let's look at an example: a simulation of wildlife we want to write about. In a closed area (farm, forest) there live some animals. The animals live, eat, move, etc. there. We will have a variety of animals, each are slightly different from each other, how they move, eat, reproduce themselves. We will have two different types: herbivores and carnivores. We would like to build a base class "animal", and then specialize to "herbivore" and "carnivore" directions, and then further specialize them for specific animals. We create an "area" class, which helps us to manage the location of the animals, and contains other supportive methods to find an animal, or empty spaces in the area where the animals can wander to, find other animals in given specie, or find preys for the predators.

Without attempting to be comprehensive let’s examine the rules of reproduction. The criteria of the reproduction (simplified version) are:

- being female,
- adequate aged animals (not too young, not too old),
- sufficient time has elapsed since the previous reproduction,
- well-feed animal (healthy),
- an appropriate male animal is nearby,
- the newly born young animal will have enough free space around (starting position).

We're trying to plan what fields are needed, and how to write method. First, we introduce an auxiliary class, which coordinates can be stored (in its own local coordinate within that area, fields are empty coordinates, coordinates of prey etc).

We're trying to plan what fields and methods are needed. First, we introduce an assistant class to store coordinates (a location of an animal within the area, empty fields, a prey animal, etc).

```
class coord

{

  public int x;

  public int y;

  public coord(int x, int y)
```

```
  {
    this.x = x;
    this.y = y;
  }
}
```

In a child class we store the animal at a given coordinate:

```
class living : coord
{
  public animal beast = null;
  public elo(int x, int y, animal beast)
    : base(x, y)
  {
    this.beast = beast;
  }
}
```

The animal class consists of the following fields to store data:

```
enum sexes { male, female, other }
class animal
{
  protected int age; // in month
  protected sexes sex; // male or female
  protected int lastProd; // how old was in the last production
  protected int wellFed; // scale [0..10], 0=starving, ..., 10=well
  protected coord kpos; // the position in the are
}
```

The method which produces the baby animal is written as a virtual method, the child classes probably want to redefine it according to changing some rules:

```
public virtual void reproducing()
{
  // #1: is it a female
  if (this.sex != sexes.female) return;
  // #2: is its age appropriate ?
  if ( ???? )
}
```

The first problem: how to deal with the age? This interval is highly depends on what kind of the animal we are talk about. An elephant become sexually mature at the age of 12 years, while red foxes at 10 months. We cannot

store the initial value of puberty in a constant. Which is the good value? How to modify its value when we develop another kind of animal class? For similar reasons, we cannot store it in a class-level field, as for an elephant baby we would write 144 months, and for a red fox baby we would change it to 10 months, which will override the value for the elephant babies as well.

We can store these values (the start and the end values of the interval of being sexually mature) in instance level fields, but in this case for every red fox animal we would store the same (int) values differently (int values, each is 4 byte). It is not a good solution.

Instead of this, we can create a function to check the age of the animal instance. This function can hold the proper (minimum and maximum) values as literals:

```
protected virtual bool isMatured()

{

  // good for red foxes

  if (10 <= this.age) return true;

  else return false;

}
```

Assuming that we have this function - we can continue the development of the method:

```
public virtual void reproducing()

{

  // #1: is it a female

  if (this.sex != sexes.female) return;

  // #2: is its age appropriate ?

  if ( isMatured()==false ) return;

}
```

Yes, but what should we write into the ‘isMatured()' function at the base ‘animal’ class level? This class is at a very low level, it has no knowledge about anything of the proper values, it knows nothing about what kind of animal classes will be developed later. Let's look at three solutions!

Solution #1: write the method with some fictitious data:

```
class animal

{

  protected virtual bool isMatured()

  {

    if (0 <= this.age) return true;

    else return false;

  }



  public virtual void reproducing()

  {

    // #1: is it a female
```

```
    if (this.sex != sexes.female) return;

    // #2: is its age appropriate ?

    if ( isMatured()==false ) return;

  }
```

The solution is disadvantageous for two reasons:

· The virtual methods are not required to be overridden in the child classes. So the developer of the child class might forget to override, but it won't turn out as the child class will inherit a working function. The simulation will work, but some animals will produce a lot of (or a small amount of) babies. The analysis will point at that the very young ones starts to make babies immediately, exploring the reason: we forget to rewrite this virtual method!

· This method is a small one, contains only two lines of code, but in fact it is completely unnecessary, as its fate is to be overridden in each child class. However, the compiled version of this method is stored into the .exe file, and allocates some spaces in the memory although normally it is never called.

Solution #2: write the method, but in fact leave the body empty. It is not easy in this case, since the function must return a boolean value. Return false:

```
protected virtual bool isMatured()

{

  return false;

}
```

There is a better solution, because there is a function body again, with no real benefit. Now it is lucky that we found a return value, which is quite revealing, if a child forgets to override, their instances will disappear soon from the simulation as it never create baby animal instances. But again, this would mean that we would think that everything is all right, the program is complete and correct. Only the canalization of the run time log will reveal the fact of the error.

Solution #3: write the method, but in the body we throw an exception only!

```
protected virtual bool isMatured()

{

  throw new NotImplementedException(" isMatured() ");

}
```

The situation is a little bit better, as at the execution of the program reaches this method, it will throw an exception, and this will probably terminate the program. Of course, this still means that the compilation is successful, we can start the program (because we think everything is fine), but soon the run time error will indicate that we forget about something. At least very quickly it will find out what is wrong, what should be done to make the code complete. The function body is short; it is a fairly good solution.

Each of the three previous solutions have a common behaviour: the compiler does not show a syntax error, only at run time will it turn out that the code is not completely finished. The reason is: the virtual method is not mandatory to rewrite, so the compiler does not warn us for child classes which do not contain any overridden version.

The best solution for the problem is to make the compiler that it is not just a virtual method, but one that must be rewritten. Such a function must be marked with the 'abstract' keyword. Another advantage is that an abstract method has no body at all:

```
class animal
```

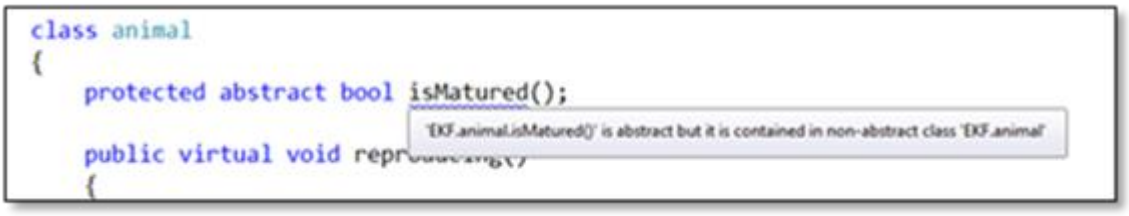
```
{

   protected abstract bool isMatured();
```

Note the following:

· we cannot use the 'virtual' keyword with the ‘abstract’ keyword, as abstract includes the meaning of virtual as well,

· an abstract method is not followed by a body (code between {...}), but is closed immediately using a semicolon.

Another basic information: if there is an abstract method in a class, the class must be marked with the abstract keyword as well (abstract class) - otherwise a syntax error is indicated. The following figure shows the problem (“*animal.isMatured() is abstract but is contained in a non-abstract class ’animal’”):*



Therefore the solution is to mark the class with the same keyword:

```
abstract class animal

{

   protected abstract bool isMatured();
```

Let see what we collected. An abstract method must be redefined in the child classes. If we forget it, a syntax error is raised. In other words, the compiler warns us that we forgot to override the method!

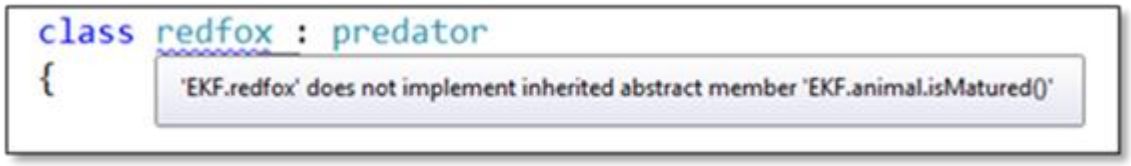
This error is indicated at the name of the class: “*predator does not implement inherited abstract member animal.isMatured()”.*

Developer of the child class will receive a warning not to forget to override the method. Unfortunately, the 'predator' class still cannot write this method, since it still does not know any maturity data. Because the compiler is very stubborn, we should redefine this method. But how? Again, we still can use solution #1 or #3, but the same problems will arise. Moreover, we will commit a big mistake! When we override an abstract method, the pressure passes. The child class of a predator (red fox class) will not be reminded, do not forget to override the method. Because an overridden method is not obligatory to override again. Do not choose this way!

Since the 'predator' class contains an abstract method (inherited), why not mark it with the 'abstract' keyword again?

```
abstract class predator : animal

{
```

The compiler will now find out that although the method is not overwritten, but we never will. Since the class is marked, the code is syntactically correct! We are ready to create a child class:



The same error message, the same root problem. We must write the method, or the class must be marked with ‘abstract’. Since we can write the method in this case, select the first option:

```
class redfox : animal

{

  protected override bool isMatured()

  {

    if (this.age >= 10) return true;

    else return false;

  }
```

An abstract method must be redefined with the “override” keyword. As it is overwritten, the class must not be marked with the ‘abstract’ keyword any further:

If you we take a closer look to the 'isMatured()' method, in fact, it might convert to a read-only (with only a get part) property as well. Can a property be abstract? Of course!

```
abstract class animal

{

  protected abstract bool isMatured { get; }
```

As we can see, in the case of an abstract property, we must specify the type of property (bool), and which parts need to be developed in the child classes. In this case, only the ‘get’ part is required. If we need both parts, we must specify the '... {get, set;}' form in the source code.

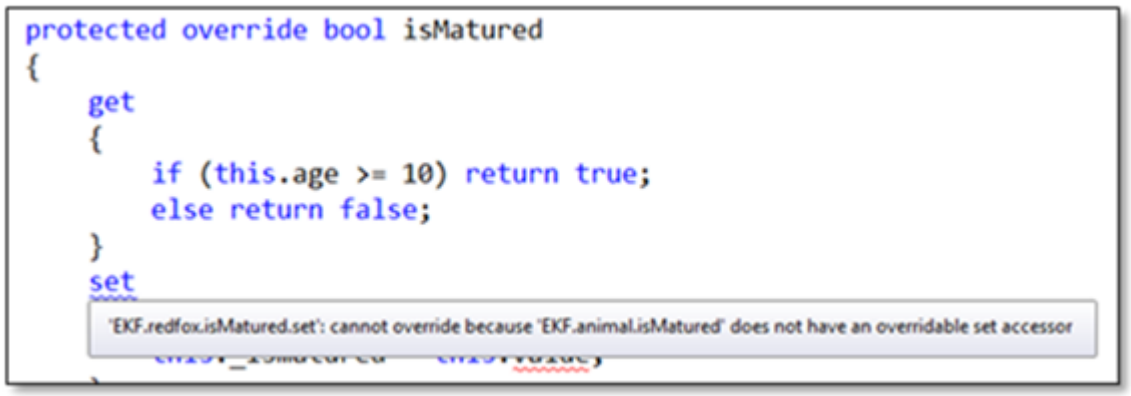
```
abstract class predator : animal

{
```

Because the predator has not developed the abstract property (still contains abstract elements), it also marked with the abstract keyword.

```
class redfox : predator

{

  protected override bool isMatured

  {

    get

    {

      if (this.age >= 10) return true;

      else return false;

    }

  }
```

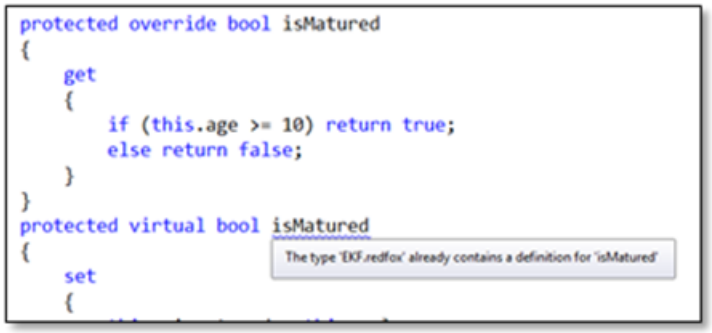
The red fox class overrides the property, so it must not be marked with the ‘abstract’ keyword.

A question arises: is it possible not only the ‘get’, but also to develop a ‘set’ part as well? Give it a try - but unfortunately the compiler indicates an error:



The message is “*isMatured.set cannot override because animal.isMatured does not have an overridable set accessor*” shown. The problem is quite simple. In the next chapter we will discuss the structure of the VMT table and it will be turn out that if a base class does not contain a 'virtual' method (now in ‘abstract’ form), it is not added as an item into the VMT table, so that the override will not be able to change this item in the table.

There is no other choice; we must try to divide the property into two:



The error message *‘redfox’ already contains a definition for ‘isMatured’*"' indicates we cannot.

Last notes:

· Constructors cannot be abstract because it cannot be virtual. In fact, the child class would not be able to overwrite a constructor as the name of the constructors in the child must match with the name of the child class, so the name must change.

· A field cannot be abstract, since the declaration of a field must include the type, name, so everything is given, there is no attribute which cannot be specified at the declaration of a field.

· Destructors cannot be abstract, they are the overridden version of the Finalize() method. Therefore, to create a destructor is not mandatory.

19. 19. VMT and DMT

In OOP we cannot talk about classic function call because the functions are called methods, and they are always part of an object class. There are three cases when we call a method:

· **Class-level method:** to identify the method to call, we must specify the class name as well. If this class contains more than one method with the same name, then the actual parameter list identifies the method to call. If there are other classes containing methods with the same name, it causes no problems, since we identify the class as well. So it is very clear which method we want to call. Therefore, the compiler uses early binding every time to call a class-level method.

· **Instance-level method but not virtual**: we can call these kind of methods, their names preceded by an instance name. Earlier we must declare the instance variable, and at that point we specified its type (static type). The method contained by this class will be called later. The containment means that the method is firstly presented in this class, or was inherited by it. Since the method is not virtual, if it is inherited, in the class we cannot redefine it with the “override” but with the “new” keyword. When a method is call through an instance variable, a search is started from the static class backward in the inheritance tree until the newest version of the method is found. It is selected to call. As the starting point of the search is known at the compile time, it can be finished at compile time. The early binding is used every time in this case.

· **Virtual instance-level method**: the problem is that the compiler cannot base its decisions on the static type of the instance in this case. Due to the type compatibly the static type must be interpreted as: "at least this type". The current value of the instance variable can from this type or any child types. The compiler cannot determine which child class is the current value type, so it cannot determine which method version must be called. For the lack of information the compiler must apply late-binding.

In the early binding, the compiler is able to recognize all aspects of the method call at compile time. The code generated by the compiler holds the point of the program where the execution will continue, where the method entry point which is called is. In the machine code level the 'call' instruction can be used. Normally the 'call' has a parameter, a memory address, where the called method starts.

In the late binding, the point in the memory cannot be identified at compile time. However, it is a function call as well; code must be generated by the compiler as well. The 'call' instruction cannot be used here as the compiler doesn’t know the address of the method entry point. What code is generated in the case of late binding?

19.1. 19.1. The VMT table

How does the compiler work? It reads the source code, divides it into small parts, reads them and compile them one-by-one, and builds larger units. The code generation is finishes when a larger block is understood and ready. Such a large block is a class. It needs to know the class name, where it begins and ends, what kind of fields, properties, and methods are included, and their details before the code generation starts. This also means that until a class is generated, work with no other classes.

When the source code of a class has been collected, a table[21] is built, called the **Virtual Method Table**, abbreviated as VMT. This is a table with a special role, contains information about only virtual methods (and properties). Now focus on the case of methods, on the basis of it case of the properties will be understandable as well.

The VMT table is built by the compiler during the reading of the source code at compile time. At that moment the structure of the table is:

· the name of the method and its parameterization,

· the class containing this method,

· the memory address of this method.

The parameterization of the method is required because there can be methods with the same name but with different parameterization, so the method name is not enough alone. To simplify the matter, disregard this attribute now. The memory address is determined by the compiler at the code generation phase, the actual values we put a mark there.

The table is made by the following algorithm:

· start the VMT table as empty (let it empty now, we will discuss it later)

· read the source code of the class, and when we meet the 'virtual' or 'abstract' keyword (introducing a new method), we add a new row to the end of the table,

· when we meet an 'override' keyword, find the appropriate row in the table and change the first column (containing the class name) to the actual class name.

The following is a simplified example where the name of the methods and the parameterization is not so important:

```
class probe1

{

  public void A() { /* ... */ }

  public virtual void B() { /* ... */ }

  public virtual void E() { /* ... */ }

  public void D() { /* ... */ }

}
```

The corresponding VMT table looks as follows:

class 'probe1'		**VMT**
probe1	B	[mem. addr of probe1.B]
probe1	E	[mem. addr of probe1.E]

In our case, the memory address (3rd column) contains the class name as well, we drop the 1st column:

class 'probe1'	**VMT**
B	[mem. addr of probe1.B]
E	[mem. addr of probe1.E]

The methods A() and D() are not present in the VMT, as they are non-virtual methods. A total of two entries are in the table, because we have only two virtual methods. Both are brand new, so - even though the initial state of the VMT table was empty - two new rows were added to it. Create a child class:

```
abstract class probe2 : probe1

{

  public abstract void C();

  public override void E() { base.E(); }

  public virtual void F() { /* ... */ }

}
```

The 'probe2' class inherits all the methods from its ancestor class, the virtual methods as well. This means the new VMT table will contains at least two rows as well, so the initial state is the same as the final state of the VMT of the base class. Furthermore, the "algorithm" above must be executed: we modify the rows of the table, and add new rows to the end:

class 'probe2'	**VMT**
B	[mem. addr of proba1.B]
E	[mem. addr of **proba2**.E]

C	-
F	[mem. addr of proba2.F]

In the case of the abstract method C() a new row is added to the end of the table, but the memory address cannot be filled because there is no body for this method (no entry point is defined). Because of the ‘override’ we modify a table row. The ‘virtual’ method ‘F’ is added to the end of the table.

What is the difference, when 'probe2' contains the same methods but in a different order? Let's see:

```
abstract class probe2 : probe1

{

  public override void E() { base.E(); }

  public virtual void F() { /* ... */ }

  public abstract void C();

}
```

In this case the two rows at the end of the table might be exchanged, the order for the last two entries in the table may be inverted, but the first two entries in the table won’t be changed, and the resulting VMT still will contain four rows. The table is essentially not changed:

class ’probe2’	**VMT**
B	[mem. addr of proba1.B]
E	[mem. addr of **proba2**.E]
F	[mem. addr of proba2.F]
C	-

If virtual property is added, the same things happens (starting from the later version of the VMT table):

```
abstract class probe3 : probe2

{

  public abstract int H { get; }

  public override void C() { }

  public virtual double G

  {

    get { return 0; }

    set { }

  }

}
```

class ’probe2’	**VMT**

B	[mem. addr of proba1.B]
E	[mem. addr of proba2.E]
F	[mem. addr of proba2.F]
C	[mem. addr of **proba3.**C]
H.get	-
G.get	[mem. addr of proba3.G.get]
G.set	[mem. addr of proba3.G.set]

The get and set make different entries to the table, as they can be used separately. The property H is abstract, so there is no memory address. The method C() has an overridden version, so a memory address is presented.

Note the following:

· The VMT table of a child class always contains at least as many rows as the base class, as the VMT of the child class starts with copying the VMT of the base class.

· The first rows of the VMT table contain the same methods in the same order as in the base class VMT table. If there is an overridden version of a method the memory address is replaced.

· New entries may be added to the end of the table.

· In the case of abstract classes, there might be entries in the table where the memory addresses are not filled.

Let's go back to the late binding! Supposing that for a 'p' instance we call method E():

```
static void E_call(probe2 p)

{

  p.E();

}
```

In this case, the call of ‘E_call()’ method and the actual instance which is passed as a parameter determines the dynamic type of ‘p’, so the compiler cannot know it at compile time. Late binding is used, as the compiler detects that in the 'probe2' class (the static type) the method E() is a virtual method.

What does the compiler do? What code is generated for 'p.E()' call?

1. determine the dynamic type of ‘p’,
2. look up the VMT of this class for entry of 'E',
3. extract the memory address of ‘E’ from this row,
4. jump to this method, it is to be called.

Let's see in practice (not forgetting that probe2 and probe3 are abstract classes in real, so practically we cannot instantiate them, but for this small example let it be allowed as the method E() is not an abstract one, and calls no other abstract method):

```
probe1 a = new probe1();

probe2 b = new probe2();
```

```
probe2 c = new probe3();

probe3 d = new probe3();

E_call(a);

E_call(b);

E_call(c);

E_call(d);
```

The case of 'E_call(a)', the dynamic type will be probe1, so the VMT table is used, the 2nd row targets the probe1.E, it holds a valid memory address, so this is called.

At 'E_call(b)' the dynamic type will be 'probe2', its VMT table will be selected. The 2nd row will contain the entry for 'E', which targets 'proba2.E()' so it is called. It's okay, because for instance 'b' this is the latest method version.

The case 'E_call(c)' the dynamic type is 'probe3' (although the static type of 'c' is probe2, but it is unimportant for now). In the VMT table of probe3, there is the entry 'E' in the 2nd row, which holds the memory address of 'probe2.E()' (since it is not overridden in probe3). Thus, the 'probe2.E ()' method is called.

It is very similar to 'E_call(d)', as the dynamic type of 'd' is also probe3, so the 'probe2.E()' is called again.

There are only two things to understand:

· Actually we do not determine the dynamic type of the instance, instead we simple store the memory address of the proper VMT table to the instance in a specific field (which increases the memory requirement of the instance by 4 bytes). We call this field the 'VMT' field.

· We do not need to "search for" the necessary row in the VMT table, as for a given method it is always the same row in all the VMT tables. The reason for this is that the child classes VMT table starts with copying the VMT table of the base class. If method E() is in the 2nd row in the base class, it will the 2nd row also in the case of each child classes.

The line number of the row in the VMT tables is known at the compilation time. As the compiler knows the static type of 'p' (probe2), and in the VMT of probe2 the method E() is in the 2nd row, this information can be used for the code generation phase as well.

Since we do not need to search for the row at runtime, the 1st column of the VMT is not required at runtime. For this reason the VMT contains only one column at runtime, containing only memory addresses.

```
static void E_call(probe2 p)

{

  p.E();

}
```

The code for 'p.E()' using late binding is the following:

```
read the the memory address of p.VMT[2]

jump to this memory address
```

Some important things to know:

· A memory address is a 4 bytes thing[22] (it does not matter that it is a memory address of data elements of a function),

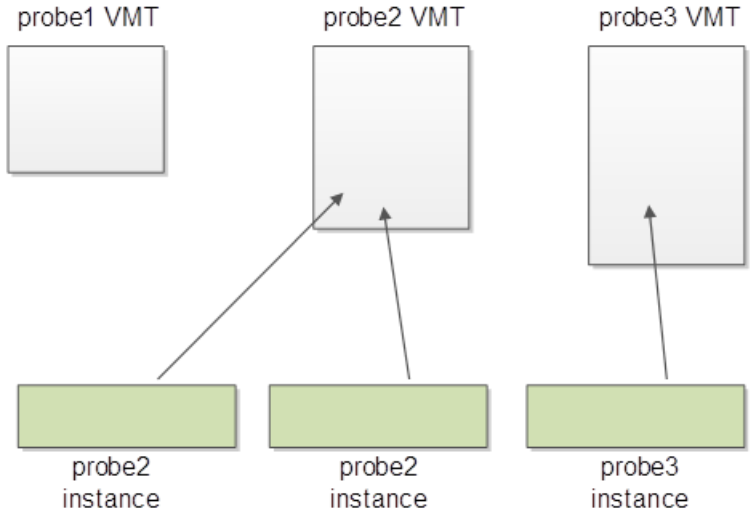
· So the memory requirement for a VMT table can be determined by the number of rows, each row is requires 4 bytes.

· A VMT table requires as many bytes during runtime as described above.

· A VMT table for a class is always in the memory, no matter when there is no instance from its class.

· A VMT table for a given class is presented in the memory only once, no matter how many instances are there. Each instances points to the same VMT table.

· This address field is defined by the instance level constructor of the class, so this field is automatically filled up during the instantiation process.



19.2. 19.2. The DMT table

The structure of the VMT table is simple; with this method the implementation of the late binding is simple and fast. However we have one note: if a child class does not override any methods, the child class VMT contains the same rows with the same data as are in the ancestor class VMT. We begin to create the child class VMT by copying all rows from the ancestor VMT, so in an extreme case (no override, not virtual, no abstract keyword is used in the child class) the two VMTs are identical. Thus, the process seems to be unnecessarily and is a waste of memory.

We present a possible alternative method for the same problem. The alternative method is called Dynamic Method Table (shortly DMT). The main point is to try to save memory in such case. When a child class does not override a method, the table does not store the same unhanged entry data as was in the ancestor DMT. Only those entries are stored, which were modified compared to the ancestor class DMT (overridden or new virtual one). We hope this means less memory allocation at the end.

Note that a DMT table is not complete, it does not contain all of the virtual methods. To resolve a late binding it means that examining a single DMT will not be enough. In an extreme case a child class DMT can be empty if it overrides no method, and does not introduce new ones.

Therefore, all DMT table contains a link to the ancestor class DMT. If we want to find a method in the child table, but it is not there, the search must continue in the ancestor table. If it is not success, then continue in the ancestor's ancestor table. Accordingly, the DMT tables of the classes presented in the previous section looks as:

class 'probe1' DMT	
	[link to Object.DMT]
B	[mem.addr. of probe1.B]
E	[mem.addr. of probe1.E]

class 'probe2' DMT	
	[link to proba1.DMT]
E	[mem.addr. of proba2.E]
C	-
F	[mem.addr. of proba2.F]

class 'probe3' DMT	
	[link to proba2.DMT]
C	[mem.addr. of proba3.C]
H.get	-
G.get	[mem.addr. of proba3.G.get]
G.set	[mem.addr. of proba3.G.set]

What kind of code can be generated to resolve a late binding when using the DMT table? The code is not as simple as in the case of VMT. For example it is no longer true that method 'E()' is always in the 2nd row. It is not even true that method 'E()' is in the DMT table. When method 'E()' is presents in the table, it can be in almost any row. Therefore, it is no longer enough to store memory addresses, we must store information about which memory address belongs to which method. How?

Do not store the method name, because the names are long. The comparison of strings is a very slow process. Instead, give each virtual method a number (id). Let's start numbering them with 0, and when the developer introduces a new method (abstract or virtual keyword) then increase the counter and assign them a new number. When we override one, we must assign the same id to it the same owned by the inherited old method.

