Namespaces and keyword `using`

- Last time we remarked that each class must be defined in some namespace.
- When we want to call anything