Storing numbers is also space-consuming. How much space we separate for this purpose? If we give 1 byte, the numbers can be between only 0..255. If 2 bytes, then the 0..65535, in a case of 4 bytes it is 0...4 milliards. The first choice (1 byte) appears to be insufficient. The 2 bytes makes us uncertain: in a large project we might have a lot of methods, maybe more than 60.000. The 4 bytes seems to be enough for all kind of very large projects. For now we select the 2 bytes method for the sake of simplicity. Each row in the DMT table then costs 2 + 4 = 6 bytes (method id 2 bytes plus the memory address is still 4 bytes). In addition, each tablet contains a link to another (ancestor) DMT, which is a memory address, so it is a extra 4 bytes.

Accordingly:

· VMT for 'probe1' contains 2 rows = 8 bytes,

DMT table with 2 rows = 4+ 2 × (2 + 4) = 16 bytes,

· VMT for 'probe2' contains 4 rows = 16 bytes,

DMT table with 3 rows = 4 + 3 × (2 + 4) = 22 bytes,

· VMT for 'probe3' contains 7 rows = 28 bytes,

DMT table with 4 rows = 4 + 4 × (2 + 4) = 28 bytes.

We can see that the DMT method does not necessarily uses less memory. Obviously it depends on the example, in these examples we quite often introduced new virtual methods, and often overridden them. Of course, the DMT can allocate less memory than a VMT when the child class varies little.

Let's see the second aspect, the execution speed. The late binding with a VMT was a two-step thing, we must read from the table a specified row (1 memory read operation), and then jump to the given address. In the case of DMT it is not known at runtime, which is the row holding the information, in fact, not even known if this information presented in the table at all.

Instead, the table holds identification number. The compiler knows at compile time, when the late-binding code is generating, the proper id of the method to call (indicate it with 'M'). Therefore, a code is generated as:

#1: try to find the row of 'M' inside the actual DMT,

#2: if failed, go back by the link to the ancestor class DMT, and continue with step #1.

For 'finding a row' task we have two basic search algorithms. The first is a sequential search, we start at the first row of the table then we go to the next one again and again until we examine the rows for the given id. Obviously, this method is quite slow.

The second method is the binary search. This requires the table to be sorted by the ids of methods. This may be completed at compile time, as the compiler has enough time to sort the DMT table.

The search starts with the middle row of the table. If it is not what we are looking for, then choose the lower or the upper half table to continue. We examine the middle element of the half table, and then we continue the search. Until we find the row or the remaining array to be searched is reduced to zero - then we can conclude that the table does not contain the method.

Thus the binary search can be useful, but it is also noticeable that the search is much more time consuming than in the case of VMT. Moreover, a variable amount of time, as it is very fortunate to find the proper row for the very first try; otherwise we must search many tables over backwards until we find the id of the method.

In summary:

· The VMT despite being quite bulky, does not necessarily allocate more memory than the DMT.

· The DMT usually allocates less memory than the VMT, when little modification are done in the child class.

· A method call has a much slower execution using DMT at runtime.

Both ways we can be solve with the late binding. Nevertheless, most of the programming language is based on the VMT table method, as it is predictable fast, and the waste of memory is less important nowadays, when 4 GB of RAM is costs less than 5 thousand HUFs.

Some programming languages[23], however, leave the developers to decide which method they want to use. We can choose between VMT and DMT table methods. In this case:

· When during the design time we have a strong feeling that the method will be rarely redefined in the child classes, it is better to put into DMT. Namely when the child classes redefines the method, each DMT will contain a row for this method, each costs at least 6 bytes, while a VMT row costs only 4 bytes.

· If the method is often called (eg, inside a loop), then it is better to put into VMT, to ensure the maximum execution speed[24]. If the method is rarely called the DMT method can be selected.

20. 20. Partial classes

When OOP developers work as a team, there are many ways they can divide the work among themselves. First, the designers prepare the plans of classes, which are required to assemble the entire program. Determine what methods should be, and what kind of tasks and the parameterizations. They can generate a draft code of the classes, which contains the methods, but with an empty body. Developers divide among themselves, which class belongs to which developers, and then they fill the empty bodies with actual codes.

But what happens when we have a huge object class, a lot of method, properties, complicate task. Can multiple developers to work on it simultaneously?

The answer is usually no, at least not easily. The source code of a class ultimately a simple text file, and there are a few tools which allows more than one developer to work on, edit the very same text file simultaneously.

It can be solved in such a way that one of the developer works on a base class, most of the methods are abstract, the second developer works on a child class to write the abstract methods. So technically there are two classes, two source codes. Obviously, the rest of the developers work with the child class only, as it is not abstract anymore, all methods are completed. A problem is that each abstract methods are virtual as well, and therefore the early-bound non-virtual methods cannot be left to the developer of the child class to make it complete.

To simplify this process, in the C # language a new concept is introduced, which is very easy to understand. We can split a class into two (or even more than two parts), put the code parts into physically different source code. The important thing is to inform the compiler of this fact, so it can expect it and be able to merge the parts of the source code. This 'class' must be marked with the 'partial' keyword:

sourceCode1.cs:

```
namespace organisms.pets

{

    partial class dog : animal

    {

        protected double weight;

    }
```

sourceCode2.cs:

```
namespace organisms.pets

{

    partial class dog : animal

    {

        public void setWeight(double weight)

        {

            this.weight = weight;

        }

    }
```

Now two developers can work on the same class, one writes source code #1, the 2nd developer works on source code #2. There is no need for creating child classes, abstract methods.

Note: theoretically it is possible that both parts of the class are in the same source code. In practical terms it is a less typical case, but it should mean a syntax error.

Another typical application for a partial class when on the same class works one developer – and a computer program. Some parts of the class, especially the fields, the properties, constructors, but in many cases methods with certain task can be generated by a software tool. The OOP designers often use an UML design tools to plan the object classes. These are graphical tools, able to design the relationship between classes, the interactions of instances (instances calling methods of other instances etc.).

With the information entered into such tools part of the source code of a class can be generated. Can generate the fields (name, security level, type), and properties to them. The software created this way is a nice C# source code, a partial class form, added to the project. The part of the class which cannot be generated by this tool is

written by a developer (human) into another source code as well as a partial class. This is very good, because if the designer modifies something in the UML tool, it re-generates the partial class again, without confusing the code created by the developer.

Two very important information:

· To let the compiler merging the two parts into one, both partial classes must be inserted into the same namespace.

· If the class have a specific base class, it is enough to indicate at one of the part. If we define the base class at both parts - the same class must be referred, otherwise the compiler raises a syntax error.

sourceCode1.cs:

```
namespace organisms.pets

{

    // the base class is defined here

    partial class dog : animal

    {

        protected double weight;

    }
```

sourceCode2.cs:

```
namespace organisms.pets

{

    // seems there is no base class here (but there is!!!)

    partial class dog

    {



        public void setWeight(double weight)

        {

            this.weight = weight;

        }

    }
```

21. 21. Destructors

The “constructor” is a method which executes for the very first time when the instance is created. Apart from some special syntactic rules they are basically conventional methods. Often, however, there is a need for methods which executes *at the end of the life* of the instance, to finish them. This method closes the activities started by the instance. Such is required in the case if the operating system is also involved into these activities. Most common cases:

· The instance starts to print something. It requires a permission of the operating system at the beginning, which enters the process into the print queue, but the actual printing starts only when the instance signals the successfully end of the process (completed), or interrupt it.

· The instance opens a file for writing: the operating system does not allow[25] for other processes to do this for the same file.

· The instance opens a network ports to enable other computers connections and receives data.

· The instance connected to a network port to communicate with and exchange data.

· The instance allocates a large amount of memory, not through instantiation (creating another instances), but with a direct call to the operating system[26].

· The instance connected to an external data source (e.g. SQL Server), there started some transaction processes, and shall indicate the end of the transaction or terminate the connection.

This 'closer' method previously has not been separated from the general-purpose methods. Only the developer knew the *Destroy()* or *Finish()* method is actually developer for this task. The call of such a method was manually (explicitly) coded: when the program executed, it was a method call at the appropriate point in the code. But the explicit call often caused problems:

· It was called too soon: the process was closed, but the instance lived, in the later call of its method tried to use the process as it was active (which raised a runtime error).

· It was called too late: the process could have been terminated earlier. (The optimal moment is important as the operation system cannot manage the resources well).

· It was not called at all: the call was not coded to the program or was inserted into a selection branch which was not executed.

Of these the first case is the most dangerous. It raises a runtime error in a remote point of the source code (where we still try to use the 'unclosed' state of the process). To find the point in time backward where we closed the process - not always an easy task, but it is always very time consuming.

The mistake is often due to the fact that the instance variables are really stores just memory addresses. The same memory address can be stored many different variables through simple assignment operators, can be added to lists, vectors, pass as parameters to method calls. Using any of these variables the 'finalizer' method can be called, and the other instance variables will drop the runtime error as the instance itself is closed. It also makes debugging more difficult.

While the method (explicit call) remained a significant part of the programmers required an alternative way, an automatic way. The strongest demand arose on allocating and releasing memory areas. At the very beginning the memory allocation of large and complex typed variables (arrays, records, lists, queues, stacks, etc.) and the release was done by manually (explicitly). After years of experience programmers rarely forget to allocate the memory, but often forget to free it up. This phenomenon is named as **memory leaks**. This meant that when we started the program, we had plenty of RAM, but as the program was running, the amount of free memory become lower and lower, as it would flow away through a hole. The program responsible for this was easy to find out, as it is simple to query the amount of allocated memory of all the active programs. The most engrossing project was responsible for this (usually). If we quit the program, it will suddenly become once again a lot of memory, because the operating system when the program is closed (more or less) frees all allocated by the program, but during the execution of the memory is not released.

So the symptom of memory leaks can be handled often by restarting the programs, but it is obviously not a solution to this problem. Some programs (such as services running on a server) which start with the server itself, and restarting this program will cause disruption of service (think on a web server service). As the problem is basically a programming error, so it must be solved by correcting the program.

These two problems are linked together and can be solved together. When memory of the instance is released, the processes left open can be closed.

The Garbage Collecting is a memory management technique. It will be discussed later, but already we know the essence of a mechanism: it discovers memory areas which can be released, and arrange for the releasing or reusing of them. Before the releasing the GC will invoke an instance method with a special role. The method can verify what processes are still active, and take action to close them. If this method finishes the execution - GC will free the memory associated with the instance.

For the GC to do this, it needs to know exactly which method to call. There cannot be more than one method, as GC will not choose between them. This method must not have any parameters.

The method is described by the above is named **the destructor**. So the destructor is a special method, written with special syntax (to let the GC recognize it), without any parameters, and is implicitly called by the runtime system.

The special syntax rules of the destructor are:

- the name (of this method) is the same as the class name,
- the name must be preceded by a tilde (~),
- the level of protection is missing (private),
- there is no return type (even the void is not marked).

Semantically:

- the destructor is fast, because the GC must wait until a it finishes its execution, which would not be practical,
- it checks whether there is any unfinished operation of the instance to be terminated,
- if so, must initiate the termination.

Constant and repetitive formulation error: the destructor is responsible for releasing memory areas. As we have seen, the releasing the memory actually belongs to the GC. Of course, the sentence can be true - if actually the instance allocated some extra memory outside the GC territory, then the destructor must free up that extra memory.

We need to understand that there are actually two "worlds". One is the world of operating system (it is cold and dark, cruel, procedural world). The operating system uses the Win32 programming model, which consists of thousands of functions, its favourite concept is the "handle", and fond of using 16-bit hexadecimal constants which are named as "flags", and there are also thousands of them. If we want to allocate memory directly by the help of the operating system, we can use e.g. Marshal.AllocHGlobal() class-level method, which receives a parameter to specify how much memory we need (in bytes). The other world is the world of NET Framework (bright, gentle, friendly), which is based on the OOP concept. Here we use the 'new' to allocate memory, which calculates the required number of bytes alone.

It is important to remember that GC deals with only the memory allocated by the 'new'. If we, the GC won't deal with the ones allocated through the Marshal class. So if we do that, in the destructor function we must call the Marshal.FreeHGlobal (...) function to release them. Then we can say that the destructor is responsible for freeing up memory. But remember: the destructor is never deals with deallocating the memory associated with the instance.

```
class externalRecord

{

  protected IntPtr extraMem;


  // constructor

  public externalRecord()

  {

    // allocating 200 bytes

    extraMem = Marshal.AllocHGlobal(200);

  }
```

```
  // destructor
  ~externalRecord()
  {
    // if has not been deallocated before
    if (extraMem!=IntPtr.Zero)
       Marshal.FreeHGlobal(extraMem);
  }
}
```

21.1. 21.1. If we do not write any destructor

The concept of the destructor existed before the C# language. For this reason, the C# language also included this concept. However, in C# there is no destructor actually (in the background working). Instead when we write a destructor we overrides an inherited method (from class Object the virtual Finalize() method is overridden). So the code above is the same as:

```
class externalRecord
{
  // ...
  // if has not been deallocated before
  public override void Finalize()
  {
     // ha meg nincs felszabaditva
     if (extraMem!=IntPtr.Zero)
          Marshal.FreeHGlobal(extraMem);
  }
}
```

It follows from the foregoing that if we do not write a destructor at all, there is one for our class, since we "inherit" one. The GC actually calls the 'Finalize()' method, and at least one is inherited for our class (eq. from the Object).

21.2. 21.2. When not to write any destructor?

There is one very important note on the GC. The GC at a moment handles not a single instance. Most of its time is spent on discovering of instances to release, and collecting their memory addresses. In addition, when an instance becomes unnecessary, it might engulf a lot of other instances as well. This is especially true to lists and arrays, which stores memory addresses of several other instances. If we lose a list, the instances in the list are usually lost. It is also often the case that an instance stores references of other instances in its fields (if the field type is a member of the reference type family).

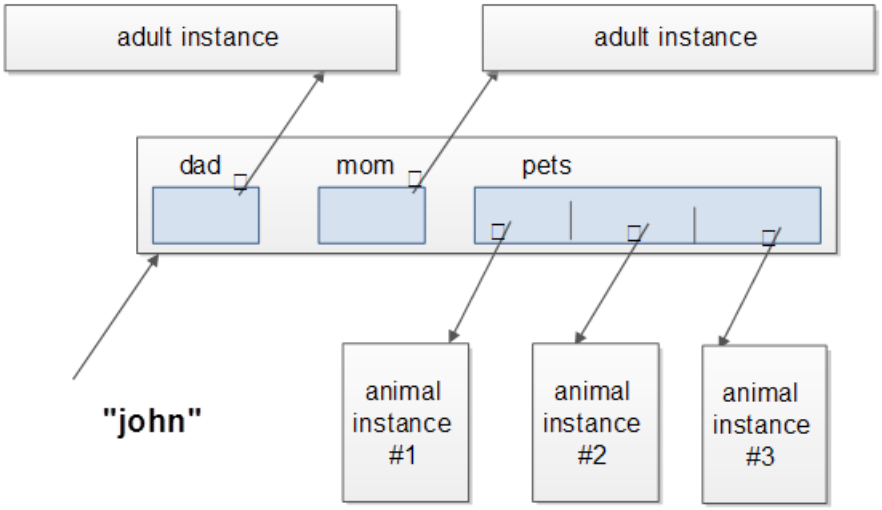
When the GC switches to 'releasing state', it owns memory addresses of several instances. From the view of GC, they are all the same; all of them are waiting for releasing. There exists no special order between them. In other words, the GC releases them one-by-one in a "random" order, this behaviour of the GC is a non-deterministic process.

Consider the following hypothetical case: a class (children) stores data of parents (father, mother), references in a list of favourite pets (dogs, cats):

```
class child

{

  protected adult dad;

  protected adult mom;

  protected List<animal> pets = new List<animal>();


  ~child()

  {

    Console.WriteLine("good-bye dear {0} mom", mom.name);

  }

}
```

You would think that the code above is correct, there's nothing wrong with it. It is possible that if you start it, it will work well, and prints "good-bye dear Anna mom." In other cases, however, it won't be printed. Nothing is displayed.

Let us examine how the memory looks like:



When we lose the memory address of "john", it is possible that we lose the father and mother instances as well, mainly when we do not store their memory address elsewhere. We might lose memory addresses of several animals. There are two possibilities:

- GC releases the child (john) first, then the "adult" mother,
- GC releases the mother first, then "john" itself.

The first behaviour seems more natural to us, because we think we lost the child first (mainly) in the program, but there is a reference to the mother from the (lost) child instance. But that's the GC never deal with: they are all the same, never examines the connection between the unnecessary instances. It cannot investigate it, as there might be a reference to the child in the mother instance:

```
class adult

{

   protected List<child> children = new List<child>();

   // ...
```

```
}
```

In this case the GC could easily get in trouble, because a good order cannot be determined. Simplest: do not deal with these internal links. Thus, which is a natural though for us - unfortunately is not the truth in real world. Finally the GC will set up an order, it may be the case which starts with “john” then “mom”, or the opposite order can happen as well. In the first case when “john” is reached, the mom instance still exists, the Console.WriteLine inside the destructor can write the name of the mother.

In the latter case the mother instance is released first, then GC hit “john”. In this case there is a memory address (a reference) in the filed ‘mom’ of john (where the adult instance *was* earlier), but it is invalid now. Then the name of mom cannot be written, as the mom instance does not exists anymore, the Console.WriteLine will drop an exception (run-time error). The GC detect it, but does not care about it too much – finishes releasing the memory of the child (john) instance.

An important rule: the references stored in the instance-level fields inside the destructor are usually not available, it is possible that the corresponding memory address is invalid, already been processed by the GC. Just the value-type fields can be used inside the destructor.

The idea might arise for us when we create a log class (similar to the one above) to close the log file inside the destructor. Suppose that the instance constructor opens the file (name given through a parameter) as a log file, and we give a simple method to close this file, but when it is forgotten to call, the destructor will close the file finally.

```
class fileLog

{

  protected StreamWriter w = null;

  public fileLog(string filenev)

  {

    this.w = new StreamWriter(filenev, true, Encoding.UTF8);

  }



  public void writeln(string esemeny, params object[] parms)

  {

    if (w!=null)

      this.w.WriteLine(esemeny, parms);

  }



  public void close()

  {

    if (this.w != null)

       this.w.Close();

    this.w = null;

  }



  // destruktor

  ~fileLog()
```

```
  {
     close();
  }
}
```

However think it over: if we lose the reference of our fileLog instance, at the same moment we lose the reference of the StreamWriter instance, although the field ‘w’ stores its memory address. The order of destroying these two lost instances is undeterministic. In other words, it is possible that the memory address of field 'w' is not 'null', but it is invalid. In this case the destructor calls the 'close()' method, which found that ‘w’ is not null, but calling the ‘w.Close()’ drops a runtime error.

Not important what happens to those instances in which we are linked to our instance. They are responsible to themselves, because the main instance is no longer able to reach them in the destructor. Cannot inform them that it is about to destroy, so there is no needing them anymore.

21.3. 21.3. When to create a destructor?

The findings described above under the title question is easy to answer. We'll almost never need to write destructor. As long as the other tasks. NET Framework help solve instances, you will not:

- If we want to connect to other computers, instantiate the TcpClient class,
- If we want to read from files, use a StreamReader class instance,
- If we wish to write to files, instantiate the StreamWriter class,
- If we want to print, a PrintDocument class instance is required.

Note: these tasks can usually be solved by other classes, other instances, these are only examples.

In other words, in C#, in .NET Framework most of the problems can be solved by instantiating an existing class. As we have seen, to other instances in a destructor cannot be referred as it is possible that their memory addresses are invalid (can no longer be used). This means writing a destructor usually is not worthy, but simply not possible.

The exception is if we do not use .NET Framework instance, but call an external, not managed code function, for example we call a Win32 environment function. In this case we often need to store an external memory address in our fields (or handle values[27]) which we often use an instance of the IntPtr class. These classes are not managed by the GC, as they are not reference type instances, but value type instances. At this case it is worth to deallocate the external memory areas and call the appropriate close function to the handles.

22. 22. Generic

Examine carefully our general-purpose complex data types. The simplest is the vector. Each vector are the same as they have a 'Length' property, which tells us how many elements are in the vector. Each vector is indexed from 0, each vector are given the chance to set or retrieve the value of an element identified by its index. What is the difference between vectors? It is the data type of an item! The base type of the vector must be given in its declaration:

```
int[] tt = new int[20];
```

The case of lists is very similar: each list has an 'Add (...)' method, and a list is able to tell the number of its elements with the property 'Count', we can query any value from the list by its index. What is the difference between vectors? It is the data type of an item!

How we create a list class? The list of item types is irrelevant for a property like 'Count', but it is very interesting for a method like 'Add (...)':

```
class ourList
```

```
{

  public abstract void Add(??? t);

  public abstract ??? this[int i] { get; set; }

}
```

Here is a problem where the keyword ‘abstract’ won’t help, because to replace ‘??’ a particular type name must be entered, even if the body of Add(…) is missing, as well as the body of the indexer (property ‘this’), without knowing the type name we cannot continue.

We must specify the type name even we use the keyword ‘abstract’, as in the child class in ‘override’ we can modify only the body not the parameterization. For this reason, we cannot actually write anything here, because if would write a concrete type name here (such as ‘int’), then it will never become a ‘double’ list.

The solution is the **generic** types. A generic type is a class, where we could develop their methods and properties - but one thing is missing what we don’t know at the moment when the development is starting: a name of a type which must be used to work with. Now this type is a kind of parameter of the class. The parameters of the class must be declared after the class name, not in parentheses (parameters of methods), not in square brackets (parameters of the indexer property), but in angle brackets:

```
class ourList<T>

{

   /* ... */

}
```

The above indicates that the 'ourList’ class has a type parameter which is referred as ‘T’ hereafter. The exact type represented by T, will be get known only in the instantiation:

```
ourList<double> ll = new ourList<double>();
```

In this case, the 'T' will means 'double' (T = double). Let us return to the class to write:

```
class ourList<T>

{

  public abstract void Add( T t);

  public abstract T this[int i] { get; set; }

}
```

The type “T” can be referenced anywhere in the class. Might be used as:

- type of method parameters, such as in Add (...)
- a method or property return type (such as indexing only)
- as a type of a field or constant,
- but can also appear in the body of a method, in variable declarations.

The class is syntactically complete and correct (although the actual value of T is not known at compile time). However, working code will only be generated if the specific type of T (instantiation) is known.

How do we solve a similar problem in a programming language, where the generic type concept does not exist? Similarly! Write a class for a given type:

```
class ourList double

{
```

```
  public abstract void Add(double t);

  public abstract double this[int i] { get; set; }

}
```

Then it should also work with other type, we select (highlights) the source code, copy and paste, and all word 'double' must be replaced to the given class name, (e.g. 'duck'):

```
class ourList_duck

{

  public abstract void Add(duck t);

  public abstract duck this[int i] { get; set; }

}
```

If we need another type (such as int), we can proceed in the same way, select, copy paste, change:

```
class ourList_int

{

  public abstract void Add(int t);

  public abstract int this[int i] { get; set; }

}
```

This method seems to work (no ‘generic’), but:

· during the "change" we can make a mistake. For example, if we look at the last version, the parameter of the indexer is ‘int’, as well as its return type. If we use this version to make a copy again, and then replace every 'int' word to a new type (eg. 'student'), it must be ensured that not all 'int' can be replaced, or we will get the following:

```
public abstract student this[student i] { get; set; }
```

· The source code of the original class is needed to apply the copy-paste properly. It means that the source code of the class must be given to the other programmers, which is not necessarily an advantage.

· If we have already made a lot of copies of the original class, but something we want to expand or improve in it later, we must copy these changing into all the replicas!

All the problems above can be solved by the ‘generic’ concept. A generic class can be compiled (syntactically complete and error-less), then the compiled binary form can be given to the other programmers. Since the source code of the generic class is defined only once, it can be expanded and improved on, it can be done in the original generic class, and nothing must be copied to the replicas as there are no replicas at all.

We must know there can be not only one type parameter exist, but many:

```
class sDoubleParams< T,M >

{

   public M field;

   public sDoubleParams(T a, M b)

   {

     this.field = b;

   }

   public M something(T x)
```

```
    {
        M a = field;
        /* .. */
        return a;
    }
}
```

In the instantiation each type parameter must be defined:

```
sDoubleParams<int,double> f = new sDoubleParams<int,double>(12, 44.3);
```

Here in this example 'T' means 'int', 'M' means 'double', and with the ‘replacements’ a full working instance can be compiled and generated.

Not only a class but a method can be generic. A typical example is the swap method, which is written in the way of generic, can be applied to swap the values of two variables of any kind:

```
public void Swap<T>(ref T lhs, ref T rhs)
{
    T temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

Calling this method we must specify the type of variables to which we want to implement the swap. If the ‘int’ is given, then of course, the parameters must be ‘int’ as well:

```
int a = 1;
int b = 2;


Swap<int>(ref a, ref b);
```

Cases have subtle errors can be obtained. For example, we want to write a function to select the maximum value from three values:

```
public T maximum<T>(T a, T b, T c)
{
  T x = a;
  if ( x<b ) x = b;
  if ( x<c ) x = c;
  return x;
}
```

Note that if we have three double values, the return type of the method can be double as well, so the 'T' also can be used as a return type. A temporary variable is needed ('x') with the same 'T' type. Unfortunately, the compilation of this code will not be successful because the compiler does not believe that all 'T' types are suitable to use with the function, as there does not exists comparison operators (< operator) to all types. For example, think about what would happen if there would be three ‘duck’-typed variables, k1, k2, k3, and we would call this method as:

```
duck m = maximum<duck>(k1, k2, k3);
```

How to compare two ducks, which is the greater? It is not impossible, for example, we can choose by their weights, it can be used as the base for comparison. But at this point we should see that the compiler cannot be sure that we have done it. It cannot be sure that the 'x' and 'b' variables can be compared with the < operator.

In fact, we must describe that "the method works with a T type, but only those T are acceptable, which has an < operator developed". This exactly cannot, but similar to this can be described with the help of the 'where' keyword (constraint):

```
public T maximum<T>(T a, T b, T c) where T: IComparable

{

  T x = a;

  if (x.CompareTo(b) < 0) x = b;

  if (x.CompareTo(c) < 0) x = c;

  return x;

}
```

Here it is described that the T type must implement the IComparable interface (the interface are discussed in the 23rd chapter), which in this case means that it is guaranteed that T must have a 'CompareTo(...)' method. It should also be used in the body of the method. The CompareTo method is used to determine the relationship of two variables, (eg in the form of 'a.CompareTo(b)'). The CompareTo returns -1 if a <b, return 0 if a == b, and +1 is returned when a> b. Thus, the CompareTo(...) can substitute the missing comparison operator.

Most important generic classes are inside the System.Collections.Generic namespace:

- the list class itself, which is written as List<T>,
- the Dictionary<T,M> which has two-parameters, it was mentioned in the chapter “the indexer” earlier,
- a universal Stack<T> class,
- a Queue<T> a universal queue,
- LinkedList<T> is very similar to the List, but its internal representation is a linked list.

23. 23. Interface

Let's start with the following problem: create a list class, which stores objects during the execution. When the user click on a "save" button, all the objects must be saved (as they have been changed during the execution of the program). As long as the user does not choose saving, no need to save. Therefore, in our program when something changes in an object, we will put them onto this list.

Handling this list must be solved. Items can be added to the list (each item only once, so specific attention must be paid to), then process them in a single iteration. Since we do not know how to save an object, we will ask them to save themself – by calling a Save() method of the item.

```
class saveList

{

  protected List<object> l = new List<object>();

  public void Add(Object p)

  {

    if (l.Contains(p) == false)
```

```
        l.Add(p);
  }
  public void SaveAll()
  {
    foreach (Object p in l)
       p.Save();
    l.Clear();
  }
}
```

There are two problems here:

· in method Add(...) the parameter type ‘Object’ is not good, as we cannot accept anything, but an instance which has a Save() method,

· in saveAll() the ‘Object’ is not good at ‘foreach’, as in this case we cannot write ‘p.Save()’ as an object has no ‘Save’ method (here a syntax error arises).

Let's start with the first problem. We want to press using a type (instead of Object) which owns a Save() method. Create a class like this:

```
class saveObj
{
  public virtual void Save()
  {
  }
}
```

We require all instances ever wanted to be added to this list to be inherited from this saveObj base call. Then it is sure they have a Save() method, and according to the type compatibility the type ‘saveObj’ can be used as a parameter type:

```
classsaveList
{
  protected List<saveObj> l = new List<saveObj>();
  public void Add(saveObj p)
  {
     if (l.Contains(p) == false)
         l.Add(p);
  }
  public void SaveAll()
  {
     foreach (saveObj p in l)
        p.Save();
```

```
      l.Clear();

   }

}
```

The idea is perfect; but there are some problems in the implementation. The first is easy to notice: useless in 'saveObj' the Save() method, as there is no body (cannot be implemented), and child classes will tend to forget about overriding this method. Again, we should use an abstract class, but the 'saveList' class therefore will not change in any way:

```
abstract class saveObj

{

   public abstract void Save();

}
```

Much bigger problem is that in the C# language it is a strong restriction that a class can have only one ancestor. This means that the classes developed in the program must use this very base class. Knowing this, however, in the beginning (when we start to develop our classes) is not insured. Assume that we develop an engineering program with a graphic user interface, and the templates are developed by another team of programmers. Those object classes know nothing about our 'saveObj' class, so they will not derived from it. Each template class might have a Save() method, but these classes are not derived from our ancestor 'saveObj' class.

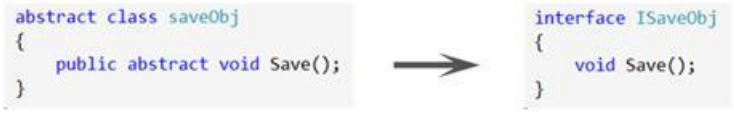
We must clarify our requirement. It was "the type of the parameter of method Add(...) must be type compatibly with the saveObj class (a child class)." In fact, what we wanted to require is that "the Add (...) method can receive only instances which has a Save() method."

To describe requirements like that the interface concept can be used.

· The interfaces are very similar to abstract classes: they contain prototypes[28] of methods and properties, since they never contain bodies.

· They are not classes, so the rule "there can be only one ancestor class" are not concerned to them.

The interfaces describe exactly what the problem was formulated at the beginning: "let this class a Save() method". The interface name usually starts with the large letter I (tradition). Compare an abstract class and an interface definition:



So the differences:

· keyword ‘interface’ instead of 'class' should be used,

· no need to use mark 'abstract' before 'interface',

· do not have to be used 'public' and other protection levels,

· methods or properties must not mark with the ‘abstract’ as well,

· while an abstract class can contain fields, an interface never can,

· while the abstract class can contains fully developed methods (methods with body), or properties, an interface never has one (fully developed method or property),

· the abstract class can contains constructors, a destructor, the interface never has any constructor or destructor,

· classes may have class level elements (fields, constants, methods), but never interfaces.

An interface is a description of a list of requirements, list of prototypes of methods and properties. However, an interface **is a type**, so it is capable of declaring variables, parameters, or can be appeared as basic types of vectors and lists:

```
class saveList

{

    protected List<ISaveObj> l = new List<ISaveObj>();

    public void Add(ISaveObj p)

    {

        if (l.Contains(p) == false)

            l.Add(p);

    }

    public void SaveAll()

    {

        foreach (ISaveObj p in l)

            p.Save();

        l.Clear();

    }

}
```

However, the interface as a type is not suitable to instantiate: "cannot create an instance of the abstract class or interface ISaveObj":

However, the interface type may appear as a “base class”, although it is not a class. Then according to the type compatibility the "child class" will be compatible with the interface type, and may be passed to the 'Add(...)' above:

```
class graphicalElement : graphicalBase, ISaveObj

{

    /* ... */

}
```

Since the interface does not count as a “class” type, it may be included to the list of ancestors, there can be more than one interfaces at the same time. The "child class" will be type compatible with all the interfaces on this list.

Of course a great price must be paid. If an interface is added to the list of ancestors, the class must develop (implement) all the methods and properties defined by the interfaces. The compiler will check it strictly and the punishment is hard: syntax error arises and refuses the compilation of the class:

When an interface is included into the list of ancestors, and develops all the required methods and class we might say "the interface is **implemented**". If a requirement is left undeveloped, the compiler writes the following message: "class does not implement interface member ISaveObj.Save ()" which means "the class does not contains the Save() method required by the interface ISaveObj".

In a case like this above we have several options available:

- develop the Save() method,
- remove the ISaveObj interface from the list of ancestors, which will removes the error message as well,
- we develop the method Save() as an abstract method.

The last option is interesting. The compiler understands that the class contains the method required by the interface (even though it is not). But do not forget, an abstract class cannot be instantiated until the abstract methods are developed. So after all the requirements are met, sooner or later, as this requirement is inherited to the child classes.

The question is what are the requirements about developing the Save() method:

- the methods required by an interface must be ‘public’,
- they can be ‘virtual’, but it is not required.

Since the methods in the interface cannot be marked by “abstract” or “virtual”, the class which develops these methods cannot mark[29] them “override” because they are not really an overridden version of an existing method:

```
class graphicalElement : graphicalBase, ISaveObj
{
  public void Save()
  {
     /* ... */
  }
}
```

However, sometimes one of the ancestors already contains the required method, even though it (for some reason) did not added the interface to its ancestor list:

```
class graphicalBase
{
  public void Save()
  {
    /* ... */
  }
}
```

Then the child class already owns the required method (inherited it), so one thing remained for him to do: add the interface to the ancestor list and enjoy the benefits of type compatibility:

```
class graphicalElement : graphicalBase, ISaveObj

{

    // the "public void Save()" is inherited, must do nothing else

}
```

If the base class contains the Save() method but in a form of ‘virtual’ (or ‘abstract’), and the child class wants to override it, of course, we can use the 'override' keyword. But it is important to know that in this case the ‘override’ is not due to the interface but due to the definition of the base class!

```
class graphicalBase

{

  public virtual void Save()

  {

     /* ... */

  }

}



class graphicalElement : graphicalBase, ISaveObj

{

  public override void Save()

  {

    /* ... */

  }

}
```

However it is, let us assume that the class has successfully implemented the ISaveObj interface. We have finished the ‘saveList’ as well. At this time we can connect the two things:

```
saveList m = new saveList ();

// adding one item

graphicalElement p = new graphicalElement();

m.Add( p );

// other items are adding

// ...

// then save everything

m.SaveAll();
```

The interfaces are used so describe correlations between classes (similarity), where the similarity comes not from the inheritance chain. The fact that the 'F15Tomcat' class owns a 'takeOff()" method, might means it was inherited of one of the ancestors (ie. from the 'fighterJet' class), and the 'duck' class has a 'takeOff()" method as

well, but there is a little chance that it is inherited from the 'fighterJet' class as well. An interface can describe this uniformity:

```
interface ICanFly

{

    void takeOff();

}
```

If both of these classes implement this interface, then the instances of both classes can be passed to the following method:

```
static public void saveYourLife(ICanFly p)

{

    p.takeOff();

}
```

According to the type compatibility, both function calls are acceptable:

```
F15Tomcat f = new F15Tomcat();

duck k = new duck ();

//

saveYourLife(f);

saveYourLife(k);
```

Let's look at an example of an interface which defines a property. Consider again a list that can hold objects of different ‘weights’ (like a bag). The list accepts only items which has a ‘weight’. The storage capacity of the list is limited; it can be specified in the constructor of the list instance. First, we define the interface:

```
interface IWeight

{

  double weight { get; }

}
```

Then develop the list class:

```
class bag

{

   protected double maxWeight;

   protected double _actualWeight;

   protected List<IWeight> l = new List<IWeight>();

   public bag(double maxWeight)

   {

      if (maxWeight<=0)

         throw new ArgumentException("bad max weight value");

      this.maxWeight = maxWeight;

      this._actualWeight = 0.0;
```

```
    }
    public void Add(IWeight a)
    {
        if (a.weight + this._actualWeight < this.maxWeight)
        {
            l.Add(a);
            this._actualWeight += a.weight;
        }
    }
    public double actualWeight
    {
        get { return _actualWeight; }
    }
}
```

Then the Add (...) method accepts instances as parameter and places to the list whose weight is known (and adds items to the list only when the bag won't be overweighted). We base it on that these elements have a '.weight' property, which surely has a 'get' part, so the weight can be read.

```
class duck : animal, IWeight
{
    protected double _weight;
    public double weight
    {
        get { return _weight; }
        set
        {
            if (value < 0 || value > 20.0)
                throw new ArgumentException("bad weight for a duck ");
            this._weight = value;
        }
    }
    /* ... */
}
```

23.1. 23.1. Generic interface

An interface can be generic as well. In the previous example, the weight property required by the interface – is a double value. But what if we want to describe the weight as an 'int' instead?

```
interface IWeight<T>
```

```
{

  T weight { get; }

}
```

In this case, the class ‘duck’ may indicate that he implements the interface 'IWeight' as an int-typed property:

```
class duck : animal, IWeight<int>

{

    protected int _weight;

    public int weight

    {

        get { return _suly; }

    }

}
```

When we try to write a generic ‘bag' class, we face with serious problems:

```
class bag<T>
{
    protected T maxWeight;
    protected T _actualWeight;
    protected List<IWeight<T>> l = new List<IWeight<T>>();
    public bag(T maxWeight)
    {
        if (maxWeight <= 0)
            throw new ArgumentException("bad max weight value");
        this.maxWeight = maxWeight;
        this._actualWeight = 0;
    }
    public void Add(IWeight<T> a)
    {
        if (a.weight + this._actualWeight < this.maxWeight)
        {
            l.Add(a);
            this._actualWeight += a.weight;
        }
    }
    public T actualWeight
    {
        get { return _actualWeight; }
    }
}
```

The reasons of the syntax errors (red lines) are the T type, as the compiler does not know anything about it. It does not know that a T-type element is comparable to 0 or not, can be examined whether it is less than or equal to zero. It cannot be known as a field of type T ‘_actualWeight’ can be initialized by 0 or not. It is not known that two T typed value can added together, and is it possible to examine whether that amount is less than a third T-typed value, etc.

23.2. 23.2. Inheritance between interfaces

There exists the inheritance among interfaces, but a base of an interface can be only another interface (a class cannot). Above this rule, inheritance works the same as among classes.

```
interface ICanFly
```

```
{
    void takeOff();
    void landing();
}

interface IFighterJet : ICanFly
{
    void aim(Object target);
    void shoot();
}
```

It is obvious that the class wants to implement the interface IFighterJet must develop all four methods. However, at the end it will be type compatible with the ICanCly interface as well.

23.3. 23.3. IEnumerable and foreach

In C# we know about the *foreach* that is capable of working with vectors and lists, it reads the elements one by one. In fact, the foreach is much more universal than this: it can work with any class instances which implements the IEnumerable and IEnumerator interfaces. It should be imagined that the instance to process (list, vector, anything) must implement the IEnumerable interface. It is an easy interface to implement, as it has only one function 'GetEnumerator()' to create. This function must return an instance which implements the IEnumerator interface. This instance must work with the *foreach*: the *foreach* will order this instance to retrieve the 1st element, then the 2nd, and so on.

Let's develop a 'backyard' class, which is a 20 × 20 matrix, each element of the matrix represents a small area where one animal stands at a moment. We have several kinds of animals (ducks, rabbits, chickens, pigs, etc.), and an area can be empty as well. We develop all kind of supportive functions to put animals into the backyard, query the number of animals of a given type, query the nearest animal of a given type, etc. (These are not detailed now.) The main storage unit will be field 't', a protected matrix:

```
class backyard : IEnumerable
{
  public animal[,] t = new animal[20, 20];
  /* ... supportive methods ... */

  public IEnumerator GetEnumerator()
  {
     var p = new backyardEnum(t,20,20);
     return p;
  }
}
```

If we make an instance from the backyard class, and want to process with *foreach*, the loop will call this GetEnumerator() function for the very first time to get the appropriate helper instance. For now we develop the helper class in a separate class. To the helper instance we pass the field 't' and the sizes as parameter (the size can be query from the matrix field, but this way it is more simple). Thus, the backyard instances can be processed by a *foreach*:

```
backyard u = new backyard();

// add aninmals to 'u'

// then ...

foreach (animal p in u)

{

   /* ... */

}
```

Let us create the helper class (of which the GetEnumerator() will instantiate):

· We will need a function 'Reset()', but it is only for the COM compatibility, it is not used by foreach.

· Need a MoveNext(), which is called by the foreach systematically to move on to the next item. The MoveNext() must return a boolean value. When it is true it indicates the there are elements waiting to be processed (we are not at the end yet). The foreach starts with calling this MoveNext(). If it returns a false it means there are no elements at all to process. In this case the foreach won't executes its body for the first time either.

· We need a Current property, through which the current element can be read (when the MoveNext() indicated with a 'true' value that an item is ready to be read).

```
class backyardEnum : IEnumerator

{

   protected animal[,] t;

   protected int N;

   protected int M;

   protected int i;

   protected int j;

   // ........

   public backyardEnum(animal[,] t, int rows, int columns)

   {

     this.t = t;

     this.N = rows;

     this.M = columns;

     // start position

     i = 0;

     j = -1;

   }

   // ........

   public void Reset()

   {

     throw new NotImplementedException();
```

```
    }
    // ........
    public bool MoveNext()
    {
      while (true)
      {
         j++;
         if (i >= N) return false;
         if (j >= M) { i++; j = -1; continue; }
         if (t[i, j] == null) continue;
         return true;
      }
    }
    // ........
    public object Current
    {
      get
      {
        return this.t[i, j];
      }
    }
}
```

The 'backyardEnum' constructor saves the received parameters to the instance-level fields (the matrix itself, and its dimensions), and initialize the 'i' and 'j' fields. These variables will be used as the current position of the process in the matrix. The 'i' presents the row number, the 'j' stands for the column number (x,y cords). The 'i' is set to 0, and 'j' is set to -1, because the first thing in MoveNext() is to increase 'j' by 1, so this case the MoveNext() will test the [0,0] cell first.

The Reset() method is required by the interface, but drops and exception (it is not used by the foreach, it does not matter this time).

The MoveNext() is called at the very beginning, and after processing an element every time. We assume that the last [i, j] is processed by the foreach, so the MoveNext() first step to go to the next element. Since the matrix is processed line by line, so first the value of 'j' is increased. If we exceeded the end of line to the right (the 2nd if) we increase the value of 'i' and set the value 'j' back to -1. If we reach the end of the matrix (the value of 'i' is too big, the 1st if), we have no more elements to be processed in the matrix. If we have good coordinates in 'i' and 'j', but the cell is empty (null, the 3rd of), then this cell is not suitable for us, so continue looking for another cell (containing instances of animals). If all the tests are done ('i' and 'j' is within the matrix, the cell is not empty), the MoveNext() successfully found an item to process (and returns true).

If the MoveNext() reported a success, the foreach will call the Current property to read the value. Since the MoveNext() set the value of 'i' and 'j', these coordinates point to the item itself to be processed, the only thing to do is to give back the value of this cell.

24. 24. Nullable type

The concept of the nullable types is that the value types (bool, int, double etc.) are not able to store a special value that indicates that "has not received any value." For example, the double can indicate that, there is a constant called Double.NaN (NaN = not a number), but there is not a such special value for a boolean type. However, if the boolean value is read from a database, it is frequent that the SQL bool (in MS-SQL it is 'bit') value is allowed to be null. Reading such a value into a bool variable is very risky, the SQL value should be examined before put into a boolean variable and the null value must be replaced with (for example) false. Unfortunately, it also covers the fact that in the SQL table it was not false but null (cannot be differentiate later from it was originally false or from it was null in the SQL record).

The nullable type differs from other value types with the ability to store the null value. Easy to use them in the source code: their "type names" consists of the name of the base (value) type name and a question mark behind it:

```
bool? b = false;

int? x = 0;

double? d = null;

char? c = '\x32';
```

The question mark is a meaningless modifier along with a reference type, as a reference type variable is capable to store a null value by its nature:

```
string s1 = null; // its ok
string? s2 = null; // syntax error
```

This is actually a syntax sugar of the compiler, as it is the alias name of the System.Nullable<T> generic class. So:

```
double? d
```

… is the short form of:

```
System.Nullable<double> d
```

… and because of "using System":

```
Nullable<double> d
```

also would be the same.

The nullable type can also be used in the boxing operation. This is a very smart way, because if the value of a nullable type variable is actually the 'null', then the boxing is not executed, the object type variable can take the value null in the straight. (As a reference typed variable, he has an inherent capacity to store null values):

```
bool? b = null;

object o = b;
```

In this case, take care of the unboxing, it's easy to produce a syntax error. An object value must read back into a nullable type:

```
bool? m = (bool?)o;
```

Here, along with the nullable types we must talk about a special operator of the C# language – the ?? operator! It is a binary operator, the first operand is a value which can be null. In terms of the behaviour of ?? if the first operand is null, then the final value of the expression is the 2nd value this time. However, if the first value is not null, the final value is equal to the 1st value (in this case it is not important what is the 2nd value is). They are very

useful when we want to put the value of a nullable type variable into a non-nullable variable. We will face problems here:

The 'x' cannot store the actual value of 'b' in general as this value (of 'b') can be null, and 'x' cannot store this kind of value. Therefore type cast is needed. In this case, the 'as' operator cannot be used as we have a value type here:

```
bool? b = false;

bool x = (bool)b; // runtime error?
```

However, if 'b' is actually null, then this type casting will cause a runtime error (exception). A good solution for test the value with an ‘if’ statement:

```
bool? b = false;

bool x;

if (b != null) x = (bool)b;

else x = false; // default ’false’
```

The same a bit shorter:

```
bool? b = false;

bool x = false; // default

if (b != null) x = (bool)b;
```

It can be also described by the ?: (ternary) operator:

```
bool? b = false;

bool x = b!=null ? (bool)b : false;
```

The simplest way is still the ?? operator:

```
bool? b = false;

bool x = b ?? false; // b or false
```

The memory allocation of a nullable type depends on the basic type, and an extra bool typed field is added, which indicates that the variable is null or not. So if a conventional double allocates 8 bytes, a 'double?' requires 8 + 1 bytes. However, since the 32-bit compilers usually align the variables to a memory address which is equal to some multiple of the memory word size (it is 4 byte for a 32 bit CPU). In fact, when a “double?” is allocated to the memory, 3 bytes of meaningless data is added to it before the next data is allocated. So in reality with a “double?” we might lose 12 bytes. Along with this a nullable type is still a value type.

25. 25. Exception handling

The exception handling is an extremely important topic; the way it is solved is strongly linked to the OOP world.

Almost from the beginning the computer program was built from functions. In the very beginning it was not a trivial idea, because the first programming languages are not familiar with the concept of a function. However, later this concept became a basic method of construction, so the syntax of programming languages imported

this concept, and raised this concept to the syntax level, they started supporting the creation of functions, we can name them, we can define parameters, return values, etc.

The functions are program blocks with separate task. Each function is responsible for their task, when we call a function we ask him to carry out that task successfully. If the task needs some additional data to understand exactly what is the task – we pass parameters to it.

But what happens if the function is unable to perform the requested task? It can be for many reasons, but the most common reason is the bad parameter values. In other words, a task that was asked, but with the actual parameter values cannot be performed. Let's examine two specific cases:

· The Sqrt(..) function is responsible for calculating the square root of the given number. The number can also be a fraction, so the type of the parameter is double. The Sqrt function is not able to compute the square root if the parameter value is a negative number.

· The FileCreate(...) function is responsible for creating files on the disk, where the name of the file is given as an argument. The file name is a string, may be a full path is included. But FileCreate cannot perform the job if the file name is wrong (a file with the given name cannot be created).

We must agree with the treatment of these cases. If a function is unable to perform the task, somehow must indicate it. Consider the following ways:

· the function writes an error message to the screen,

· plays a series of sounds, writes into a log file, or indicates the error in a similar way,

· the function set the error code into a global variable,

· the function returns with a special value.

Let us consider the cases in which one is to understand why it's not considered a good solution. In the first case, the function prints an error message:

```
static double Sqrt(double x)

{

    if (x < 0) Console.WriteLine("cannot be computed");

    // ... continue

}
```

Two problems arise in this approach. One is that the function return type is double, and the compiler expects a return value from us, regardless of what we write to the screen. What we give back in this case?

```
static double Sqrt(double x)

{

  if (x < 0)

  {

    Console.WriteLine("cannot be computed");

    return 0;

  }

  // ... continue

}
```

Think about - is 0 a really good return value? The code which called the function will it really know that trouble happened? Can we find a more appropriate return value? The -1 seems to be better for example, because the 0 is

the square root of 0 itself, we can't decide seeing the 0 as the return value that is indicates an error, or its simple the computed square root. The -1 cannot be the computed square root, so this value always indicates an error. Of course, in this case we can find an even better solution:

```
return Double.NaN;
```

The NaN (Not-A-Number) is a double constant, which indicates the case "there is no any number." The value is well able to indicate that the parameter has no square root at all (eg. because it is a negative number). Could this be the solution?

Notes: this solution expects the call side to test the returned value, like this:

```
Console.Write("enter a number:");

double x = double.Parse( Console.ReadLine() );

double b = Sqrt( x );

if (Double.IsNaN(b)) Console.WriteLine("has no square root");

else Console.WriteLine("sqrt={0}",b);
```

It seems to be a nice and appropriate solution, but we tend to turn a blind eye over a few things:

· when you executes the program in fact it is already written "cannot be computed" (by the function) before the "has no square root" would appear on the screen,

· since we wrote the function and the main program as well, it's easy to remember (for us) that the NaN value will indicate the problem,

· the function is not portable, it cannot be re-used in other programs.

When we write an error message to the screen, always think about what is our goal. The error message will help whom and how? If it is huge program which works with a great amount of data, which computes for minutes silently, then a message "cannot be calculated" appears, would it really help anything? Who will read this message? The user? Will it give him any useful information about what he should do in the next minutes? Will he understands the reasons? If he would pick up the phone and read the error message to the help desk (who gives the error message to the developer to correct the error), will it help the programmer find out what happened and where is the error? Finally: what language do we use in the error messages? Hungarian or English?

Larger projects with more conventional solution to create log file which contains the information achieving important milestones, the time, the current status of the program. When an error is raised, it contains exactly which function detected the error, what were the conditions (function call chain starting from the Main), the current values of parameters, and all the other information needed to reproduce the errors, might including machine type, operating system and anything else. A one-line error message obviously cannot contain so much information and therefore is useless in most cases.

Writing error messages in functions helps a little. We can help it: transform the error message by writing it into comment. Well, but what happens if the user code (the call side of the function) is significantly more complex than this. For example, the function is used in an expression?

```
double b = Math.PI * Sqrt( x ) / 2;
```

If we comment out the error message in the function, it will not turn out that something went wrong. As it evaluates the expression with the NaN inside, the final value will be NaN as well, which will be stored into 'b'. Assume that the expression is more complicated:

```
double b = Math.PI * ( Sqrt( x ) + Sqrt( y ) ) / 2;
```

Such an expression when we found a NaN in variable 'b', how will we find out what was the reason? Either (or both) function might have resulted a NaN, in such case the entire expression will be evaluated to NaN. Should we evaluate and test the function results separately before the evaluation of the expression would start?

```
double b;

double b1 = Sqrt(x);

if (Double.IsNaN(b1))

     Console.WriteLine("square root cannot be computed for x");

else

{

    double b2 = Sqrt(y);

    if (Double.IsNaN(b2))

        Console.WriteLine("square root cannot be computed for y");

    else b = Math.PI * ( b1+ b2) / 2;

}
```

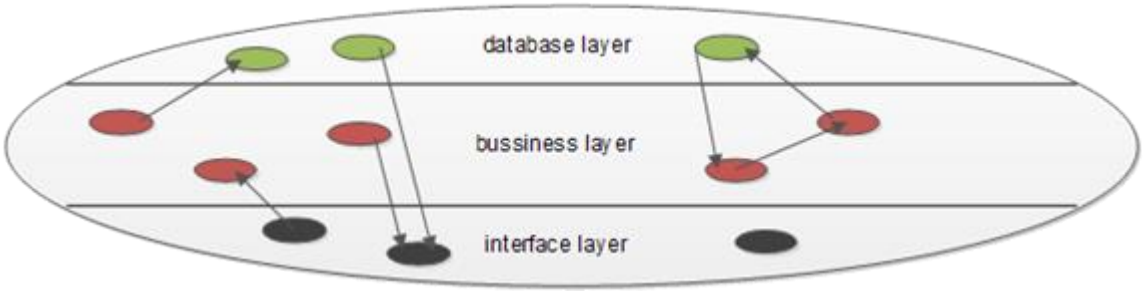
First: it becomes too complicated. Second: it is always another problem that the calling side should check whether a function was able to complete the task or not. This side usually forgets about it. No one develops a program by putting each function call into an 'if' statement. It complicates (especially when nesting) the calling code, but significantly slows down the execution time. However, if the calling code does not do so, then the problem can grow even bigger, as the evaluation process is not stopped, which will cause a fatal error, but many dozens of lines away. In this case, it is very difficult to trace back which was the critical point from which the error started out and led to this fatal event.

An error message written in the function of course might help a lot, because we at least can see on the screen that something has happened there. However, we have to understand: a professional developer – according to the principle of the three-tier application development model – never prints out an error message in a function body. According to this principle the parts of a computer program must be categorized as the following:

· **User interface layer**: (presentation logic) is responsible for the communication between the software and the user. This allows the user to control the program, allows him to start or stop processes, enter data, read the calculation results.

· **Application logic layer**: (business / application logic) responsible for the calculation and computations duties. In general, this is the program "core", contains the know-how, it is a very essential part. For example, for a CAD program in this part are encoded those functions which will check when the designed components together can form a functional structure, etc.

· **Data management layer**: (database) is capable of managing mass amount of data to be stored and retrieved, the data that the program is working. Nowadays SQL servers help lots, typically solved with them, but other solutions (eg. use of XML file) exists as well.



Our functions must be classified into layers. The correct way is if a function is clearly assignable to exactly one layer (according to the extension of the principle a whole object class must be classified into exactly one layer).

The square root function should be placed into the application logic layer as its main activity is a computation. Such a function cannot prints *anything* to the screen, not even an error message. Why? Because if we want to migrate this function into another environment (for example, in a Windows GUI program or in a web

application, or in a mobile application) the Console.WriteLine simply won't work there. We should better not do it a priori.

Principle: a function which calculates a result does not print the result or a message of failing. Which is a function prints something to the screen never calculates anything.

The three-layer architecture sometimes means physically three different programs. IN a web application the SQL database server is one of the participants, the application logic runs on the server (web server), while the program responsible for data visualization is the browser, which runs on the client machine.

This means that printing an error message within the Sqrt function violates the rule, it is not recommended to use.

We have problem with the return value still. We would have chosen the '-1', similar to the NaN. But even the Double.NegativeInfinity value as well (and, of course, the -2, or even -1.5 would have been chosen, etc). If we write the function and the calling side code, then it does not matter. If different developers works on the two parts, the other programmer how to figure out what is the return value when special error happens? And when the user code is finished, but we modify our function body (make another decision) and choose another return value, how will he know that he must change the calling side error handling method?

Summarize the problems:

· according to the three-layer model it is not recommended to print error messages in the functions,

· such error messages would be useless in larger projects, cannot be understood,

· a function must return a value when an error happens as well, must choose an extreme value in these cases,

· often we can choose from several extreme values, in which case the calling code may be uncertain which of them was chosen,

· latter modifications lot of damage can be done, when the calling code does not follow then changes,

· the calling code should always check whether the extreme error value was returned or not, otherwise critical (fatal) error may be caused, but in a distant point of code from the real reason of the problem, so it is difficult to trace back the problem,

· these checking slows down and complicates the caller-side execution.

Many developers handle these problems by preventing them. Then *before* they call the function, check whether it can be executed (computed) or not:

```
Console.Write("enter a number:");

double x = double.Parse( Console.ReadLine() );

if (x<0) Console.WriteLine("square root cannot be computed for x"); else

{

   int b = Math.PI * Sqrt( x ) / 2;

   /* ... */

}
```

We might wonder if in the body of the function we can skip the testing of the parameters. Apparently not: we cannot trust the outside world supposing that the number passed by is suitable to calculate its square roots is verified properly and correctly. We can feel a problem here, the parameter value is double-checked (in the outside and in the inside of the function body as well). The double-check affects to the execution speed of the code. Also raises another problem: it is certain that the outside world is able to check properly and correctly the parameter values? Consider another example, the function to create a file. What can be the reasons for failing the creation of a file:

- file with the same name already exists,
- the file name has an invalid structure, for example: contains invalid characters,
- the file name is too long,
- the directory where the file would be created, does not exist (or either its parent directory),
- there is no such drive in which the file should be created,
- the drive exists but is read-only (e.g. CD-ROM),
- the drive is full, it cannot create one more file (e.g. on a USB stick),
- the drive is a network drive, and we do not own the ‘create’ right on it,
- it is a network drive, but apparently is not available.

Could we be sure that at each point of code all the reasons are checked in the program where we want to create a file using this function? Considering that all the options will be tested by the function body as well? Obviously not.

In summary,

- The function body must check all the parameters of the task to be performed, as it is best suitable to check them all, and cannot trust in the outside world.
- The outside world do not try to take over the role of the parameter checking.
- The outside world does not check the parameters, this will mean double-checking and will slow down the execution.

As stated above, we can summarize:

- the function must perform the checking,
- if a problem is detected, indicate it to the calling code,
- the problem should not indicated to the user, not in a form of an error message.

The last one is the most important: if the function was not able to do its job, it is not the user who must know about it, who knows nothing about the structure of the program, what kind of functions are there, what is the call chain. The function must indicate the error to the calling side code, which can make a decision about to continue or abort its (higher level) task.

All must be done by not requiring the calling code to write ‘if’-s around the function calls every time to check if all is in order. We like to be optimistic: things are generally in order. But we still want to know if something went wrong, something exceptional has happened.

25.1. 25.1. Throwing an exception

Suppose, we develop the Sqrt function, the ‘if’ is added, we checks the parameter value, and we see that it is wrong, the square root cannot be calculated. We cannot print an error message, we do not want to return any value (neither an extreme value), we are worried about the calling side, and we do not want to make him believe everything is all right. Then what should we do?

A mechanism will helps us. We can imagine: the program at the start is in a normal state. Then what we know so far happens: instructions are executed in the given order, we call functions, they return with values, the expressions are evaluated, and everything goes normally. But there is another state that a program can have: the error state. If we go to error state, the instructions of the normal state are not executed. The iterations must end immediately, the selections in the program are skipped, the evaluation of expressions aborts. The call chain goes back toward the Main function by returning from the functions back without any return values. If we do not switch back to normal state, then this continues in the Main function as well, skipping the normal state

instructions inside Main as well, and the Main will terminates as well. This means the program itself will terminate (finish).

So if we force the program to be in error state, it will easily cause the program (not instant, but fast) to terminate. How to switch to the error state? With the help of the **throw** statement.

The throw statement itself may be presented alone (without parameters) in some circumstances (which we will discussed later). When it has a parameter, it is an object instance. This instance stores in its instance fields all the error information that is passed back to the calling side code to help understanding the exact reasons of failure. In a very simple case it is an error code (integer), in a more complex case a great number of fields might store the required information.

In the C# language a rule describes what types of classes aresuitable for this task. In other languages there is no such rule. In C# the classes which are type compatible with the Exception class are allowed to be the parameters of a throw. Which classes are compatible with? The Exception class itself, and its child classes. There are lots, in fact, and we can also develop our own ones. A simple (but not the simplest) example:

```
static double Sqrt(double x)

{

    if (x < 0)

    {

        Exception e = new Exception();

        throw e;

    }

    // ... continue the calculation

    return double.NaN;

}
```

The Exception class has three public constructors. The first needs no parameter, will only indicate an error without describing the reasons. The second constructor requires a string: we can specify an error message (as in the case of printing an error message). The third can receive a nested exception instance, but this time we do not want to discuss it. Most often we use the 2nd form with the string parameter:

```
Exception e = new Exception("Square root cannot be calculated");

throw e;
```

We do not meet throwing an exception like this too much in codes. Think about the fate of instance 'e'. There is a saying: there is no life in the function after the ‘return’. Well, there is no life in the program after the ‘throw’. The 'throw' will switch the state of the program to error state; we cannot use the 'e' later (for the moment leave it at that). The error indication above consists of two-steps, it is considered to be useful only after the instantiation step we refined the error description using instance ‘e’:

```
static double Sqrt(double x)

{

  if (x < 0)

  {

    Exception e = new Exception("Square root cannot be calculated");

    e.Data.Add("parameter x=",x);

    e.Source = "static double Sqrt( x ) func";

    throw e;
```

```
  }

  // ... continue the calculation

  return double.NaN;

}
```

Without doing this, we use a simpler form: the instantiation and switching to error state is formed into one single step:

```
throw new Exception("Square root cannot be calculated");
```

In this case the newly created instance won't be saved to any variable 'e'. The program goes to error state and the instance describing the error is attached.

Note that in this selection branch the function does not require a return statement anymore, we do not have to choose between returning back -1 or NaN. The function call essentially returns the Exception instance this case. We can imagine as the throw will executes (substitutes) the return immediately as well. Let's see what happens at the calling side. The following code will not work:

```
double b = Sqrt( x );

if (Double.IsNaN(b)) Console.WriteLine("no sqrt value");

else Console.WriteLine("sqrt={0}",b);
```

If the square root cannot be computed, and the Sqrt function with the help of throw switches to error state, then the execution of 'double b = ...' statement won't be finished (normal state instruction), and the 'if' in the next line will be skipped as well. Dare to write the following:

```
double b = Sqrt( x );

Console.WriteLine("sqrt={0}",b);
```

The WriteLine will be executed only if the Sqrt does not indicate any error – haven't switched to error state. Otherwise, this statement is ignored. Similarly, we can peacefully insert the calculation into an expression:

```
double b = Math.PI * ( Sqrt( x ) + Sqrt( y ) ) / 2;

Console.WriteLine("final value={0}",b);
```

If any Sqrt call fails, the program will not execute the printings, as it is a normal-state instruction - **and for this the caller does not have to use any if**!

The second half of the sentence is important because this is what we usually do - forget about it. We do not like writing if-s, often superfluous, un-optimized, sometimes we make mistakes coding the complex condition expressions. There is a lot of trouble with the *if* statement. Using the throw is not needed, because when there is trouble, the execution of the program no longer continues, nor in the calling side as well. In fact, the calling side we can use the optimistic coding attitude: code as everything is fine, no error presents. If no error occurs in fact, then the code is actually executed as fast as possible. However, if an error occurs at any point of the code, the statements on the caller-side are skipped automatically because of the error state of the program.

25.2. 25.2. The reason of the error

The question is: in this way how can we indicate the reason of the error to the caller side? How will they identify the actual reason of the failure of the FileCreate() call?

The problem can be approached in several ways, but none of them will be proposed later. The very 1st idea is the using of the error message text:

```
throw new Exception("invalid char in the filename ");

throw new Exception("directory does not exist");
```

```
throw new Exception("network drive error");
```

If 'e' is the instance attached to the throw, and somehow we can examine it, we can use ‘if’-s to check the reason. In the instance the field ‘Message’ stores the error message passed to the constructor at throwing:

```
if (e.Message=="network drive error") ....;
```

Obviously, this method is not appropriate. The reasons:

· if we want to change the error message text, it must be done in two places: at the throwing of the exception, and here in the ‘if’ conditional expression (eg. we want to reformulate the text, or use a different language),

· comparing strings is a slow operation,

· the calling side code must know the exact text of the error message at the moment of development.

Moreover, we can use the filed ‘Data’ in which we can store (add) any amount of data, information. But this is also not the general way. Instead, we usually classify the reason by the type (class) which was instantiate to describe the error, and was raised by the throw statement, this is the recommended method.

As we already know not only the instances of the Exception class can be used to raise an exception (throw), but the child classes also. The child classes represent the reason of the problem, the error. According to the error (exception) handlings the Exception class the represents the same as for the type compatibility the Object class is. Currently, it is considered the root element. The .NET Framework predefined error types (but not limited to) are the following:

Exception

ApplicationException

SystemException

ArgumentException

ArgumentNullException

ArgumentOutOfRangeException

FormatException

IOException

DriveNotFoundExceptionFileNotFoundExceptionPathTooLongExceptionPipeException

NotImplementedException

NullReferenceException

OutOfMemoryException

PrintSystemException

PrintCommitAttributesExceptionPrintingNotSupportedExceptionPrintJobExceptionPrintQueueExceptionPrintServerException

StackOverflowException

The above list is indented, which represents the inheritance tree. In other words, for example, the 'ArgumentNullException' is the child class of the 'ArgumentException', whose ancestor is the 'SystemException', whose ancestor is the 'Exception' class itself.

It is not a rule but a naming convention that the end of the exception classes are '…Exception". The. NET Framework contains dozens of exception classes, and additional ones can be defined. When we want to indicate

an error by throwing an exception, we need to select an appropriate object class, instantiate it, and attach to the statement ‘throw’:

```
throw new ArgumentNullException("1st parameter is null");

throw new PathTooLongException("file name was too long for OS");

throw new OutOfMemoryException("4Gb must be enough for everything");
```

25.3. 25.3. Handling the error

If our program is switched to error state by executing a throw statement, it will skip all the statements of the normal state, reaching the end of the function block it will go back (returns) to the call side, and this will continue until it reaches the Main function, the initial function call in the call chain. Because of the error state it will skip the instructions in the Main as well, and by reaching the end of the Main function the program stops (the run-time error will terminate the program).

This process can be stopped anytime using the **try … catch** statements. The **try** has its own block of code, inside an exception can be thrown (inside the block or inside a function called from the block in any depth). When an exception is thrown, the **catch** statement will block the error state, and after the execution of the catch block instructions, the program returns to normal state. Draft:

```
try

{

   // normal state

   // instructions

   // which might

   // throw

   // an exception

}

catch

{

   // error state

   // instructions

   // execute when

   // exception is thrown

}

// instructions here

// will be executed at any cases

// as the catch would switch back

// to normal state
```

We will refine this draft later, but basically this is the case. The inside of **try** we write the "optimistic" code, we hope that it will run and finish correctly, or at least most of the cases, and we do not want add several ‘if’-s to check. If no error happens, then it would act as the **catch** block would be empty, not even a single instruction is executed inside it. However, if an error rises (exception is thrown) inside the **try** block, the execution of the instructions inside the try block is interrupted, and the catch block is executed. Inside the catch is plan "B", which would either try to repair, correct the reason of the error, or otherwise, choose another way to finish the task, or ultimately inform the user of the failure.

The function calls can be placed inside the try block, which to the principle of the three-layered development cannot communicate with the user, thus the errors are indicated by an exception throw. The calling side can catch the exception threw, and it might belong to another layer, it can display the error message to the user:

```
try

{

   Console.Write("enter a number:");

   double x = double.Parse(Console.ReadLine());

   double b = Math.PI * Sqrt(x) / 2;

}

catch

{

   Console.Write("an error occured during the process");

}

double c = b*2;
```

Think about what might happen in this try block:

· Console.Write essentially can't cause an error,

· the double.Parse can raise errors when we enters a number on the keyboard which cannot be interpreted as a double value,

· the function call Sqrt()can also raise an error when the "x" value is bad, it's square root cannot be calculated.

Both the Parse() and the Sqrt() functions belongs to the (middle) application logic layer, so they cannot communicate to the user, they can only throw and exception to indicate a problem. If there was no problem at all, the input is successful, the square root exists, and the value of 'b' can be calculated. If an error is raised (during the parsing calculating the square root) the error message above is printed to the screen.

The code is not entirely correct, because the variable 'b' is visible only inside the try block, and cannot be referenced outside, after the catch block. Otherwise, the code is not logical, because when an error occurs, the 'b' value is not given, so the 'c' is not calculable. Try to correct the code:

```
double b;

try

{

    Console.Write("enter a number:");

    double x = double.Parse(Console.ReadLine());

    b = Math.PI * Sqrt(x) / 2;

}

catch

{

    Console.Write("an error occured during the process");

}

double c = b*2;
```

Now, the declaration of 'b' is moved before the try block, so its scope extends after the catch as well. But the code still contains an error: when an error occurred inside the try block, then ‘b’ is undefined, and the value of ‘c’ is still not calculable (cannot be known what value must be multiply by 2). Correct this code once more:

```
double b;

try

{

   Console.Write("enter a number:");

   double x = double.Parse(Console.ReadLine());

   b = Math.PI * Sqrt(x) / 2;

}

catch

{

   Console.Write("an error occured during the process");

   b = 0;

}

double c = b*2;
```

Into the catch blocks a new statement is added, it assigns to the variable 'b' a default value of 0 in the absence of any other idea. (This is a plan "B".) Then the value of 'c' is calculable, because the 'b' is defined, has a value in any case, if everything is good it is the square root of the number we entered, in other cases it is 0.

Using only this knowledge the following programming problem can be solved: *enter 5 integer numbers, and add them to a list. The program must take care of entering invalid numbers, when a user inputs an invalid integer number: the program must alert an error message.*

```
List<int> l = new List<int>();

while (l.Count < 5)

{

  try

  {

     Console.Write("enter an integer number:");

     int x = int.Parse(Console.ReadLine());

     l.Add(x);

  }

  catch

  {

     Console.Write("invalid number, again");

  }

}
```

The 'l.Add (x)' will only be executed if the Parse(..) does not indicate an error. If it does, the error message is displayed and in this iteration no element is added to the list 'l'. The iteration will execute until 'l' expands to

five-elements, so until 5 numbers will be entered successful. If we would not put the entering a number into a try block, then:

```
List<int> l = new List<int>();

while (l.Count < 5)

{

   Console.Write("enter an integer number:");

   int x = int.Parse(Console.ReadLine());

   l.Add(x);

}
```

In this case as we enter an invalid number, the Parse(...) drops an exceptions, the program switches to an error state, and in the absence of catch, it quits from the while, and finally from the Main() function itself, the fatal runtime error terminates the program.

Let us return to the original problem with the Sqrt function. We discussed that either the Parse() or the Sqrt() can drop an error in the try block. How do we know which one dropped the exception?

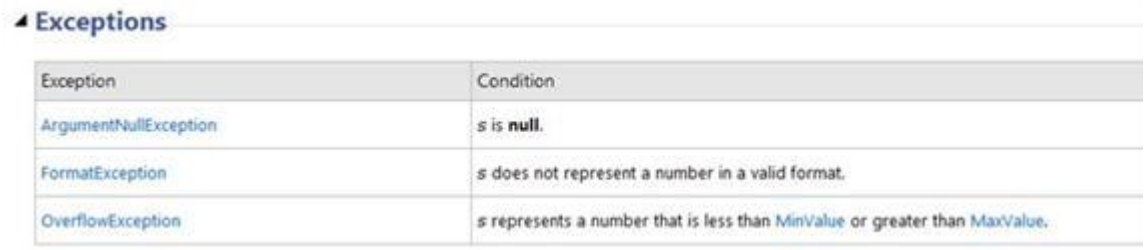
```
double b;

try

{

   Console.Write("enter an integer number:");

   try

   {

     double x = double.Parse(Console.ReadLine());

   }

   catch

   {

     Console.Write("The parse threw and error");

   }

   b = Math.PI * Sqrt( x ) / 2;

}

catch

{

    Console.Write("Sqrt threw the exception");

    b = 0;

}

double c = b*2;
```

Let's analyze the solution (except that the declaration of 'x' inside the try block is bad, its scope prevents us from passing it to the Sqrt(..)as a parameter). Embedding try blocks usually enables us to separate the problems. But if the both problems sources are in the same expression this method won't work of course:

```
double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt( x ) / 2;
```

25.4. 25.4. Finding out the reason of the error

How to identify which part dropped the exception in the expression? Well, it's time to use that information that for different types of error a different type of exception is dropped. The Sqrt() indicates its problem by instantiating the basic Exception class and throws it. Easy to find out the Parse() drops a FormatException instance if the string does not contain a recognizable number.



Exceptions

Exception	Condition
ArgumentNullException	*s* is **null**.
FormatException	*s* does not represent a number in a valid format.
OverflowException	*s* represents a number that is less than MinValue or greater than MaxValue.

If we are interested not only if it is thrown, but the specific reason of the error (described by the exception instance attached), a parameter to the *catch* keyword must be added:

```
try

{

    // instructions

}

catch (Exception e)

{

    // handling the exception

    // the error instance is ”e”

}
```

The catch parameter is similar to the function parameters. If we want to refer to the body inside the catch to the thrown exception instance, it a name must be specified, for example 'e' is a good name. We must define its type (Exception). In this case the ‘e’ will receive the parameter value passed by not a conventional function call but this parameter value is passed by the ‘throw’ statement itself.

```
try

{

   Exception p = new Exception("problem is found");

   throw p;

}

catch (Exception e) // e = p

{

   // handling the exception

}
```

What values can be stored in variable 'e'? The type compatible rule will answer it. As the static type 'e' is Exception, so any memory address of an instance which is compatible with the Exception type (Exception class and child classes). Since rule to the parameter of the *throw* statement is the same, this means all kind of thrown exception instance can be stored in ‘e’.

```
try
{
  double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
  if (e is FormatException) Console.WriteLine("Parse error");
  else if (e is Exception) Console.WriteLine("Sqrt error");
}
```

The 'is' operator can be used to test the dynamic type of the 'e' instance, so we can determine which part of the expression is responsible for the exception. Note that “e is Exception” test really does not make sense, because it surely evaluates to 'true'.

This way is working - but raises serious questions. What happens when we know what to do when the Parse drops the error, but we do not know what to do when Sqrt fails? We want to ‘catch’ the Parse error but we want to let the Sqrt exception goes away. If we catch an exception both cases, it is too late - the program is switched back to normal state, and the exception is handled.

25.5. 25.5. An exception is re-thrown

An easy way to solve the problem above – throw back the exception. The same instance which was caught:

```
try
{
  double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
  if (e is FormatException) Console.WriteLine("Parse error");
  else throw e; // exception re-throw
}
```

The instance caught in 'e' can be re-throw. It is legal, even inside a catch block exceptions can be generated. Since the catch has been cleared the error state of the program, it is legal to switch back to error state again, with re-throwing the same exception instance.

This can be done in an easier way: the parameter less throw statement:

```
try
{
  double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
  if (e is FormatException) Console.WriteLine("Parse error");
```

```
  else throw; // re-throw the same error instance
}
```

The parameter less throw can be inserted only into a catch block, and then it will re-throw the same exception instance which was caught by the catch. When we drop the original exception in the function body (for the 1st time), a parameter is always required, and as switching to error state an error description must be attached.

Naturally we have the right not to throw back to the original exception, but create a new one and throw this new one instead:

```
try
{
  double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (Exception e) // e = p
{
   if (e is FormatException)
        throw new Exception("Parse error occured");
   else throw;  // re-throw the original e back
}
```

25.6. 25.6. Classifying the exceptions

If the problem is that we caught an exception unintentionally, and because of this it must be re-thrown - it actually means the code is bad. Indeed, to catch an exception and re-throw the same is a total waste of time, a lot of time. A better approach to not catch the one we are not interesting in. A better code:

```
try
{
  double b = Math.PI * double.Parse(Console.ReadLine()) * Sqrt(x) / 2;
}
catch (FormatException e) // e = p
{
  Console.WriteLine("Parse error");
}
```

A skilful parameterization of the catch we can catch only a specific subtype of errors. This branch of catch will react only to FormatException (or a child class of it) instances. The Sqrt function simply throws Exception instances, so according to the type compatibility rule an instance of Exception cannot be assigned to a variable 'e' above. For this kind of error instances simply is not caught, so it is not needed to re-throw them. In addition, the checking with ‘if’ and ‘is’ is ignored as well, which is guaranteed by the type name in the parameterization the catch.

There can be more than one catch branch along with a single try block – when they catch different types in their parameterization:

```
try
{
```

```
  // the code
}
catch (FormatException e)
{
  /* … */
}
catch (NullReferenceException e)
{
  /* … */
}
catch (InsufficientMemoryException e)
{
  /* … */
}
```

This case is similar in structure as a switch, that is, these catch blocks are conditional branches. It can be understood with the semantics of the 'is' operator:

```
catch (FormatException e)
     // executes when the exception is compatible with the FormatException
catch (NullReferenceException e)
     // ... when compatible with NullreferenceException
catch (InsufficientMemoryException e)
     // ... when compatible with InsufficientMemoryExcption
```

It is also similar to *switch* that only one branch can be executed at a time. The conditions evaluate in the given order, so the order of the branches are important. Here is a (bad) example:

```
try
{
  // the code
}
catch (Exception e)
{
  /* … */
}
catch (FormatException e)
{
  /* … */
}
```

If a FormatException is thrown in the try block, the 1st condition evaluates to true, as it is compatible with the Exception class. Therefore, the 2nd branch will never be selected and executed. The correct order:

```
try

{

  // the code

}

catch (FormatException e)

{

  /* … */

}

catch (Exception e)

{

  /* … */

}
```

That is, the exceptions branches with the ancestor classes (types) should be added later, in order not to catch the child class exceptions. If we have Exception branch, it should be added to the very end of the branches.

25.7. 25.7. User-defined exception classes

As we have seen, for the exception handling the most important is the type of the exception, helps separating the actual reason of the failure. The. NET Framework contains a great number of pre-defined exception classes, which are useful to describe the reason of the error.

However, if we need a very special class which has fields and methods to store and retrieve our (very detailed) description of the circumstances of the error, we can define a custom exception class for it. In this class we can create our own constructors, the parameterization is chosen to clarify the circumstances of the error.

Defining a custom exception class means exactly the same as defining any other (simple) classes. Since we can throw an instance of a class when it is compatible with the Exception root class, the only special rule is to select an existing exception class to be the ancestor class. This secures that our custom class becomes type compatible with the Exception class.

We might create a custom exception class for other reasons, for example we simple want to create a new one to help to select it easily in the catch branches. In this case we add no fields and methods to the ancestor class, just creates a child class. Of course we need to write several constructors to call the base class constructors and give the parameters back to it.

```
class NullErrorExcep : NullReferenceException

{

   public NullErrorExcep(string msg) : base(msg)

   {

   }

   public NullErrorExcep() : base()

   {

   }

}
```

This is the simplest case when we create a custom exception class with two constructors, whose parameterization is similar to the base class constructors. The parameters are passed through only; they are passed back to the ancestor class. For this reason this child class is no more than a (poor) copy of the base class. However, it counts as a different type, so it might be used to define a private catch-branch:

```
try

{

   // the code might drop an exception

}

catch (NullErrorExcep e)

{

   // throw new NullErrorExcep("...") has been happened

}

catch (Exception e)

{

   // other exception is thrown

}
```

Completely analogously, we can create a custom exception class, which really extends a class with new fields and methods:

```
class ConvertErrorException : FormatException

{

  public string val; // which cannot be converted

  public ConvertErrorException(string msg, string val) : base(msg)

  {

    this.val = val;

  }

  public ConvertErrorException(string val): base("parse error")

  {

    this.val = val;

  }

  public ConvertErrorException(): base()

  {

    this.val = null;

  }

  public void writeToLog()

  {

    string msg = String.Format("parse error happened, "+

                    "original value ={0}",this.val);
```

```
    log.messageWrite(msg);

  }

}
```

25.8. 25.8. Finally

The use of the finally block is most often associated with resources. Resources can be anything, which exists in limited amount, so they may run out, and can be possessed them by fight. It could be coal, oil, people, whatever. Related to IT resources are memory, network connections, printers, database connection, etc.

To work with a resource the very 1st thing to do is to acquire an access to it. This is often not easy to obtain, we must wait it for to get free and then get the right to use it. The primary owner is usually the operating system itself, we must ask a permission from it. Then we try to use for the shortest possible period of time, and give back the right as fast as we can, to let the others to acquire it as well.

Therefore the steps are:

1. requesting and obtaining the resource,
2. use of resource,
3. give back the permission (release, set free the resource).

The task difficulty is given that after successfully acquired the resource, it definitely should be released at the end - even if we got an exception (error), and even if everything went right. However, if an exception occurs (and we do not want to handle it), it must be returned to the calling side to inform it about the failing.

Let's see an example (draft):

```
printer p = OS.gimmeThePrinter();

try

{

  // printing with "p"

  // an exception might be dropped

}

catch (Exception e)

{

  // do nothing but ...

}

p.release();

throw e; // re-throw the exception ’e’
```

There are several errors in this draft solution:

· the catch block is essentially empty, as it does not want to deal with the problem, just to store the exception to variable 'e',

· however, this variable ‘e’ is out of scope to the catch block, not available at the end where we try to throw back,

· when there was no problem during the printing (with 'p'), no exception at all, we have nothing to throw back.

Refine the draft solution:

```
printer p = OS.gimmeThePrinter();

try

{

  // printing with "p"

  // an exception might be dropped

}

catch (Exception e)

{

   p.release();

   throw e; // re-throw the exception 'e'

}

p.release();
```

Now all of the above problems have been solved, but the release of the resource appears twice in the code. Once inside the catch block before re-throwing the exception, and appears after the catch block. It is required as if there was no error, the catch block is not selected, and in this case the resource must be released as well.

If releasing the resource is not so simple, not a single function call, but it is a more complex process; to be included twice is problematic. This is where the 'finally' construction appears:

```
printer p = OS.gimmeThePrinter();

try

{

  // printing with "p"

  // an exception might be dropped

}

finally

{

   p.release();

}
```

The *finally* also appears in pair with the *try* keyword. This finally block - regardless of whether an exception occurred inside the try or not - is executed at the end. This means that the release of 'p' happens in any case, if we enter into the try block. However, on the contrary of *catch* the *finally* never handles the exception that occurred, we can imagine this as it is re-thrown automatically at the end of the finally block. So when an exception occurs a finally …

```
finally

{

  p.release();

}
```

… is the same as …

```
catch (Exception e)
{
  p.release();
  throw e; // re-throw the exception 'e'
}
```

A frequent error when the resource allocation is added to the *try* block. However, this is a mistake: the allocation of the resource can throw an exception and it because it is inside the *try* block, the *finally* will be executed, including the resource release instruction. But because the resource has been allocated, it is possible that releasing it generates another exception. Since we try not to handle the original exception, but the new one will override it, the original one will disappear However, if the resource allocation is not in the *try* block (instead it is outside), and an exception is dropped, the *finally* won't be executed, and the original exceptional goes back to the callers side immediately. Then the (unnecessary) release of the resource is not executed.

25.9. 25.9. An exception is dropped during the exception handling

The question is whose solution is worth finding: what happens when an exceptions is dropped in the try ... finally ... pair, inside the try block, then the finally starts, and inside the finally another exception is thrown?

```
try
{
  try
  {
    try     { throw new NullReferenceException("nullref");  }
    finally { throw new IndexOutOfRangeException("indexoutof"); }
  }
  catch (Exception e)
  {
    Console.WriteLine(e.Message);
  }
}
catch (Exception f)
{
  Console.WriteLine(f.Message);
}
```

This code prints the 'indexoutof' word to the screen. However in the 1st time the 'nullref' exception is dropped. This leads to the finally block, where the 'indexoutof' exception is raised. At the end of finally, the exception must be re-thrown (if there was any). But it seems that the new exception will replace (override) the old 'nullref' one.

This example reveals many concepts. Only one exception can be active at a time. When we drops another instance, the new one hides the old one, overwrites it. Therefore, if an exception occurs in finally, this new one will be the active one. A similar behaviour can be observed when the innermost try-finally block is replaced to a try-catch pair, and inside the catch another exception is thrown. This is a simpler problem, since the catch always removes the active exception. Then more interesting question is, if we have multiple catch branches and

an earlier catch catches the exception, but re-throw a new one, the other catch branches can handle this new one?

```
try
{
  try
  {
    throw new IndexOutOfRangeException("indexout");
  }
  catch (IndexOutOfRangeException)
  {
    throw new OutOfMemoryException("outofmem");
  }
  catch (Exception e)
  {
    Console.WriteLine("inner {0}",e.Message);
  }
}
catch (Exception f)
{
 Console.WriteLine("outer {0}", f.Message);
}
```

We might think that the 'IndexOutOfRange' catch block catches the exceptions dropped in the try block, then the exceptions dropped by inside this catch block can be handled by the very next 'Exception' branch, as its type is suitable for handling the IndexOutOfMemory exceptions. However, if we start the program, it prints out the "outer" message, so the out of memory was not handled by the Exception branch. Conclude the experience: a try with a (more) catch-branch acts like a switch: no more than one branch can be selected. If the selected branch drops another exception, it cannot be handled by the additional branches; an outer try-catch is required.

25.10. 25.10. Try … catch … finally …

A triple combination, the *try* - *catch* – *finally* structure:

```
try
{
  // block #1
}
catch (IndexOutOfRangeException)
{
  // block #2
}
```

```
catch (Exception)

{

  // block #3

}

finally

{

  // block #4

}
```

In this scenario (remember: multiple catch branches can exist) it is easy to understand according to the rules discussed previously. The *catch* branches - if they are suitable for - can catch and destroy the exceptions dropped in the *try* block. In any case the *finally* is executed, even if there was or was not an error in the *try* block (and a catch branch handled it or did not handle). In fact, the *finally* is executed even if an exception occurs inside the *try*, one of the *catch* has handled it, but re-throw it (or a dropped another exception).

The *try ... finally ... catch...* structure does not exist.

25.11. 25.11. Nesting

The try ... catch ... finally .... structures can be nested to any depth. The behaviour is predictable, if we do not forget the following rules:

· the *catch* can catch the exceptions that is dropped inside the corresponding *try* block,

· when an exception is dropped somewhere, but not in the corresponding *try* – the *catch* won't do anything (do not interested in those exceptions),

· the finally will be executed in any case, if there was or was not an exception (but also only when the exception was dropped in the corresponding *try* block)

· if an exception occurred outside (before) the *try*, the finally will not be executed.

```
// |

// | block #0

// |

try

{

  // |

  // | block #1

  // |

  try

  {

    // |

    // | block #2

    // |

  }
```

```
  catch (IndexOutOfRangeException)
  {
    // |
    // | block #3
    // |
  }
  catch (Exception)
  {
    // |
    // | block #4
    // |
  }
  finally
  {
    // |
    // | block #5
    // |
  }
  // |
  // | block #6
  // |
}
catch (Exception f)
{
  // |
  // | block #7
  // |
}
// |
// | block #8
// |
```

1. What happens if an exception is dropped in block #0?

The rest of the instructions inside block #0 is not executed, and since this block is not in a *try*, no other blocks executes, the function will return to the calling side in error state with the exception.

2. What happens if the block #1 drops an exception?

The remaining instruction of block #1 are skipped, one-block instructions are not executed, the corresponding catch block is the block #7, it will handle the error, the program switches back to normal state, so the program continues with block #8. If another exception is raised inside block #7, the program remains in error state, and block #8 won't be executed, the functions returns to the calling side in error state with the exception instance raised in the block #7.

3. What happens if an exception in the block #2 is generated?

If this exception was IndexOutOfRange, the block #3, otherwise block #4 will be executed. After the corresponding catch branch the finally block is executed at block #5. The exception is handled, so the program remains normal state, so it continues with block #6 and block #8.

4. What happens when the block #6 generates an exception?

Block #6 is inside a *try*, so the associated catch block #7is selected. The catch handles the exception, switches back to normal state, so it can continue with block #8.

5. When is block #8 to be executed?

If there was no exception at all, anywhere, or there was (not inside block #0) but one of the catches handled it.

Similar questions could be formulated; the answers can be given by applying the rules described earlier. With small test programs the answer is easy to check, with some throw and Console.WriteLine in it.

26. 26. Operators

The operators as basic programming elements and are required. With their help, we can formulate expressions to calculate new values from the existing ones. In C# language, like any other high-level programming language, a number of operators are known (eg. addition, subtraction, equality tests, etc.). They are parts of the basic language, these operators are interpreted only on given types. The meaning (semantics) of an operator depends on the type of data elements to apply on. For example, the addition operator is defined for numbers, it is a numerical operator. Between strings it is the mark of concatenation. Between characters is also a numeric operator (calculates the summary of the Unicode character codes).

In procedural languages, there were some complex data types, record, array, or list. The essence of object-oriented programming, however, is to move beyond this. The types we create are complex data types with its own operations. However our custom types are not able to move on at a given point: if we want to perform some operations on them methods and functions must be used. Operators cannot be applied to custom-typed data, since the compiler does not know how to generate code for a given case.

However, operators and functions have a much closer relationship than we might think at first:

```
int a = 12;

int b = 23;

int c = a + b;
```

To add variables 'a' and 'b' can be described with not only operators but also methods:

```
int c = add( a, b );
```

Thus, the addition operator (and any other operator) can be substituted by a suitable (developed to the appropriate types) method call:

```
static int add(int a, int b)

{

  return ...;

}
```

```
static string add(string a, string b)

{

  return ...;

}


static int add(char a, char b)

{

  return ...;

}
```

When we use + operator in the code, the compiler (might) simply replaces it with an appropriate function call. If a function does not exists with the given parameterization, then there is no such operator between the given data type values, so it is a syntax (compiler) error. If the compiler's internal rules defines only those interpretation of the + operator, it can replace a use of + operator to only those function calls presented above, and we try to add two boolean values with the + operator, the compiler cannot turn it into any function call[30], so a syntax error is raised.

If you want to apply operators to our custom types, we would need to understand the relationship between operators and functions. We will write functions with specific naming syntax, and therefore the compiler will recognize this function calls in a form of operators as well. In addition, many rules must be known, which will be discussed in this chapter.

The C# language compiler recognizes three types of operators:

- unary operator, e.g. the operator ++,
- binary operator, e.g. addition,
- type casting operators.

Out operator functions must match one of the categories given above. Each category has their own special rules.

In other programming languages it is also possible to expand the scope of operators to custom types. In some languages it is possible to develop brand new operator marks (introduce new operators) as a combination of marks (characters) like ~, !, +, -, *, /, %. . In other languages only by the existing operators can be re-interpreted. Furthermore, some of the languages the associativity and priority can also be re-defined, while in other languages (especially where only existing operators can be redefined) these properties cannot be changed.

The C# language is a strict language at these possibilities. Only existing operators can be expanded, but not all of them. The priority and associability of the operators cannot be changed.

The operator functions are methods, because the C# language is a pure OOP language, writing functions are still not available only within a class. There are rules where to develop a new operator function, which class must contain these functions.

26.1. 26.1. To develop unary operators

The most important unary operators of the C# language are:

+, -, !, ~, ++, --.

When ‘X’ is a unary operator, and we use it on a value of type ‘T’, the compiler checks whether there is an 'operator X (T)' function in the declaration of type T, or in one of its a base classes.

Assume that we have a custom 'myDate' type, which can store a date value. We made a decision that we choose a date value to be a start date; our choice is the 01/01/1970 date. We store a date as a difference to this date: how

many days have passed since this date. So a date like 10/02/1970 is stored as 41, indicating that our date, February 10 is the 41th day counting from 01/01/1970.

This storage method has some advantages and disadvantages. The advantage is to quickly select which is the earlier or later date of the two, it can be easily calculated how many days are there between two dates. The disadvantage is that a given value (e.g. day 4519) it is not easy (although not impossible) to tell exactly represents which day, month and year. With the latter problem for the sake of simplicity, we will not deal with.

```
class myDate

{

  protected int _days;

  public int days

  {

       get { return this._days; }

  }

  public myDate(int n)

  {

     this._days = n;

  }

   public myDate(string sDate)

  {

    // somehow we calculates that

    // the number of days to sDate

    // and store this value

    this._days = ...;

  }

}
```

We would like to apply an operator to myDate values. For example, the ++ operator has sense this case; means "go to the next day's date".

```
myDate d = new myDate("2012.01.20");

d++;
```

To let the compiler recognize and understand expression 'd++' and to know what code must be generate there, we need to create a 'myDate' interpretation of '++' operator function:

```
class myDate

{

 public static myDate operator ++(myDate x)

 {

   return new myDate( x.days + 1 );

 }

 // ... and continues ...
```

From now when we write somewhere in our code in the 'd++' expression (where the type of 'd' is 'myDate'), the compiler replaces it to 'myDate.operator++(d)' function call:

```
// what is written:

d++;

// what happens:

d = myDate.operator++(d);
```

So the operator function is called (depending on the type of the value), it produces a new value which will overwrite the previous value of 'd'. Note, the compiler does not know anything about that this operator is increase or not this value at all, as in this circumstances it cannot be verified.

Fix the rules that are needed to write a custom unary operator:

· When we want to apply a unary operator on type ‘T’, this function must be placed into the same class ‘T‘, ie. if the type of the value is myDate used by the operator, we must develop this function into 'class myDate’.

· The unary operator functions must be ‘public’ and ‘static’.

· This function can have only one parameter (since it is a unary operator function).

· The type of the parameter must match the type of the class containing it (ie. the first rule again: an operator function with T parameter type must be placed into class T).

· The function return type must be the same type (a unary operator cannot change the type of the value).

The general form of a unary operator is:

public static <type> operator <operator>(<type> <pname>)

{

...

}

Discuss the function body in the implementation of the operator! The task is to receive a date, and the next (higher) value must be given as a return value. Basically it can be achieved in two ways:

```
public static myDate operator ++(myDate x)

{

   return new myDate( x.days + 1);

}
```

or:

```
public static myDate operator ++(myDate x)

{

   x._days++;

   return x;

}
```

In the first case we create a new myDate instance and return it. In the second case we modify the parameter instance and return the same instance was passed to us. What are the differences between the two implementations?

Do not forget that the 'class' types are reference types. That actually means a memory address was passed to the function, and it is also what the function returns - a memory address. Consider what happens with the code below according to the different implementations of the operator function:

```
myDate d = new myDate("1970.01.10");

myDate p = d;

d++;

Console.Write(p.days);
```

For the first version of the function: we make an instance 'd', and the 'days' field is set to 10 as 10 days is passed from the starting date. The variable 'd' stores the memory address. Then we make a copy of this memory address to variable ‘p’, so the days field of 'p' is also 10. Increase the 'd' value by 1 with the ++ operator. During this the operator function creates a new instance, and returns its memory address. The variable 'd' gets this new value. The date represented by variable 'p' (which still holds the old memory address) is unchanged, still 1970-01-10, while the ‘d’ is 1970-01-11.

In the second implementation this will not happen. When we pass ‘d’ to the function as a parameter, it will increase the days of 'd' by 1. This solution does not create a new instance, but the implements the change on the given instance. Thus, at the end of the function he memory address remains unchanged in variable ‘d’, although the date was increased by 1. This is quite good, but also the date value of 'p' increased by 1, even though we have not requested it.

The lesson here is that both implementation works. It is up to us to choose between them. If we can accept the side effects of the last implementation, we can select this one. However when we work with classes, typically the first implementation is chosen because the user (other developer) may not understand why the value has changed in another variable, when no operator was applied to it.

26.2. 26.2. To develop binary operators

Not only unary, but also binary operators can be developed. These are:

+ - * / % & | ^ << >> == != > < >= <=

The binary operators can be applied on two (may be different) type values. Therefore, the operator function must have exactly two parameters. Using the previous example we create the possibility to add a myDate and an integer:

```
myDate d = new myDate ("1970.01.10");

myDate p = d + 10;
```

Adding a myDate and an integer we mean to increase (or decrease) the number of days, so the result is a new myDate value.

```
class myDate

{

  public static myDate operator +(myDate d, int n)

  {

    return new myDate(d.days + n);

  }
```

In the operator function the (first) parameter type is 'myDate', the second parameter is an 'int'. Its result is a 'myDate' value, so the expression above is correct and can be calculated.

The rules to develop a custom binary (two-parameters) operator function:

· if the two types of the parameters are T1 and T2, the method must be places into class T1 or T2 (ie. “myDate + int” function can be placed even into ‘int’ or ‘myDate’ classes).

· the binary operator functions are ‘public’ and ‘static‘,

· the function must have exactly two parameters (since it is a binary operator)

· the function's return type does not have to match any of the types of T1 or T2 (eg. the logical comparison operators return with boolean, while their parameter types are not bool).

The general form of the binary operator:

public static <type3> operator <operator>(

<type1> <pname1>, <type2> <pname2>)

{

...

}

As an example, write the comparison operators for two myDates:

```
public static bool operator <(myDate a, myDate b)

{

  return a.days < b.days;

}
```

When we want to compile this code, an error message appears. In fact, the compiler will recognize that from now we can write such a condition test:

```
myDate d = new myDate("1970.01.10");

myDate p = new myDate("1970.01.12");

if ( p < d ) ...
```

But we cannot write the following:

```
if ( d > p ) ...
```

So it can be tested whether 'p' is less than ‘d’, but cannot be tested that ‘d’ is greater than 'p'! Therefore ’*the operator < requires a matching operator > to also be defined*’ error message appears to warn us to develop not only the < but the > operator as well:

```
public static bool operator >(myDate a, myDate b)

{

  return a.days > b.days;

}
```

Similar pairs are the == and != operators and the <= and >= operators as well. We cannot develop only one of the pairs alone, the other one must be developed as well. By overriding the == and != operators additionally must consider the overwriting of the GetHashCode() and Equals() methods!

```
public static bool operator ==(myDate a, myDate b)

{

  return a.nap == b.nap;
```

```
}

public static bool operator !=(myDate a, myDate b)

{

  return a.nap != b.nap;

}
```

Another question arises: we have defined the 'myDate + int' operator, but what if we write this addition in the reverse order?

```
myDate d = new myDate("1970.01.10");

myDate p = d + 10;

myDate k = 10 + d;
```

Interestingly, although in mathematics we have learned that the + operator is commutative (the order of the two operands is unimportant, it does not affect to the final result), but the C# compiler ignores it. The 'int + myDate' is not accepted, indicating that an operator function with this kind of parameterization is not exists!

For the last example to the binary operators we examine how to apply the – operator to two myDates. It means how many days are between the two dates:

```
public static int operator -(myDate a, myDate b)

{

  return a.nap - b.nap;

}
```

26.3. 26.3. Type casting operators

The third type of operator does not look like operators at the first sight. The standard example is the type casting from double to int:

```
int a = 10;

double b = a; // implicit

int c = (int)b; // explicit
```

In the assignment 'b = a' the right-hand value (int type) is converted to a double value run time to be stored in ‘b’. This is an automatic process, which is called implicit type conversion (type casting). In the third statement is the 'c = (int) b' where the right value (double) must be explicitly converted to int. These steps are done by implicit and explicit type conversion operators.

Such conversion operators can easily be developed to our own types:

```
public static explicit operator string(myDate d)

{

  return String.Format( ... ); // átgondolandó

}
```

This operator supports the string ð myDate explicit conversion. This time we can write such code:

```
myDate d = new myDate("2012.12.02");

string f = (string)d;
```

Instead of 'explicit' keyword we could use 'implicit', which means in the example code the explicit type cast is no longer needed, simply 'string f = d;' could have been written.

After writing the following code the constructor with the string parameter is no longer needed, we can create myDate value from a string directly:

```
class myDate

{

  public static explicit operator myDate(string s)

  {

    return new myDate(s);

  }
```

So since then there is an explicit string ð myDate conversion, the next code works fine:

```
myDate d = (myDate)"2012.12.02";
```

To write an implicit type conversion operators - though essentially the same as the explicit conversion - always a decision of design. With implicit operators it is easy to accidentally write the wrong codes, which raises no syntax error to warn us:

```
public static implicit operator int(myDate d)

{

  return d.days;

}
```

If there is an implicit conversion from myDate ð int, it is easy to convert a date to integer:

```
myDate d = (myDate)"2012.12.02";

int n = d; // implicit myDate -> int
```

Let’s see the rules:

· The conversion operators accept only one parameter and its return type is always a different one.

· If the parameter type is ‘A’ and the return type is ‘B’, then the conversion operators can be placed either in the A or B class.

· The type conversion operator functions are ‘public’ and ‘static’.

· The function can be either 'explicit' or 'implicit', depending on the type of the type casting we want to develop.

The general form of the conversion operator is:

public static <implicit|explicit> operator <type2>(<type1> <pname>)

{

...

}

26.4. 26.4. Final problems

We present two problems. In the C# language, the test of strings equality can be solved by operators, meanwhile to test which string is the smaller, less-equal and other tests are not. This test can be interpreted in between

strings (the “smaller” is the string which is "forward" in alphabetical order), the phone book is sorted alphabetically, the computers can use the Unicode table, where each character has a numeric code (id) which is suitable to determine the right order. However, the test of string values cannot be written by operators:

```
string a = "apple";

string b = "peach";

if ( a < b ) ...
```

Unfortunately, the string comparison operators are not developed by the Redmond boys. Instead, they offer a static Compare method of the String class, which returns with an int value indicating the relationship of A and B:

+1 (positive), when A > B,

0, when A == B,

−1 (negative), when A < B.

Based on this function the operators could have been implemented easily:

```
public static bool operator <(string a, string b)

{

 return String.Compare(a, b) < 0; // a<b if it is negative

}
```

Supposing we understand the secrets of the operator overriding, we would like to write the missing 6-8 operator functions. Unfortunately, we cannot, according to the rules. Because these operator functions must be placed into the 'string' class itself, but its source code is not available. Here's where the story ends (though we offers some hope in the extension methods section, but just for this cases (to operator functions) the method cannot be applied).

The second problem may arise when we have plenty of conversion operators. Previously, we wrote the myDate ð int implicit operator, and the myDate + int addition operator. Consider the following code:

```
myDate d = new myDate(12);

myDate b = d + 10;         //  expression #1

int c = d + 10;            //  expression #2
```

Expression #1 is ambiguous, but we have bigger problems with #2. In #1 the 'd+10' may also mean that applying the additive operator between myDate and int, and the new myDate value can be stored to ‘b’. But it may also mean that the 'd' is implicitly converted to int, then add them together. In this case it would end, as the int cannot be converted implicitly to myDate as this operator is not developed.

In #2 it cannot be decided when 'myDate + int' is the one which is applied first, and then it is converted back to ‘int’ again implicitly – or the myDate is converted to ‘int’ first, then the two ‘int’ areadded, and this result is stored to ‘c’.

This example also reveals that the implicit conversion operators more often causes problems than solve them. If there were only explicit conversion from myDate ð ‘int’, then the expressions above would not be ambiguous, and the second way could be forced with an explicit type conversion:

```
myDate d = new myDate(12);

myDate b = d + 10; // myDate + int

myDate c = (myDate)((int)d + 10); // int + int, converted back

int e = (int)d + 10; // int + int
```

26.5. 26.5. Extensible methods

The extensible methods give us solutions to a special problem. A case where we do not want to override an existing methods in a class, but we want to add a new method to it. Through inheritance it is simple and natural way to add new methods, but the C# language (from version 3.0) provides us an alternative way.

To add a method to an existing class later (from outside), without modifying the original source code of the class: define these new methods into a ‘static’ class. The static class is a class that contains only static methods and fields, so we cannot instantiate it. We cannot add instance-level constructors to static classes, and the compiler won’t add default constructor as well.

The extensible methods means writing static methods, but these methods becomes - oddly – instance level methods. Supporting this the very first argument of this kind of methods are always the instance, so the type of this parameter matches the type of the instance. This type defines the class itself which is extended by this ‘instance level’ method.

The static class must be added into a namespace, and cannot be nested into another class definition, and cannot be generic either. Let’s suppose that there is a circle class, which has fields to store the coordinates of the centre of the circle, the radius, and there are methods to calculate the perimeter and the area of the circle. Complete the circle class with a method to return a Boolean value which specifies whether the center point lays on one of the axis (x or y):

```
namespace additions

{

  public static class anAddition

  {

    public static bool onAxis(this circle k)

    {

       if (k.x == 0 || k.y == 0) return true;

       else return false;

    }

  }

}
```

The interesting thing in this code is that it is a static method, the first parameter has a type, but before the type name the ‘this’ keyword is given. To use (call) this method is only possible if the namespace is opened by the ‘using’ keyword:

```
using additions;
```

After this step the code completion shows this method in the suggestion list, and the compiler accepts this method call as well:

```
circle k = new circle();

if ( k.onAxis() ) ...
```

The methods can have additional parameters, which must be included after this very first special parameter:

```
namespace additions

{

  public static class anAddition

  {
```

```
    public static void move(this circle k, int x, int y)
    {
      k.x += x;
      k.y += y;
    }
```

Usage:

```
circle k = new circle();
// ...
k.move( 10, 30);
```

Unfortunately, with the help of extensible methods neither properties nor operator functions cannot be added **later**.

27. 27. Assemblies

Thinking of larger projects, it is necessary that the code base must be separated into smaller parts, to be distributed. There are several reasons for this, for example more than one developer are working on the project, all of them want to separate their parts away from the others, separately develop, compile and test. Another reason may be that the task itself suggests splitting into parts. General principle of the development of the three layered (multitier, three tier architecture) application model in which the activities of the program are classified into three different layers, and these layers can be developed and tested separately. It was discussed above in chapter 25 “Exception handling”. Summarizing shortly:

· **User interface layer**: (presentation logic) is responsible for the interactivity between the software and the user, handling of the input devices (keyboard, mouse), printing out the messages, etc.

· **Application logic layer**: (business / application logic) responsible for the calculation and computations tasks. It contains the core computations which makes the software characteristic (if it is a bookkeeper software, the calculation of the year bonus is a function to this layer).

· **Data management layer**: (database) stores and retrieves the data of the program to a 3rd party service (SQL server, web services).

In addition, it comes up that certain function tasks can be re-used in different applications, so it should generally be written in a way to support this activity, to be easy and comfortable to re-use them when a new project is started, and add them to the project. If this code is modified later (bug fixes, optimization, new functions), it also would mean a lot if these modification (the new version) can be added quickly and easily to the projects.

Emerging problem is that in some cases the job is not to write a complete application, but rather a well-defined part of. For example, create a code that can encoding / decoding video files, encrypting, compressing data with a very high speed etc. Sometimes these sub-tasks require professional skills and deep knowledge in the area and the final product (a handful of functions) separate itself is a product. In this case the source code cannot be published, this would be unfortunate, because the know-how is often more expensive than the license. How do we sell our finished product to developers to use these functions in their projects without giving the source code to them? How can we write a part of code to be re-used and embedded it into other applications easily?

The problems are not yet in close contact with the OPP, as they are as old as the assembly-level (second-generation) programming languages, and have been arisen related to it as well. Therefore, first we will examine the issue and possible solutions in other languages, before the OOP world.

When the programmer writes the source code, the first step is that the compiler will read it and try to understand[31] (interpret). The result of its task is the object code, which is written in machine-level code[32] version containing the instructions. This object code cannot be started yet. The case is similar to when a writer has written a book, his hands are full of papers, but it is not the book. The final form of the application is made

by the linker program, which are often left out because of his activities while are important, but not spectacular. (Unlike the compiler, who frequently rejects the source code to compile because of the syntax errors.)

The relationships between the object codes (calling a function in one object code meanwhile the function is compiled and presented in another object code) is checked and resolved by the linker. Therefore, this is called **static linking**.

The object code produced by the compiler is interesting because it contains the original code written by us, but not in the source code form. The source code is beautiful, contains expressive identifiers (variables, functions), comments, is well-wrapped, high-level control structures are used (iterations, selections). The object file contains no comments, the identifier names we used are replaced by the compiler into meaningless ones (which are often simple numbers), and our nice control structures are translated into low level instructions (using jump-like statements). The object code therefore in a certain point of view is a more secure version of the original source code. Reading and understanding it requires a lot of time and energy to a foreigner, the theft of know-how thus greatly troubled.

On the bases of the object code the original source code, the know-how can be rebuild, possible, but difficult. The process is called reverse engineering. In doing so, the reasons (why we solved the problems in this way), the circumstances (design, testing) are rarely known, but the technical details of the solution are discovered. The reverse engineering is prohibited by law in most countries, and the perpetrator risks a very serious penalty.

In addition, while the original developer owns the original source code, the thief has only a version with a much lower quality. Other modifications, enhancements, bug fixes can be done by the original developer much easier, than by the thief.

If it is not a goal to make it more difficult to steal the source code (it is used only "in the house"), still also useful to do it in object code form. Since you do not have to copy the source code into the projects source code (insert it into a folder, where many similarly named source files are, or copy-paste physically it into another source code), because in this case when we compile the project, the compiler must read and understand it and generate the object code again-and-again. Instead, it is enough if the linker meets the object code, and adds it to the final executable. When the source code is modified, we simple recompile it and copy the new version into the relevant projects.

However, there are some problems with the static linking method:

· If the functions are changed, we must rebuild (recompile), copy it to the relevant projects, but the projects itself must be re-compiled and refreshed to the execution environment (on the user computers).

· It is difficult to track which projects are the relevant ones, do not miss one, it must be replaced everywhere.

· The executable programs become large in size, as the linker merges the object files into a big executable (.exe) file.

There is a method for **dynamic linking**. In doing so, the executable file physically will contain all the machine code versions of the functions. The executable file (.exe) is therefore essentially cannot be started alone. However, the linker generates "packages", which contains the missing codes. We can imagine, as if the full, final, all-inclusive standalone executable file is split into smaller parts, like a picture is broken into puzzle pieces. The packages together are ready to start, but separately to the others none of the parts are viable.

If we split a complete exe file into pieces, there will be exactly one piece which will contain the Main function, and there will be pieces without the Main function. The special piece is named and saved to the disk by the linker as .EXE, because it must be started by the user, but the other pieces are named differently (with another file name extension). The other parts are saved to disk on the Windows platform as .DLL (stands for dynamically linked library[33]).

The assembly concept therefore indicates that a train contains a locomotive (.exe) and coaches (.dll) and they together define the whole train. When the linker meets a unit, it must recognize its type (.dll or .exe). If it is a DLL, it is not considered an error if there is no Main function (typically there is not). However, in the executable code (. exe) an application entry point must be filled, so it must contain the Main function.

27.1. 27.1. The DLL-s in Windows

The DLL technology is supported on Windows OS platforms from in the first version. Even the Windows operating system consists of a number of DLL files which together gives us the functionality of the operating system. The functions stored in the DLL-s of the OS can be called not only from another DLL of the OS but from any user application. The DLL s are not safe from this point.

The linker might reckon an executable (.exe) as good when it contains not all the function codes required to be started. But in this case the linker must know:

- what is the name of the DLL which stores missing function,
- what is the identifier of the function inside that DLL[34].

This information will be saved by the editor into the executable program. When the operating system starts such a program, it reads the required external DLL names from it and verifies that they are available and whether they actually contain the given function identifier. If these DLLs refer to further DLL-s, they will be checked as well. If one is missing - it is not possible to start the program, the operating system refuses it.

If we have a DLL, how can we determine what are the names of functions stored inside? If we are the one who developed this DLL, then we will know the names of the functions, their parameters and their tasks. If it was not created by us, it is necessary to get some kind of documentation. In C, C++ language itself a big help is that in these languages a "header" files must be created, which contains the prototypes of the functions (name, parameterization, return type). These header files are not created for documentation, but they are required by the compiler as a normal and necessary part of the development process. So these header files are ready, always up to date, and contains all the necessary information required to another programmer to call the functions from this DLL.

If no documentation is available, the situation is not hopeless, because the DLL file itself also contains the list of function identifiers. This list inside the DLL is necessary, required for the OS, as it checks this list at the program starts (described above). This list can be recovered by simple utility programs, although this activity is very close to the reverse engineering boundaries.

Note, however, that the DLL stores the list of functions, but does not store any information about the parameterization (number and types of function parameters). These are stored in the C language header file, but in the absence of it only the names can be revealed. To get the known parameterization we must read and understand the function body as well, but it is really a kind of reverse engineering and therefore is usually illegal.

On Windows OS to use a foreign DLL without documentation is not easy task at all, and often cost more time and energy than rewriting the desired functions.

About the DLL-s of the Windows OS, the function names, parameterization, tasks are documented by Microsoft. This documentation is named as Windows API (WinAPI, Windows Application Programming Interface). It has several versions, as the various Windows OS are different and there are 32-bit and 64-bit editions. The Windows '95 OS WinAPI describes than 9,000 functions, about 29,000 constants and about 4800 type definitions[35].

Note: in the Linux operating system the DLL-s are named "shared object", therefore its extension usually is ".so". In this platform a Windows program cannot be started for a number of reasons, including the fact that on Linux the Windows OS DLLs are not presented. In the Linux world there is a project "WinE", the aim of this project is to create the required DLL-s to a Windows program be launched on Linux.

27.2. 27.2. The DLL hell

Using DLLs many problems can be solved. A DLL is a standalone self-descriptive compilation unit. It is presented on disk as an independent separated file. If we modify a function of a DLL, we can re-compile the DLL, and simply replace the old file with the new one on the disk. If our application runs on different computers for different clients, it is enough to send the new DLL to them to be replaced. If the program will automatically update itself it is enough to download a small file and overwrite the old one on the disk, which is good for the

network resources as well. The DLL is still a compiled binary code, so for a DLL the know-how is “encoded”, to decode it is difficult.

They may be the case when on a PC more than one application of the same company are installed. Each of the applications (or more) uses the same DLL of the company (common functions). When a new version is made from this DLL, we must replace all the DLL instances on the disks for all the applications.

Of course it is possible to store this DLL into a separate shared folder, where all the applications can find it and use it. In this case we have only one DLL instance on the disk, so upgrading means replacing one DLL file. Along this idea it can be solved that an application uses a DLL of another application. This is common when the application developers agree with each other, provide the required documentation to each other, and so group of applications are being developed which can cooperate with each other so its functionality is higher than their rivals.

Therefore became typical the sharing of DLL-s between the applications. The shared folder is typically at C:\Program Files\Common Files or C:\Windows\System32 directory. When an application is installed, they might copy their shared DLLs into these folders. It was unnecessary when the application recognized each other on the disk, was able to discover the folder names of the other application, and load the foreign DLL directly from the other application install folder. Supporting this activity on Widows the central INI files (win.ini) was used, later the more robust version - the Windows Registry could be used.

The DLL Hell phenomenon known as the practical problems arising from these activities:

· An application copies a DLL into the shared folder, and overwrites an DLL with the same name installed by another application. As a result, the previously installed application cannot run properly and stops. The problem is difficult to identify, as the installation process is usually unknown by the user and cannot see a direct relation between the two events (a new application is installed, old one is failure).

· During an applications removal (uninstall) it can delete a shared DLL, which also means other applications will behave wrong, which would still need this DLL.

· When removing (uninstall) an application, it leaves the shared DLL on the disk. Then it is difficult to determine which DLL belongs to which application, when it is safe to delete from the disk.

· An update of an application might update a shared DLL. It can cause failures in other applications which uses these shared DLLs, they might become incompatible with the new DLL.

· As a way to solve this we can try to store all the versions of a shared DLL, the previous ones and new ones as well. This is problematic, because the operating system does not handle the version number information of a DLL during a program start, only the name of the DLL is required. So the different versions still must have the same physical file name, which makes the version management extremely difficult to handle.

27.3. 27.3. The DLL-s in .NET Framework

We can create DLL-s in the .NET Framework (C# language). A .NET DLL has the same extension on the disk, and has the same role for the application development, but there are fundamental differences.

First, although the file name extension is DLL but the inner structure is totally different. The most significant differences are:

· A .NET DLL is not a machine code version, but CIL code (common intermediate language, also known as MSIL), which is a virtual machine code version.

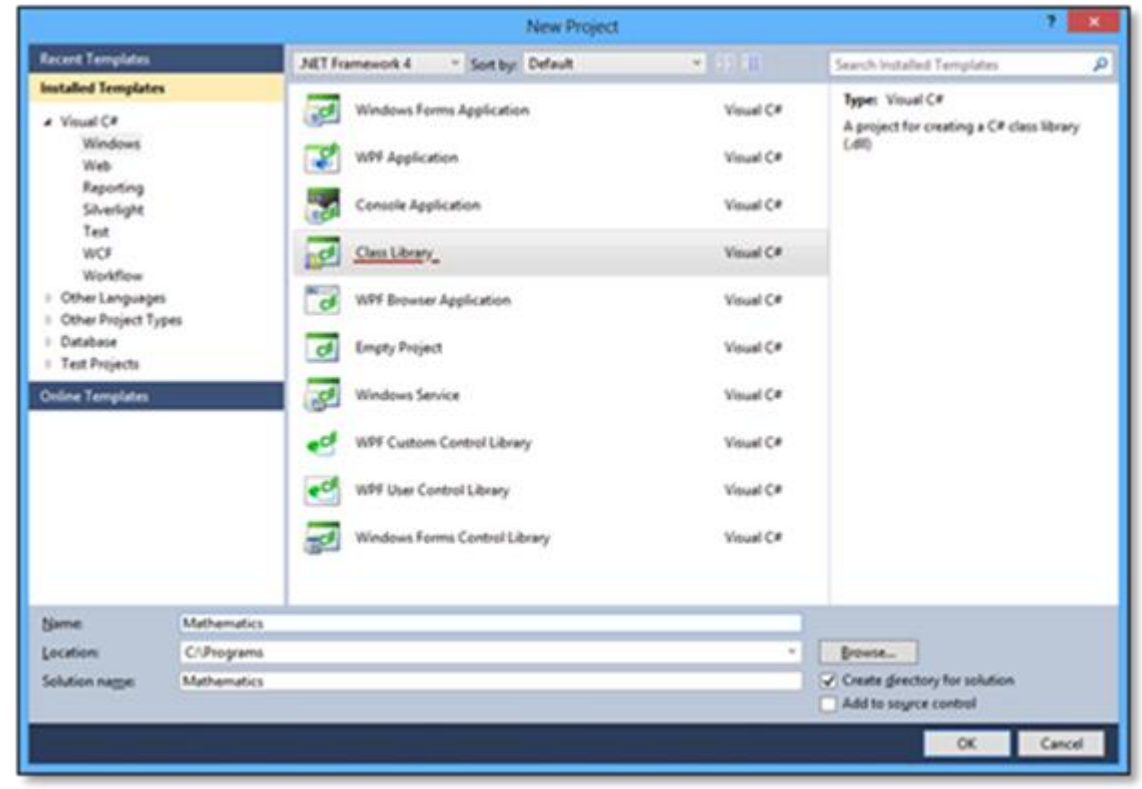
· they contain not simply functions, but completely supports the OOP environment, ie. complete classes, the level of protection, and all other things are supported,

· The company's digital signature, the version number is part of the DLL, so identifying the DLL such information can be used as well (optional),

· loading a DLL is not done by the OS, but the Framework, which knows another DLL sharing mechanism called GAC (see below).

27.4. 27.4. Creating a .NET DLL

When we want to create a DLL in the Visual Studio, we must choose the “Class Library” project type (File / New Project / Class Library)



In the case of a Class Library not the usual empty source code appears, it contains no Main function, so it is a much simpler one:

```
namespace Mathematcs

{

    public class Class1

    {

    }

}
```

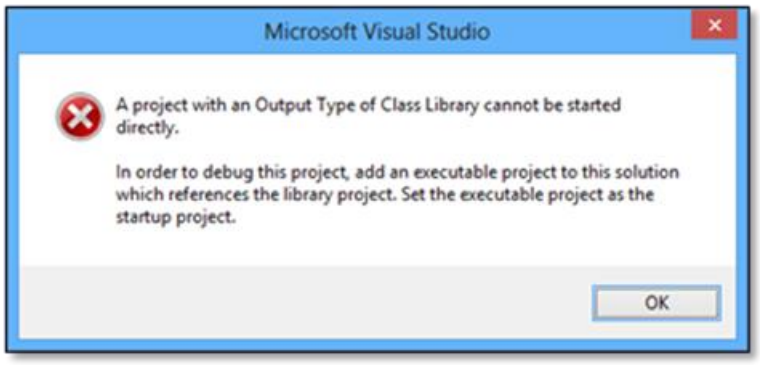
The namespace contains only one class marked with a 'public' modifier, and there is no Main function. This is normal, because the DLL does not contain the main program. The ‘public’ modifier before the class keyword will be important later.

The first step is to write a simple function. The function receives a string parameter, which describes an integer number (for example it has been read from keyboard by the Console.ReadLine() function). We try to parse it, and if it success, the function returns with this value. If this fails, then the second parameter as the default value is returned:

```
public static int IParse(string val, int defVal)

{

    try

    {
```

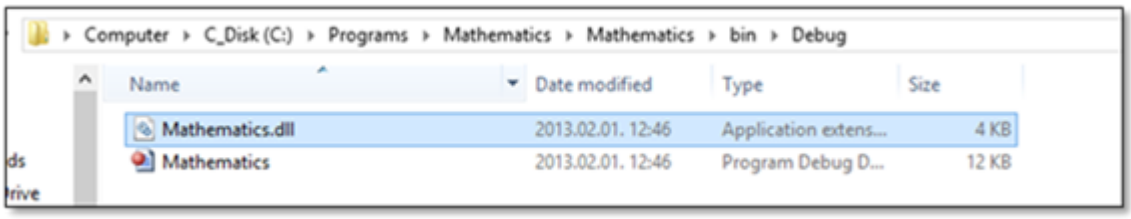
```
        return Int32.Parse(val);

    }

    catch { }

    return defVal;

}
```

This is our function; we intend to use it in several applications. Let's compile the DLL. Clicking on the usual green arrow icon (program start icon) drops an error dialog. It is a DLL not an application, without a Main function it cannot be started: "*the project with an output type of class library cannot be started directly*":



For this reason to compile a DLL the *Build/Build Solution* or *Build/Rebuild Solution* menu is used. These activities mean only compilation and code generation, but do not start it. However, syntax errors are checked and displayed.

After compilation, if we look at the directory structure the normal *Bin/Debug* directory now will contain not an .exe but a .dll file instead:



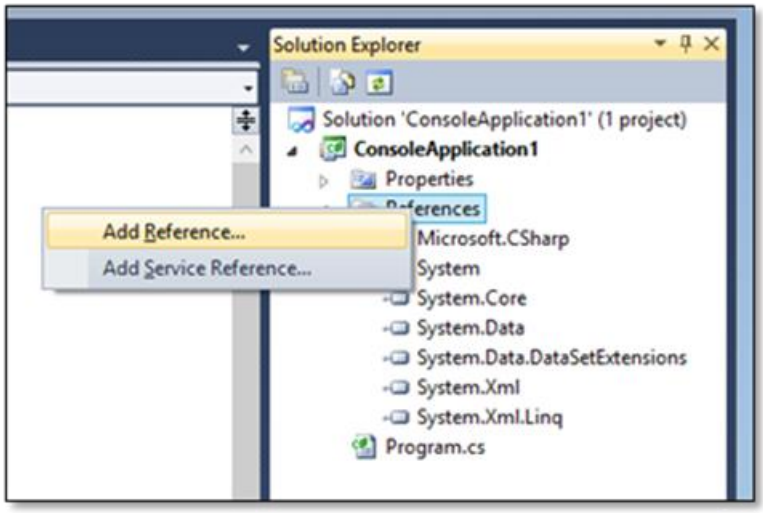
We will need another project, for example a console application that contains the Main function. Here we can choose to start another VS, or just close this project and starts another one, or make the Solution Explorer to create another project into the tree into this solution. For simplicity, select the first option, starts another Visual Studio to create a console application to call and test this function:

```
using Mathematics;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            string s = Console.ReadLine();
            int x = Class1.IParse(s, -1);
        }
    }
}
```

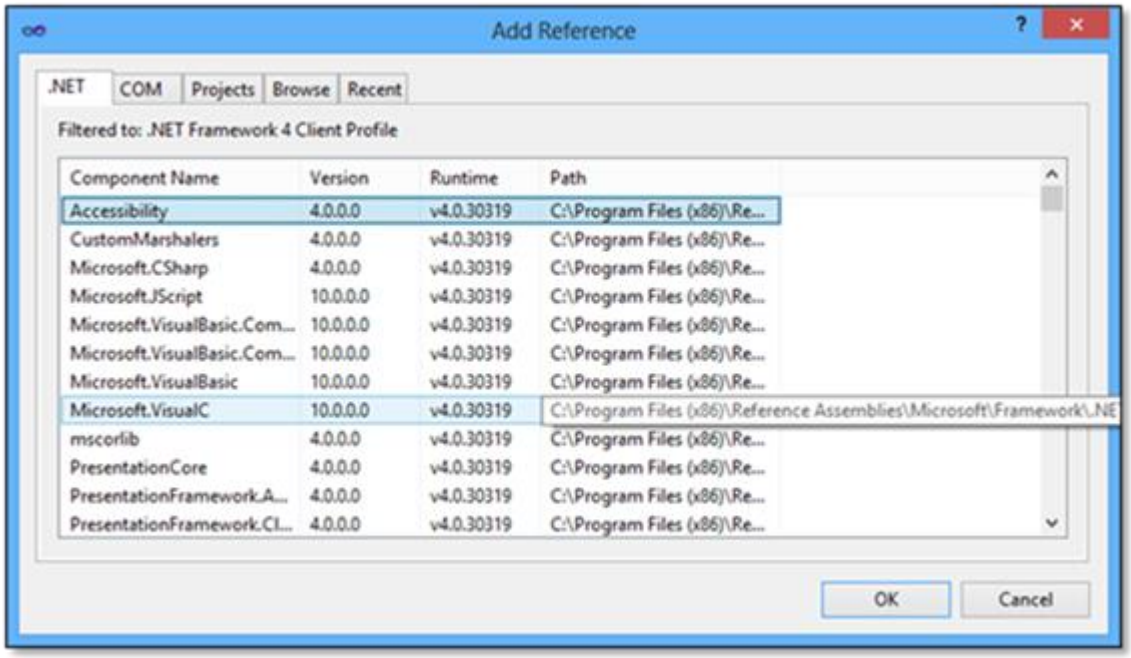
The name 'Class1' does not exist in the current context

The error message indicates that the compiler does not recognize the class name. The namespace with the 'using' contains error as well. The problem is that when you write a console project, the compiler knows nothing about another project, about the DLL, it does not know its name, and it does not know what we are referring to. Obviously it cannot see into our mind it, won't search over the entire disk looking for this DLL which contains this namespace and class. We must identify and adds this DLL to the console project manually.

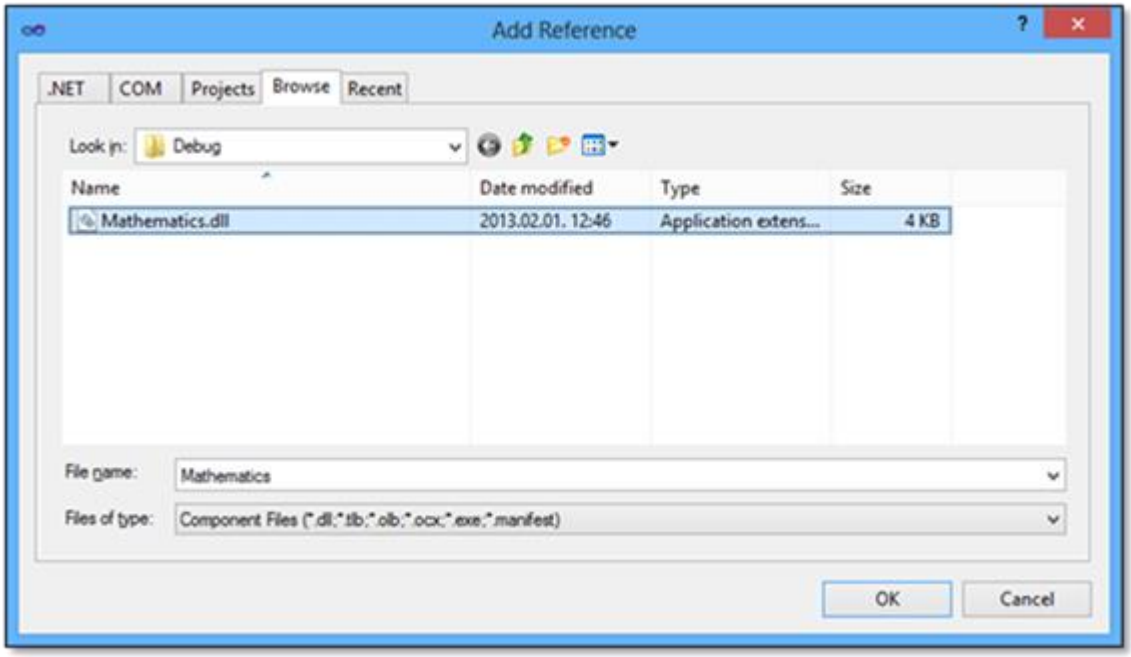
It can be done through the Solution Explorer. In the solution tree there is a *References* branch, opening it we can see the external DLLs actually used by the program. Into this list we must add the mathematics DLL. Let's right click on the References item and select the *Add Reference* menu item:



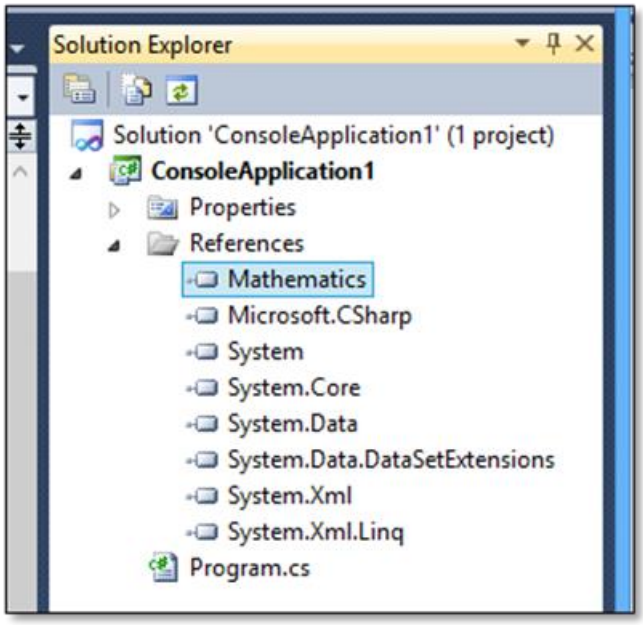
A serious complex dialog box pops up; we can identify the DLL we want to add to the project:



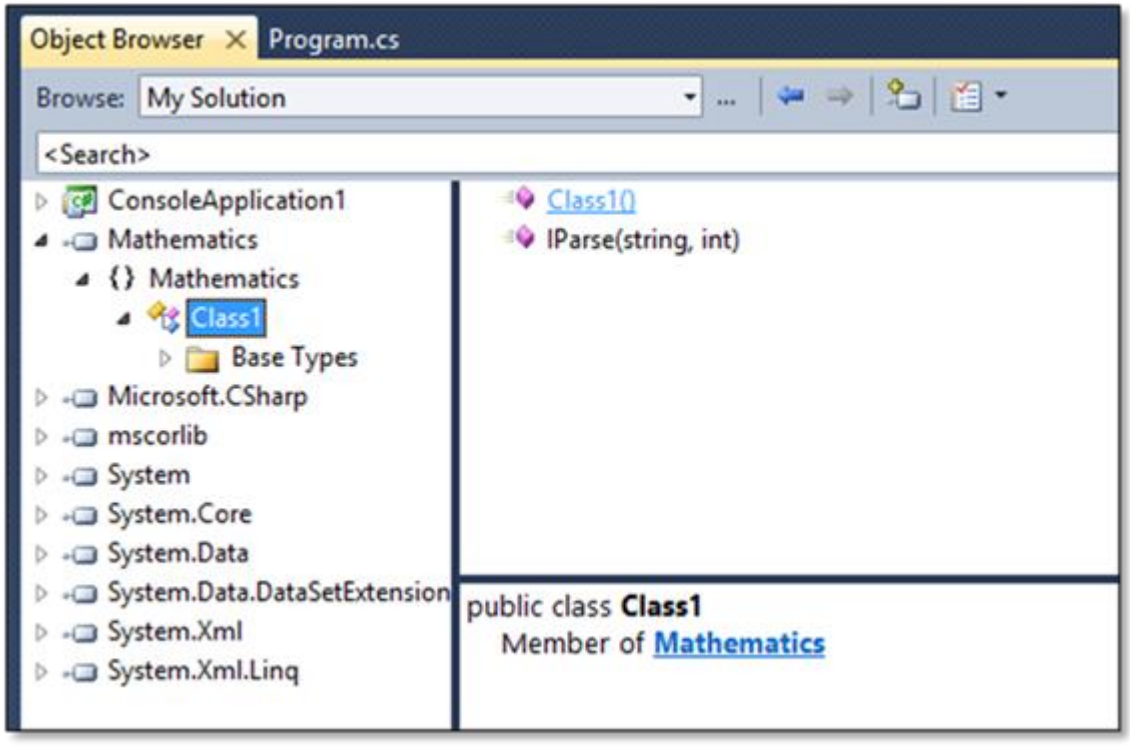
On the .NET tab the content of the GAC (see below) is listed. On the COM tab we can see the registered DLL-s which uses a previous Windows technology (Component Object Model). On the Projects tab we could see the DLL-s from the same solution. What we need is the Browse tab, where we can find any DLL stored in the disk by browsing the file system and folders. Navigate to *C:\Programs\Math\Bin\Debug* directory (where the DLL is compiled and stored), and click on the DLL file name:



If we did it right, the DLL is added to the References list, and it can be seen in Solution Explorer:



If the DLL is not created by us, and we do not have documentation, we will not know what classes and methods are in the DLL. However, right click on the DLL's name and select *View in Object Browser* menu item:



With a few clicks in the Object Browser window we can reveal the content of this DLL, we can see that a Mathematics namespace is defined (in the left tree the *{}* indicates the namespace) and at the bottom right "Member of Mathematics" indicates the same. We can see that in this namespace we have a class named Class1, which has two functions. One of the constructor (we have not created any constructor, but the default constructor is generated by the compiler), and an IParse function with, and we can see its parameterization (string and int). This is already much more than what a non .NET DLL can discover.

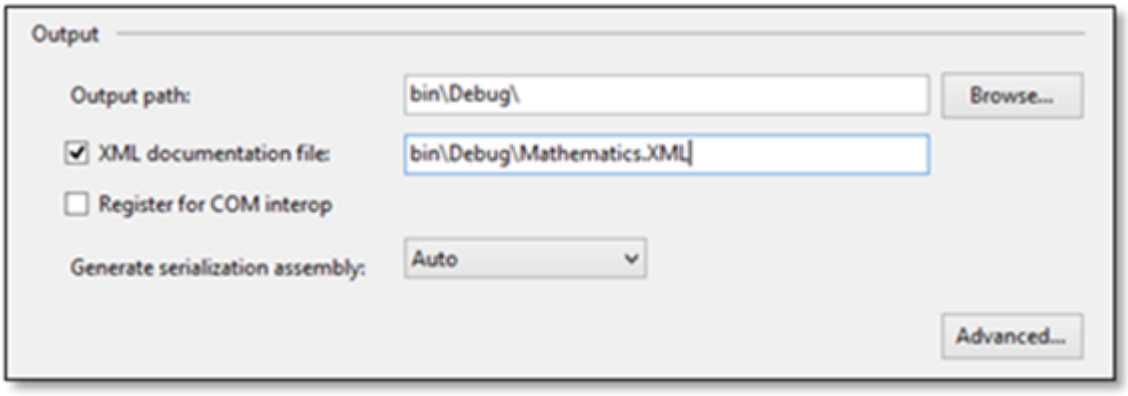
What is not here, what is the task of this function, what it will do, and what the meanings of the parameters are? We can help on it, if we go back to the DLL project, and write documentation comment to the function:

This function is similar to x, parses string to number. if it cannot be done, the default value is returned.

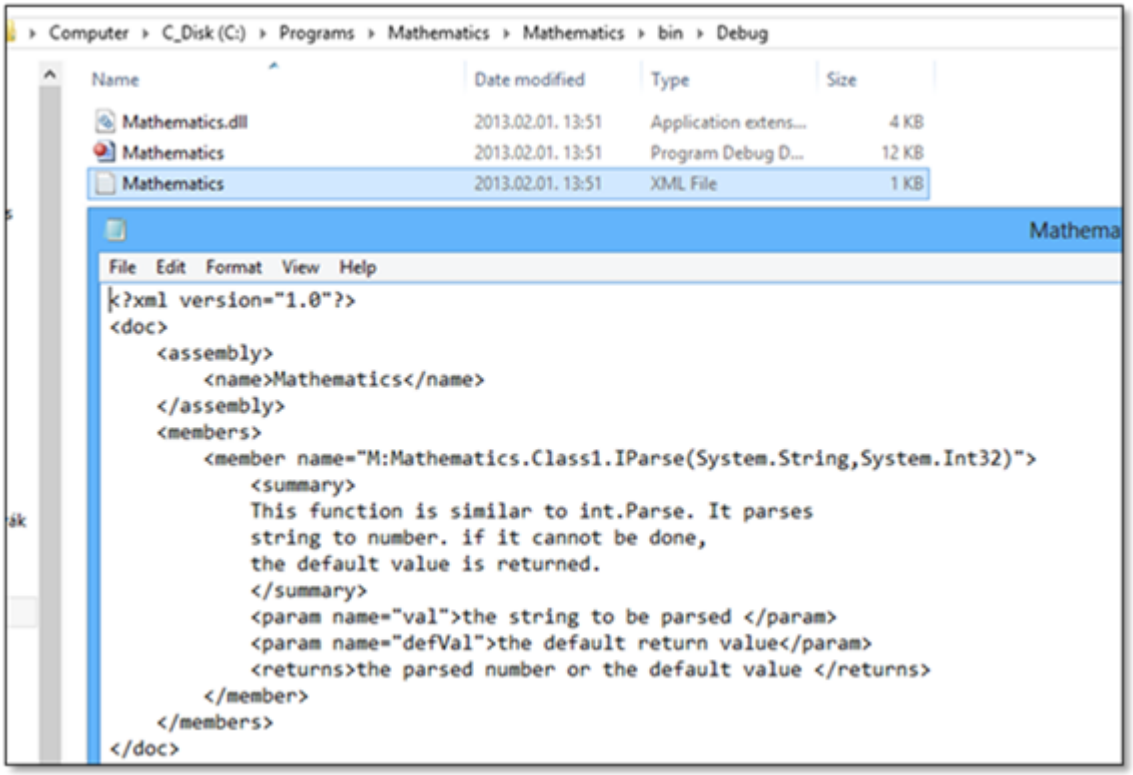
```
/// <summary>
/// This function is similar to int.Parse. It parses
/// string to number. if it cannot be done,
/// the default value is returned.
/// </summary>
/// <param name="val">the string to be parsed </param>
/// <param name="defVal">the default return value</param>
/// <returns>the parsed number or the default value </returns>
public static int IParse(string val, int defVal)
{
  try
  {
    return Int32.Parse(val);
  }
  catch { }
  return defVal;
}
```

Documentation comments can be written using three slashes. In a case of a function the summary shall be described, that is, what is the task to fulfil, what it will do. Then the parameters must be described one by one, their roles, and the possible values. In a case of function (not void) we can describe the return value, how it is calculated.

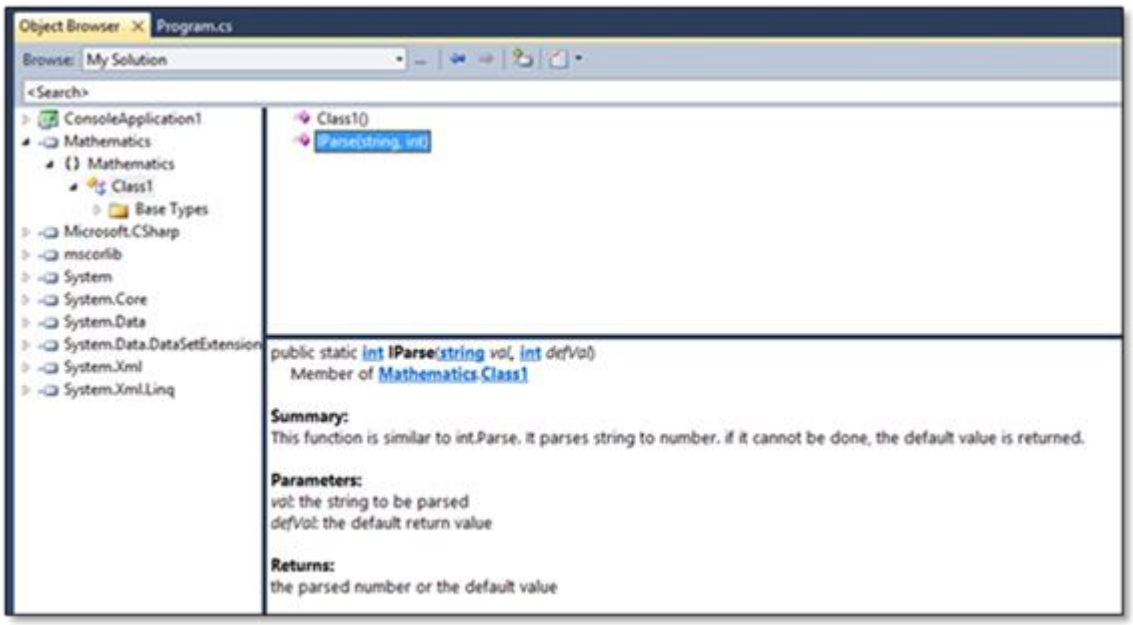
To write a documentary comment into the source code alone is not enough. The compiler should be instructed to collect these comments and extract them into an xml file and place it into the folder along the DLL. To ask this in Solution Explorer, right click on the name of the project, select *Properties*, then at the bottom of the *Build* section check the *XML Documentation File* check box:



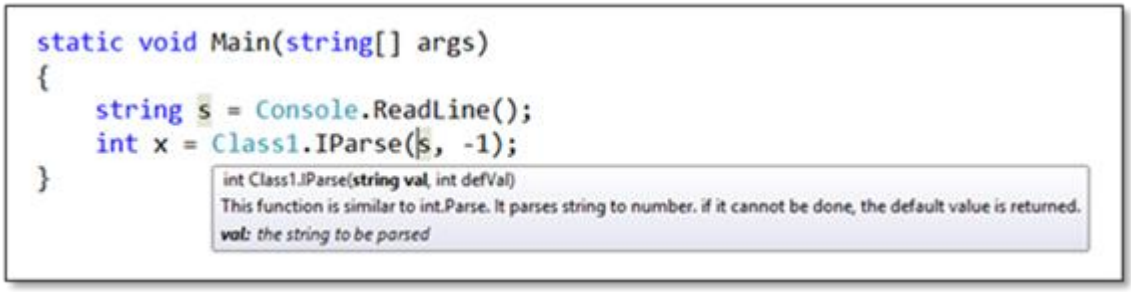
After the re-compilation (*Rebuild Solution*) next to the DLL an XML file will be created. The comments are collected:



Switching back to the console project selecting the IParse function the comments will appear in the Object Browser. Although to achieve this: the DLL must be refreshed, which can be done by successfully compile this the console project again. The earlier version of Visual Studio often forgets to update the xml documentation file, so it was necessary to remove and re-add the DLL into the references list:



In addition, editing the source code around the function this help also will appear:



27.5. 27.5. The DLL and the GAC

The GAC - Global Assembly Cache is a folder where applications can share their DLL-s with each other. The location of this folder for example

„C:\Windows\Microsoft.NET\assembly\GAC_32”

subdirectory, but it can be different, depending on the framework and the Windows OS version. We store DLL-s in the GAC folder, but only digitally signed DLLs can be copied here. The digital signature identifies the company which created the DLL, so if two different companies make two different DLLs- with the same name, both can be stored peacefully in the GAC, as the system can distinguish them from each other. On the other hand, it can store different versions of the same DLL created by the same company. The GAC works reliably from this aspect.

We can copy our custom DLL into the GAC by the *gacutil.exe*. This is a command line tool, we use to open a command window. To ask it to copy our DLL to the GAC, we must use the /i switch (install):

```
gacutil /i ourOwnFuncs.dll
```

The gacutil verifies whether the DLL is digitally signed, reads its version number, then copy the DLL into the GAC folder. Furthermore, in the Visual Studio in the *Add Reference* menu item we will find this DLL in the list, and can add to the project.

27.6. 27.6. The DLL and the OOP

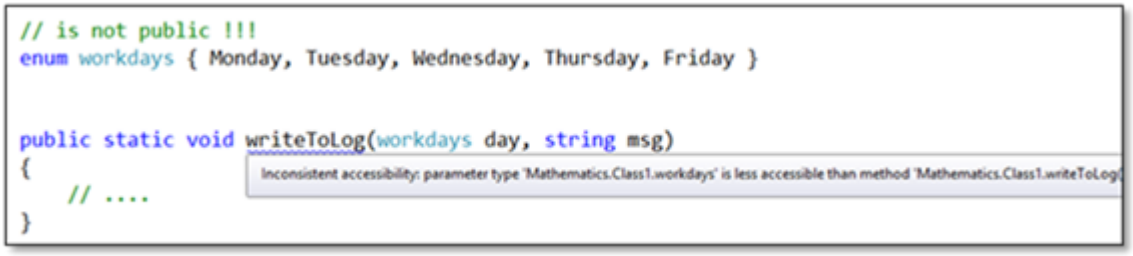
Making a DLL is a simple step with the help of VS. Discuss the additional rules defined by OOP and C#! What we should take care of?

A single DLL can contain several class definitions, some of them are public - made for the external application, while other classes maybe intended for internal use only. These can be “secret” classes (part of the precious know-how), or just utility classes that are not made for external use.

The classes made to be available to the third-party applications must be marked with ‘public’ attribute (**public class**), while the inner (secret or utility) classes are simply 'class'-s, without any qualification (which is essentially the same as 'private class', although this keyword cannot be add to the class directly). Inside a DLL obviously at least one 'public class' must exist[36], otherwise this DLL is quite unusual for the external application.

A public class is presented in the external application as it would be written there. If the same PC stores the source code of the DLL, and it is compiled as ‘debug’, debugging the main application VS is able to switch into the DLL source code, as well as we add a breakpoint to the source code of the DLL, the VS will stop the execution as well. This will help to test the DLL.

A public class may contain many functions. The public and protected ones are accessible in the external application (the same rules as the class was developed in the same source code, the public-s are directly, the protected ones only if a child class is inherited from the class defined inside the dll). Although, what will happen if a parameter of a public function has a type which is also defined inside the DLL but it is not public?



Inconsistent accessibility! The writeToLog() function cannot be published, as the outside world is not able to pass a parameter with this type as this type is inaccessible for him! Rule: if a method is available to the external application, all types of the parameters must be public as well (and the return type as well). It is interesting that if it throws any custom type exception, but this custom type is inaccessible (not public) to the outside world, it is not an error. The only consequence is that the outside world cannot write a catch branch with this type, only with the nearest available class type.

27.7. 27.7. Additional protection levels of DLL

If the DLL contains a non-public class still can have public methods and fields. This is comfortable, since inside the DLL all the codes can call and access them directly (this is considered as reliable, the same developers write the code who developed the class), but the outside unreliable world the entire class is hidden, cannot be accessed.

However, carefully set the public protection level to fields and methods for a public class. The code which will use (call) this fields and methods are written by a 3rd party developer. If we set the protection level to private, the fields and methods cannot be accessed neither by the code in the DLL, but we do not wish to hide them from ourselves. What is the solution?

Therefore, two other protection levels are added in case of DLL. Above the common (public/protected/private) protection levels we can use:

- protected internal,
- internal private.

The internal level of protection is equivalent to “public” for the code inside the same DLL. So “private internal” means “public” towards the DLL, and “private” for the outside world. The “protected internal” is public for the DLL, and “protected” to the 3rd party external application. The third possible pair would be “public internal” which would mean public towards to both direction, but the simple “public” does the same.

28. 28. Callback

The *callback* is technology, which is available from the beginning of the history of programming and proved its worth. We discuss it in one of the last chapter, because the mechanism is contrary to the principles of OOP, in fact, most principles can be avoided by using the callback method. However, the principles behind it inherent greatness, and worth to learn it as it means efficiency in programming.

The story began with the pointer. A pointer is a data type which is very close to the reference type in C#. In a pointer-type variable a memory address can be stored. This type have been introduced in the low-level programming languages, but the high-level languages tries to eliminate it as using the memory addresses are very efficient, but the compiler cannot check for suitable use of it, thus the code while executing fast, but only the skill of the developer can guarantees the proper execution.

An application typically divides the memory into three areas. First area is the code segment, where the program code (the function body etc.) is loaded by the OS. The second area is in the data area where variables are stored (this is the authority of the GC). The third is the stack, where we store the function parameters, local variables, return addresses of functions (this area is not interesting for us now).

The common data pointers are variables, which stores memory addresses pointing inside the data segment, so points into a data element directly. In fact, most variables are a kind of pointer, when we assign new values to

them it really means copy the new value to the memory area the variable points to. Typical use when we points to the 1st element of the array, read its value, then by moving to the next element we can read or modify all the elements of the array. It is a simple and effective way to access all the elements of the array in a single iteration loop.

A completely different type of pointers is when they point to not a data element but into the code segment, where the functions are stored. These pointers usually points to an entry step of a function. These pointers are named “function pointers”. With such a pointer the appropriate function can be called (started).

In C language, each function name is also a variable which stores the memory address of the function by default. In this language the absence of function start operator (the parenthesis) the function is not started but the result is the memory address itself (the value of the variable). If we add the round brackets (function call operator) to the function name it means we want to call this function.

In the C# language there also can be a variable whose value is a memory address of a function (method). The hardest part in to declare such a variable.

The difficulty caused by security (type correctness). Such a variable can store (in principle) the memory address of any kind of functions, as technical all the memory addresses look like a 4-byte[37] simple numerical values, technically an int type variable is also capable of storing one. In practice, however, the compiler must not only be aware of the fact that the variable stores a memory address of a function, but also must be sure what is the parameterization of this function is, and what is the return type of it.

To allow this we must create a **type**, which describes these information of a function. This type can be created by the **delegate** keyword. Assume we have a function which accepts two int-s parameters and returns with an int value:

```
static int add(int a, int b)

{

  return a + b;

}
```

If we want to store the memory address of this function in a variable, then its type must be defined as follows:

```
delegate int fv_add(int x,int y);
```

The type name is chosen 'fv_add’. Further information is included in this line the parameters of this function (int,int) and the return type (int). So the 'int' before the type name (fv_add) represents the return type of the function, the parts behind the type name represents the parameterization. For some reason, in the parameterization we must also declare names to the parameters, but this means really nothing (irrelevant) in this case, as we do not have to write the body of the function. The delegate statement is terminated by the semicolon at the end.

It is very important to understand that this statement creates only a new type. The 'fv_add’ type describes a type concerned to function with two int parameters and with an int return type. Variables of this type can only store memory addresses of such functions with this characteristic!

To do that, we need a variable of this type:

```
fv_add fv1 = null;
```

The ‘fv1’ is a variable, which requires 4 bytes to be allocated, and its initial value is null (according to the code above). Given that there is a function with such characteristic (two int parameters, return type is int), we can store into this variable its memory address:

```
fv1 = add;
```

Or in a shorter form:

```
fv_add fv1 = add;
```

Note: the function name (‘add’) is now on the right side of the assignment statement, but without the parenthesis and the actual parameters. If we would have used them it would be wrong for several reasons:

```
fv_add fv1 = add(); // ERROR
```

First, the function expects two parameters, two int-s. We have not passed them. Fill the gap:

```
fv_add fv1 = add(12,30); // ERROR
```

The assignment statement becomes better in many ways, but this time it means "start the function with these actual parameters, and assign the return value to the variable fv1". However, the function returns an int, and it is not possible to store it to a 'fv_add’ typed variable! Therefore, we cannot do it. To get the memory address of the function use the name (only the name), and do not start the function with the parenthesis (function call operator)!

The question is: what is it good for? If we store the memory address to another variable, we can start the same function using the variable as well with a completely similar syntax than traditionally. The following two lines are equivalent:

```
int c = add(12, 30);

int d = fv1(12, 30);
```

The similarity of the two lines are from the fact that the ‘add’ is also a variable, which initial value is the memory address of the function, and the 'fv1' is also variable which can store the same memory address of the same function. It has not turned out what is it good for, now we have two ways to call the same function, we can feel: one way is enough to do this! In addition, in the first statement it is clear which function is called, while in the 2nd case seeing only the 2nd line it does not turn out, we have to search backwards in the code to find which function memory address is stored in the 'fv1' variable.

The "what's in it good for" the answer is: eq. can be used to increase efficiency, but you it can also make the code elegant, we can achieve amazing flexibility if we are sufficiently clever and creative. And we can drop the inheritance principle, as it is suitable to replace it[38]!

28.1. 28.1. Application logic and user interface

Let's examine what the three-tier application development suggests. The principle says that a function of the application logic layer functions cannot communicate with the user, it is exclusive for the user interface layer functions. Let’s develop a program that enters a positive whole number (e.g. 100) and then generates and prints the prime numbers from 0 to 100. If we take this principle seriously, then we can plan the program as: the prime numbers generator function cannot prints the numbers (because it is in the application logic layer), the one which prints the numbers cannot generate the numbers (because it is in the user interface), so we must generate prime numbers and add them to a list, return the list, and the list is passed to the printer function. Therefore the suggestions of the principle are taken; none of the functions belong to two layers at a time! The user interface functions look as:

```
static int input()

{

  while (true)

  {

    Console.Write("end of range?");

    int n = int.Parse(Console.ReadLine());

    if (n > 1) return n;

  }

}
```

```
static void print prime(int a)
{
  Console.WriteLine("next prime is: {0}", a);
}

static void write primes(List<int> L)
{
  foreach (int x in L)
     print_prime(x);
}
```

The generation of prime numbers is done by the following functions:

```
static bool is_prime(int a)
{
  if (a%2==0) return false;
  int h = (int)Math.Sqrt(a);
  for (int i = 3; i < h; i = i + 2)
  {
    if (a % i == 0) return false;
  }
  return true;
}

static List<int> gen_primes(int n)
{
  List<int> ret = new List<int>();
  for (int i = 2; i <= n; i++)
  {
    if (is prime(i)) ret.Add(i);
  }
  return ret;
}
```

The main function to control the process:

```
static void Main(string[] args)
{
  int n = input();
```

```
  List<int> primes = gen primes(n);

  write_primes(primek);

  Console.ReadLine();

}
```

As we have seen the solution is complete, elegant, complies with the three-layer application development! Only one problem: efficiency! In the first round we find the prime numbers and collect them to a list, the list operations increase the execution time, and require and allocate a lot of extra memory. A separate iteration processes the list, and the overall execution time is quite slow, the memory requirement is high. Without the three-tier application development principle could be solved much simpler:

```
static void find_primes(int n)

{

  for (int i = 2; i <= n; i++)

  {

    if (is_prime(i))

       Console.WriteLine("next prime is: {0}", i);

  }

}
```

As we can see when we integrate the two operations into a single function, then the function while simultaneously performs the tasks of two layers, belongs to two layers at a time, the efficiency is increased. How can we keep the previous elegant solution, and efficiency of the later?

The answer lies in the function pointer[39]. First step is to prepare the delegate type to the prime printing function. The function receives an int as a parameter (the prime value to be printed) and returns a void. So:

```
delegate void fv_prime_printer(int x);
```

If we can declare the variables of this type, we can create a function parameter as well. Add a parameter to the ‘find primes’ function:

```
static void find_primes(int n, fv_prime_printer write)

{

  for (int i = 2; i <= n; i++)

  {

    if (is_prime(i)) ... ?

  }

}
```

When we call this function, first we must pass the upper limit of the range (n), and a memory address of a function which receives an int number, and returns with void. We have one: the 'print_prime'. Also we removed the list handling which was the main reason of the low efficiency. The main program which calls this function is:

```
int n = input();

gen_primes( n, write_primes );
```

As the second parameter of the function a memory address must be given, so again we do not have to start the ‘write_primes’ function with parenthesis, but we give the name of the function only, which represents the memory address of the given function.

Going back to the 'gen_primes' function when we find a prime number, we invoke the 'write' function whatever it does with the prime number. Because it holds a memory address of an appropriate function, we can call it:

```
static void find_primes(int n, fv_prime_printer write)

{

  for (int i = 2; i <= n; i++)

  {

    if (is_prime(i)) write( i );

  }

}
```

With the help of the function pointers we lifted out the functionality which belongs to another layer, and meanwhile we keep the high efficiency!

28.2. 28.2. Decide once – use many times

Let’s discuss another example, which illustrates the flexibility of a function pointers: the events of the program must write into a log file. These records can be stored in sql database, xml file, in a simple text file. We implemented all the three versions into different functions. The user of the application has selected a method during the installation (it might even choose not to use any kind of logging). The installation saves this selection into a configuration file, which is read at start up. The selection is read into a variable of type int, 0 means no logging at all, 1 is the sql, 2 is xml, 3 means logging to text file. The different function names are logToSql, logToXml, logToTxt, each function needs a single string parameter, the message to write to log.

In several point the program wants to write messages to the log. Obviously it is not a solution to insert the following code snippet into all the places:

```
string message = "calculation starts";

if (logMethod == 1) logToSql(message);

else if (logMethod == 2) logToXml(message);

else if (logMethod == 3) logToTxt(message);
```

Among other things we do not want to, because if we develop a fourth logging method, it must be inserted at each points and it is hard to maintain (and this is true even if we use *switch* instead of the *if*-s). It seems a good idea to replace it central message function to select the corresponding log writer method. Then the call to this central function is required only from the places of the application:

```
static void logWriter(string message)

{

  if (logMethod == 1) logToSql(message);

  else if (logMethod == 2) logToXml(message);

  else if (logMethod == 3) logToTxt(message);

}
```

This is a better solution, fairly centralized system, new methods can be added easily, so it is easy to maintain. But it is not effective, because each time we try to write a message to the log, we must go through this selection over and over again. Do it better with the help of the function pointer! The first step is to define the specific function type:

```
delegate void fv_logTo(string m);
```

Next step: we define an (even) static variable, which will hold the memory address of the selected function:

```
static fv_logTo writeLog = null;
```

After reading the configuration file before the program starts, we select the method to for log writing (once), select the function which will be used during the execution of the program:

```
if (logMethod == 1) writeLog =  logToSql;

else if (logMethod == 2) writeLog = logToXml;

else if (logMethod == 3) writeLog = logToTxt;
```

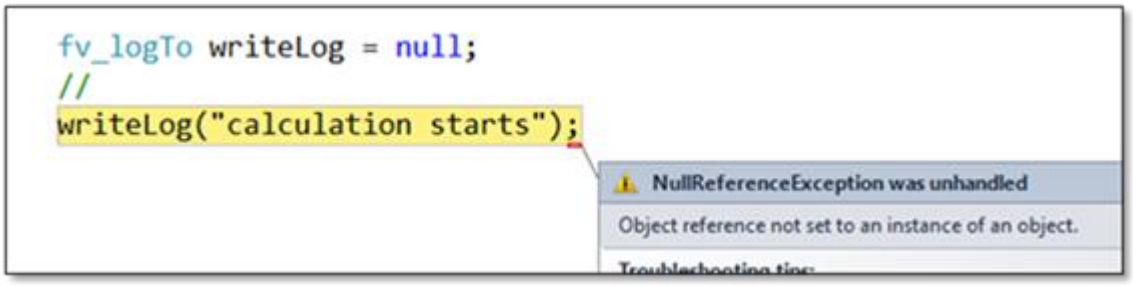
We are done. If we want to write log entries somewhere in the code we must use the function pointer:

```
writeLog("calculation starts");
```

Note that the solution still can be well maintained, as if a new log writer method is developed, the variable can store the memory address of the new function. The efficiency is improved significantly, as a call of writing a log entry does not go through the test (f-s or switch), in fact, the corresponding function is called directly without any further delay.

28.3. 28.3. Null-valued function pointers

We must consider it might happen that a function pointer variable stores no valid memory address, but null value. Such a case the function call will raise a runtime error (an exception is thrown):



The problem, unfortunately, cannot be handled elegantly. It is possible to be certain that this does not occur. For example we can create a default or empty-body log writer function, and when there is no selected function we can redirect the log writing operations to it.

```
static void logToNowhere(string msg)

{

}
```

Then we add this default selection at the end:

```
if (logMethod == 1) writeLog =  logToSql;

else if (logMethod == 2) writeLog = logToXml;

else if (logMethod == 3) writeLog = logToTxt;

else writeLog = logToNowhere;
```

Then there is no need to worry about the null value problem. If for some reason we do not want to handle the situation this way, we may check the null value before the call:

```
if (writeLog!=null) writeLog("calculation starts");
```

We can see it is not so elegant, not too efficient, but there is still no other way.

28.4. 28.4. Instance level functions

In the previous examples, class-level functions were used; the variables were stored memory addresses of such functions. Until it was simpler and easier to understand, but in fact, there is nothing to prevent that we use instance-level functions. The only thing that might be interested in: how to refer to the instance-level function to get its memory address?

Prepare a log writer instance:

```
class Email

{

  public string email_addr;

  public void send(string msg)

  {

    // TODO:

    // send the „msg” to the developer

  }

}
```

The variable (writeLog) is capable of storing instance level method address, there is nothing to do with it. However, to assign such a memory address, first we need an instance!

```
Email p = new Email("bugreport@sw-house.company.hu");

writeLog = p.send;
```

As we can see, storing instance-level method into a variable could be done very similarly; the only difference is to add an instance variable name to the method name just in front of the function name! This is because (as noted above) that the instance-level method requires the instance as an extra parameter, so the system is in fact not only stores the address of the function, but also the reference of the instance. Through the invocation of a function pointer needs no extra syntax, the system will pass the instance automatically - otherwise we would get an error.

Note: the GC cannot deallocate the instance until a function pointer stores the memory address of an instance level method, as it also stores a reference to the instance as well.

28.5. 28.5. Handling a list of functions

If we are familiar with storing and using memory addresses of a single function – step forward into storing and handling a lot of function addresses. Suppose we want the program to store the messages into an SQL database and send them as SMS to the developer as well:

```
static List<fv_logTo> logToList = new List<fv_logTo>();
```

Therefore, we add both of the functions:

```
logToList.Add( logToSql );

logToList.Add(logToSms );
```

When we want to log a message, we can call all the functions from the list one-by-one in a single loop:

```
foreach( fv_logTo f in logToList )

  f("error: disk full, program stops");
```

Here you can take advantage of the fact that the list always exists, even if it is empty. Thus, we do not have to deal with the null value. If the list is empty (no element have been added), the foreach loop will handle this case, so we will get no errors or exceptions.

28.6. 28.6. Event list

It is so common to handle not one but many function callback when a certain event happens, that the C # language has a special constructs for this task, which is called **event handler**.

Traditionally we declare a function pointer variable as:

```
static fv_logTo writeLog;
```

Such a variable can store one single memory address of a function, and we were able to call this:

```
writeLog = logToSql;

writeLog("hiba: varatlan programleallas, lemez betelt");
```

When we know the keyword 'event', we can add to the variable declaration:

```
static event fv_logTo writeLog;
```

Such the 'event' modifier the variable becomes a list (behind the curtains), and is able to store not one but many function addresses at the same time. Thus we cannot assign a memory address to this variable but add to the list with the help of the += operator. It is called “subscribing to the event”:

```
writeLog += logToSql

writeLog += logToSms;
```

Any number of functions can be subscribed to the event. However, to call all of them we need no loops at all, it looks like as a single function call:

```
writeLog("hiba: varatlan programleallas, lemez betelt");
```

The above syntax is a little bit confusing, because it looks like it would be a single function call. But since the 'writeLog' variable is not a simple variable, but also 'events', this is not a simple function call, but will call all the functions subscribed to the event in the order of the subscription.

28.7. 28.7. Inheritance in a different way

With a creative use of function pointers we can substitute the method overriding. Let us take an example of a simulation problem: our animals are been able to run, fly and swim:

```
abstract class animal

{

  abstract public void run();

  abstract public void swim();

  abstract public void fly();

}
```

For a concrete animal the development of specific functions can be completed. A domestic duck good at swimming, runs very slowly, and cannot fly. The cheetah runs perfectly, swims acceptably, but also cannot fly. The eagle cannot swim, does not run well, but fly like a goddess. With other animals, additional combinations can be imagined. How to build the inheritance tree structure to minimize the development (less code write)!? If the eagle is the ancestor of the duck, the duck also inherits the good flying ability. On the contrary the eagle would inherit good swimming from the duck. Obviously, do not set the ancestor of cheetahs to duck (and not vice versa). If the three classes are developed completely independently of each other, and no one inherits anything from the other, each method must be developed separately.

Rather than continue to rack our brains, develop the necessary methods (in this example as class-level functions):

```
class Animal

{

  public static void run_slow() { /* ... */ }

  public static void run_fast() { /* ... */ }

  public static void swim_asStone() { /* ... */ }

  public static void swim good() { /* ... */ }

  public static void swim_excellent() { /* ... */ }

  public static void fly_asStone() { /* ... */ }

  public static void fly_good() { /* ... */ }

  public static void fly_excellent() { /* ... */ }

}
```

Declare a (uniform) function pointer delegate type. Each of the above functions has return type void, with no parameters:

```
delegate void fv_move();
```

Adds the 'Animal' class with three fields, each of them is a function pointer type:

```
class Allat

{

  // ... functions ...

  public fv_move run = null;

  public fv_move fly = null;

  public fv_move swim = null;

}
```

Now we can create a given animal instance with the corresponding methods:

```
Animal duck = new Allat();

duck.run = Allat.run_slow;

duck.swim = Allat.swim_excellent;

duck.fly = Allat.fly_asStone;
```

To create another animal select another set of methods:

```
Animal cat = new Animal();

cat.run = Allat.run excellent;

cat.swim = Allat.swim_asStone;

cat.fly = Allat.fly_asStone;
```

Let's create a good constructor:

```
class Animal
```

```
{
   // .. constructor ...
   public Animal (fv_move fRun, fv_ move fSwim, fv_ move fFly)
   {
      this.run= fRun;
      this.swim = fSwim;
      this.fly = fFly;
   }
}
```

With this constructor we can create animals easier:

```
Animal crocodile = new Animal (
   Animal.run_slow,
   Animal.swim_excellent,
   Animal.fly_asStone  );
```

In summary, we can say that we got away the derivation, creation of child classes, the late-binding (as well as the VMT table with the extra memory requirement), while the usage of the instance is still elegant and efficient:

```
cat.run();
crocodile.swim();
eagle.fly();
```

If you still insist on the inheritance of the various animal classes, we can still create child classes:

```
class Cat : Animal
{
  public Cat()
  {
    this.run = Animal.run_excellent;
    this.swim = Animal.swim_asStone;
    this.fly = Animal.fly asStone;
  }
}
```

In this case, the instantiation of a cat means fill the fields with functions which are characteristics of the cat, so all the cats runs the same speed, cannot swim and fly. However, if there will be a special cat which somehow learns how to swim, for this special instance we can switch the 'swim' field pointing to another function. The usual inheritance way we must create another child class ‘catCanSwim’, where we override the inherited “cannot swim” method to a “can swim” one.

Note: this “switch” can be done either at run time. So while our little Garfield basically a traditional cat, which initially cannot swim, but later learns the secret of swimming. This case instead of dropping and re-creating of the instance we simply assign a new method to the swim field:

```
Cat garfield = new Cat ();
```

```
   // cat cannot swim yet

   // ...

   // later he learn it

garfield.swim = Animal.swim_good;
```

28.8. 28.8. TIE classes

The ‘Animal’ and ‘Cat’ classes described in the previous section are in common that they contain function pointer type fields that are set for specific functions. In addition, these classes would still contain standard OOP elements, fields, traditional methods (virtual and early-bound), properties, etc.

The TIE classes are classes that do not contain traditional methods; instead function pointer fields, so the 'Cat' is such a TIE class.

The TIE instances are not able to work at after the instantiation, since their fields must be filled with memory addresses of particular, existing functions. Each of these functions may be placed into different classes, otherwise, they can be a mix of class-level and instance-level functions. The TIE instance is a collection of these functions. Through the TIE instance the functions can be invoked in a simple syntax, as if they were actually the methods of the instance. The reason is the instance centralizes the storing and calling of a given collection of functions. The collection can be defined by a configuration file.

The memory addresses of a TIE instance can be replaced at run time, while the code which uses this instance realizes nothing about it, and continues the use of this instance.

Supposing our program handles contact information (contact list) of friends. The contact information can be saved into an SQL database immediately after the recording:

```
class contact

{

  /* ... fields and methods ... */

}

delegate void fv_save(contact c);
```

The TIE class stores only one function pointer:

```
class contactSave

{

  public static fv_save save;

}
```

The Main function sets the function pointer to the appropriate method.

```
contactSave.save = contactToSql.saveTo; // set to an function
```

We enter the contact data of a friend, then click the save button on the user interface. We call the save function to store the contact data to the SQL database through the function pointer:

```
public void btnSaveClick (Object sender)

{

  contact c = new contact();

  // fill the fields

  // ..
```

```
  // ask storage
  contactSave.save( c );
}
```

When we experience an error during saving to SQL, we can switch to another contact saving mechanism:

```
class contactToSql
{
  public static void saveTo(contact c)
  {
    try
    {
      // connect to sql server
      // exec insert
    }
    catch
    {
      // switch to another mechanism
      contactSave.save = contactToXml.saveTo;
      // and call it to save this contact as well
      contactSave.save (c);
    }
  }
}
```

Clicking on the button the code remains unchanged. It does not realize that the database save is switched to XML file save. It is all the same for it the actual method of saving, the only important is to be saved somewhere. As it is not its task to get know that the connection to the SQL server gets back, and (e.g. the SQL server has been rebooted, and in a few seconds it is able to store contacts again), the program might detected this and might can switch back to save to SQL server again.

29. 29. Reflection

As previously described, we name the .NET DLLs as "assembly", assemblies. We can add such assemblies to our projects during the development with the 'Add Reference' menu, and during the development we can use the public classes of them, we can develop child classes from them, and we can instantiate them. When the application starts, the operating system tries to locate these assemblies and load them into the memory. When one is missing, the OS will deny the start of the application.

More problematic is the case when we want to use such an assembly (DLL) which were not added to the project. First, during development we cannot refer to the classes inside it, as the compiler won't recognize them during the development. On the other hand, in the absence of this assembly our application will start, as the OS won't check its presence, it is not required at the start: it is not a registered part of the program.

Why would we do that, refer to an assembly which was not attached to the program? Among other reason because at the development phase we have not owned such assemblies, or does not want to make a strong bond between our application and this assembly. A good example might be the well-known WinAMP application, to

which many plugin can be downloaded and added, extending the basic application with several features. The playing an mp3 file is done by a dynamically added plugin, not mentioned the visual effects.

Of course we must follow some rules developing a plugin, as without them the basic (skeleton) application cannot recognize and use the content of the corresponding assembly. For example, a plugin dll must contain a well-known named class, and a method inside it with a given parameterization. If the assembly has it, the application can load the assembly, find this method, and call it when the service is needed. Unfortunately hard to determine if the method really performs the expected task, it is a matter of trust between the application and the method.

29.1. 29.1. To load an assembly dynamically

Loading external assemblies dynamically is handled by the classes and methods defined in the *System.Reflection* namespace. We give a function which loads and assembly to the memory according to its file name:

```
using System.Reflection; using System.IO;

class PluginManager
{   public static string pluginRootDir = @"C:\myProgram\myPlugins";        public static
 Assembly LoadAssembly(string dllName)   {     // if the extension is not dll -> add
it     if (Path.GetExtension(dllName).ToUpper()!="DLL") dllName+=".dll";              //
extend with the plugin root directory name     // (absolute file name)

    string dllFileName = Path.Combine(pluginRootDir, dllName);              // load the
dll and make a reference to it

    // in case of error 'a' will be equal to 'null'     Assembly
a = null;     try { a = Assembly.LoadFile(dllFileName); }     catch {}
      // return the reference to the assembly (or null)     return a;   } }
```

When we pass a short file name to the *Assembly.LoadFile* (not a full path) it tries to determine its full name with some kind of heuristics. It tries to find the assembly with the given name in the same directory where the .exe is, next in the GAC or in other system directories. Avoiding this we give a full file name adding the plugin root directory name to the file name. This root directory name can be defined in a configuration file.

29.2. 29.2. Referring to the application itself

The actual application (.exe) is an assembly as well according to the .NET concepts. It is an assembly which is not an extension ".dll" but ".exe", and there is a Main function inside it. This does not contradict the assembly concept. It is not necessary to have the ".dll" extension to the assembly, and loading a file into the memory will not start it, so it is unimportant to own a Main function or not. Therefore with the method presented in the previous section, not only .dll files but .exe files can be loaded as well. Of course, here we talk about only. NET DLL-s. This method won't work with native (Win32) EXEs or DLLs.

So the actual application .exe is a kind of assembly, but it is not required to load it into the memory as it is already loaded. When we want to get a reference to this assembly we can call the *GetExecutingAssembly* function:

```
using System.Reflection; class PluginManager {     public static Assembly
myExe()     {         return Assembly.GetExecutingAssembly();     } }
```

As we can see with both methods we can receive the same object type, the "Assembly" type, so we can work the same way further with the application assembly itself and with a dynamically loaded assembly.

29.3. 29.3. Finding a class inside an assembly

If we have an instance of the Assembly class, it is possible to find a class inside it specified with its name. We must know the name fully qualified name of the class to find (the namespace and the name of the class). During the development of the DLL the class must be defined as public (public class), since such classes are available outside the DLL.

A reference to the class is given by an instance of the System.Type class. The Type instance holds a lot of information to the class, for example we can query as a list of the methods of the class, etc.

```
Assembly dll = PluginManager.LoadAssembly("somePlugin.dll"); Type
t = dll.GetType("myNamespace.myClass", false, true);
```

We can search inside the dll with the help of GetType function. The name of the namespace and class name is given as literals in this case, but in a more advanced case it may come from a configuration file (eq. xml or ini).

The 2nd parameter of the GetType means when the class does not exist inside the DLL then we want no exception, instead we require a null value as a return value. The “true” value would indicate we needs an exception when the class does not existed.

The 3rd parameter of GetType is set to true, it means the search must be case-insensitive. The C and C# syntax are case-sensitive in other cases, but we can ask the search process to be insensitive (as in this case). This is a concession to the namespace and class name.

In the above example, the 't' will be a null value or will hold a reference to a Type instance which describes the information about the given class. This search can be started on a dynamically loaded assembly, or on the loaded application executable.

29.4. 29.4. Finding a method inside a class

When we have a Type instance (eq. through the GetType), then we can query additional information. If you are looking for methods, we will receive MethodInfo class instances as a result:

```
MethodInfo creator = t.GetMethod("Create",

                          BindingFlags.Public | BindingFlags.Static);
```

In the code above, the 't' is a Type descriptor instance (of 'myNamespace.myClas' class), and we want to find its “Create” method. According to the overloading rule, there can be many methods with this name. In the example we are searching for a method which is 'public' and 'static' (class-level method), and its parameter list must be empty (as we do not specify anything for the parameterization).

```
MethodInfo
dosome = t.GetMethod("DoSomething",      BindingFlags.Public | BindingFlags.Static,
 null,      new Type[] { typeof(int), typeof(string), typeof(int) },      null);
```

In this example, we gave an exact specification to the method to find: we want a method with the name 'DoSomething', this method must be class-level, public, its parameterization contains three parameters in the order: int, string, int.

Very similarly we can search for fields (MemberInfo instances, .GetMember (...) function), constructors (ConstructorInfo instances, .GetConstructor (...) function must be used), but we can also query for properties. We can check which interfaces the class is compatible with, who is its ancestor class, etc.

29.5. 29.5. To call a class-level method I.

When we successfully query method info with the GetMethod, we can invoke it:

```
// the method to call looks as: //    public static void Create() { ... }
creator.Invoke(null, null);
```

With the “Invoke()” we can called a method. The first parameter (null) means that this is not an instance-level method (it is not, as it is class-level). The second parameter (again null) means it does not expect parameters.

Otherwise the Invoke() will start the given method and returns with the return value of the method. We do not deal with this now, because it is a void function.

29.6. 29.6. To call a class-level method II.

If the method requires actual parameters to be passed, we must pass them:

```
// the method to call looks as: //    public static void DoSomething(int a, string r,
int b) { ... } dosome.Invoke(null, new Object[] { 12, "Hello", 20 });
```

As we can see, the first parameter of the Invoke remained null, since this is also a class-level method. The second argument, however, specifies the parameter values to be passed, we create an array with an int, a string, and other int values.

29.7. 29.7. To find an instance-level constructor and the instantiate

When we load an assembly and we have found the class inside it we want to instantiate (because we want to call an instance-level method later), then the next step is to find a constructor. This requires querying a ConstructorInfo instance:

```
// we find a constructor with (int,string) parameters ConstructorInfo
c = t.GetConstructor(

                        new Type[] { typeof(int), typeof(string) } );
```

If we have successfully discovered the instance-level constructor, we can call it and create an instance of the given class:

```
// to call the (int,string) constructor Object x = c.Invoke(new Object[] { 12,
"Hello" }) );
```

The Invoke() returns with a universal Object type, but the instance is a real instance of the given class, its memory address is stored in variable ‘x’. We could check it with the 'is' operator, but in real we cannot, as we should write the name of the class statically to the source code, and we cannot, as this assembly is not attached to the project but loaded dynamically.

29.8. 29.8. To find an instance-level method and invoke it

The static type of 'x' is Object, despite this we can call a method of 'x' without any type casting. We can discover the method with a single GetMethod() call. During this we do not add the BindingFlags.Static flag, instead we use BindingFlags.Instance:

```
MethodInfo
calc = t.GetMethod("calcDivide",      BindingFlags.Public | BindingFlags.Instance,
null,      new Type[] { typeof(int), typeof(int) },      null);
```

To call an instance-level method also we use Invoke(), but in the first parameter we specify the instance previously created ('x'). If the function has a return value, then the Invoke() will box it into an Object value, so we might need a type cast:

```
// the function gives back a 'double',

//   and it is 'double calcDivide (int,int)'
double res = (double) calc.Invoke(x, new Object[] { 12, 20 });
```

So when we know the class name (string), then use Reflection, we can invoke class-level functions, we can create instances with the constructors, and we can call instance level methods as well. This is unusual, since by statically adding the assembly to the project we can do these things by simply writing the functions calls directly to the source code, and the compiler check it strictly. However, if the type is not present at compile time, we cannot insert it into the source code. In the case of run-time dynamically loaded DLLs, this is the only way to do it.

Let’s see another example of where the reflection cannot help, nor is the assembly loaded dynamically. In our zoo simulation we have a basic ‘Animal’ class, which contains the basic functions of an animal (eating, sleeping, replicating) we want to implement. We will inherit the specific classes like panda, tiger, hummingbird, etc.

```
abstract class Animal

{

  public abstract Animal reproduce();
```

```
}
```

The reproduce() method has no parameters, it returns a new 'Animal' instance (in fact, a specific type compatible child class instance). The task is to create the baby animal from the corresponding object class. By this we mean that if it is called from a panda instance then a new panda must be returned. Therefore, we cannot write the body of the method in this class, we are forced to write the specific code to the specific child classes:

```
class Panda : Animal

{

  public override Animal reproduce()

  {

    return new Panda();

  }

}
```

We must write a similar method to each child classes – one-line functions for the tigers, and penguins as well. It's not a big job, but we want to move this job to the ancestor class and forget about it.

To do this we consider the usage of the ‘this’ keyword, as it refers to the actual instance, and whose dynamic type is the given specific type of the child class. With the Reflection we can get a Type instance about the ‘this’ dynamic class, which in the case of a panda will give the type descriptor of the panda class, for a tiger instance it will gives the tiger class information. As we can use the ‘this’ keyword, it is easy to get the Type instance with the help of the GetType instance level method (inherited from the Object class):

```
class Animal

{

  public Animal reproduce()

  {

     Type t = this.GetType();

     ...
```

After we have the Type instance, the next step to find the parameter-less constructor, invoke it and then we have a new instance of the proper (panda, tiger etc..), then we return it with:

```
public Animal reproduce()

{

  Type t = this.GetType();

  ConstructorInfo c = t.GetConstructor(new Type[] {});

  return c.Invoke(null) as Animal;

}
```

We can easily check the correct operation of this method. Let’s define a very simple child class:

```
class Panda : Animal

{

}
```

Then prepare a simple Main function to test:

```
Panda p = new Panda();
```

```
Animal X = p.reproduce();

if (X is Panda) Console.WriteLine("it is a new panda ");

else Console.WriteLine("hm... something wrong");
```

The program as is expected displays the "it is a new panda" message, so the reflection created a new panda instance. Of course, it would have been possible to make an instance with a constructor who have real parameters as well, as we can invoke a parameterized constructor as well.

30. 30. Summary

The note was targeted to illustrate object-oriented programming techniques, their applicability through relatively simple examples. We hope all of the techniques were understandable, and we hope that we were able to present understandable and acceptable field of use, and we convinced the respectable reader, that the solutions in a clever and creative hand allow the user to write a robust and elegant code.

We haven't discussed a lot of things, and many things were not examined in sufficient depth. It is possible that the examples were not always on scientific heights. Because we want to demonstrate techniques, we were not looking for examples that would use such algorithm where the understanding of them would have demanded most of the time, and would have covered the essence of the ideas. Many times we have not written the bodies of the methods, in a form of comments we indicated the task to do. We admit in many cases the exact code of the body would have been exciting as well. We hope, however, the reader puts down this note not disappointed, but with a feeling that it is not a mystical world filled with instances, where exceptions are thrown and caught, with overloaded operators, where most things have two types, and where it can be normal when we develop a method with no body at all. We wish good luck to all our dear Readers to explore this world!

throw new EndOfSemesterException("The End.");

31. 1. References

1) Illés Zoltán: Programozás C# nyelven, JEDLIK OKTATÁSI STÚDIÓ, Budapest, 2005

2) Sipos Marianna: Programozás élesben, Infokit, 2004, ISBN: 963 216 652 3

3) 2. Jeffrey Richter: CLR via C#, Fourth Edition, Microsoft Press, A Division of Microsoft Corporation, One Microsoft Way, Redmond, Washington 98052-6399

4) Bertrand Meyer: Object-Oriented Software Construction 2nd edition, ISE Inc.Santa Barbara (California)

5) Andrew Troelsen: Pro C# 5.0 and the .NET 4.5 Framework, Apress, 2012, ISBN: 978-1-4302-4233-8

6) Nyékiné Gaizler Judit (szerk): Programozási Nyelvek, Kiskapu Kft, 2003.

[1] From site http://en.wikipedia.org/wiki/Source_lines_of_code#Example.

[2] A statement which evaluates to true always during the execution of the program, like "the value of the 'age' is always between 18 and 60"

[3]http://en.wikipedia.org/wiki/Alan_Kay

[4] eq. namespaces.

[5]It is 5, the other 2 will be described later when we will be familiar with the DLL-s.

[6]Actually, not quite true, since the initial value of the field does not yet meet the criteria, but to solve this we must wait for getting know the technique of how to write a constructor.

[7]this process can be broke off using the statement ‘catch’ (see later).

[8] An idea from the Matrix movie – a little boy told Neo “*Do not try to bend the spoon — that's impossible. Instead, only try to realize the truth: there is no spoon*”.

[9] the initial value of an int-typed field is always zero which is guaranteed by the runtime system

[10] ‘neptun’ is a computer system, where each student has an unique id, which is a string, at least 6 characters long, called as the ‘neptun id’.

[11] the exact amount of bytes depends on the amount of the instance level fields and their types, but there are other factors

[12] not in all languages, for example in Delphi the name of the constructor can be chosen freely, usually ’Init’ or ’Create’ is selected.

[13] It is not the accurate memory size requirements, it is affected by the data alignment to the memory boundaries, and the presence of other technical fields in the instance

[14] The classical meaning in the form ‘func()’ the ‘()’ is the function call operator, and in some languages like C, C++ it really is. The C# is a language based on C language, so it is not a major error to name ‘()’ as the function call operator. In C# language in the table of the operator precedence it is also mentioned, although its role is lesser than in C or C++, since not applying it usually produces a syntax error.

[15] from http://www.snopes.com/humor/nonsense/kangaroo.asp site. This story is not authentic, but is interesting.

[16] It is named the “singleton” programming pattern

[17] In a console-type program the Windows.Forms namespace not be opened simply, unless the DLL (assembly) containing this namespace added with the "add reference" menu point.

[18] unless it is not overridden

[19] It is intended to use a double in this example. When it would be an ’int’, would be much harder to recognize the conflict.

[20] ss

[21] many tables are built, but now this is the interesting one for us

[22] in a case of a 32 bit processor architecture, for a 64 bit processor a memory address is 8 bytes!

[23]Such language is the Borland Delphi 7 (2002), where we can choose with the help of 'dynamic' and 'virtual' keywords. In the child classes the developers must use the 'override' both cases, since the decision cannot be changed later, cannot be modified.

[24] actually the best execution speed is given by the early binding.

[25] simplest case, but we can open such way to allow other processes to open it as well.

[26] if allocate memory through instantiation, the GC will find the memory are and free up automatically, so there is no need to free up explicitly

[27] The "handle" the concept is not an OOP concept. This is a numeric value that identifies a resource to the operating system. The handle is created and passed back by the OS to the program, which cannot do anything with it but give to the OS back later referring to the corresponding resource. Therefore, it is like a memory address, though the handle itself usually is not considered to a memory address, it is more like as an array index. It is conceivable that the array itself is stored by the OS itself, so that the index value is actually meaningless to the program, but by passing the back to the operating system (Win32) functions they are able to access this array, and are able to work with the resource.

[28] The prototype of a method is the header of it, that is, the method name, the return type and the formal parameter list. The property prototype consists of the name, and the mark of the ‘get’ and ‘set’ parts (or both).

[29] unless … example will be discussed later!

[30] It is the decision of the compiler that an operator is transformed to function call, or (especially) when the function body contains only a few instructions, they can be inserted into the operator use (inline function). Since a function call allocates an amount of memory and extra time, in simpler cases, the compiler might choose to copy the code from the function definition directly into the code of the calling.

[31] The compiler works in several phases. In the first step, the lexxer which splits the text into tokens and symbols, then the parser will check the order of the symbols (sequencing, type-correctness), and finally the code generation phase (it is often built into the previous phase, running parallel to it) is the machine code is generated. This can be followed by an optimization phase to make to the final version.

[32] Not necessarily the machine code of the processor. In C# for example a virtual machine executes the code, so the compiler generates a middle-level binary code. In general we can say that a compiler generates (transforms) the code into a different programming language, which is typically a language with lower level abstraction. For example, some C, C++ compilers can output assembly language code.

[33] Other extensions may occur, for example .drv, .fon, etc.

[34] the function name inside our program and the "function name" inside the DLL - may be different.

[35]http://www.spinellis.gr/pubs/jrnl/1997-CSI-WinApi/html/win.html, *A Critique of the Windows Application Programming Interface*, Diomidis Spinellis, University of the Aegean

[36] In a namespace there can be interface as well as struct, enum and delegate. Between these elements there must be one with a public definition, otherwise the DLL won't share anything with the application.

[37] in the case of 32 bit processor architecture.

[38][38] in several cases, see the examples later.

[39] it can be solved in OOP style as well, with an interface and an object instance, but we must admit this solution (with the function pointers) is simpler and clearer.

Index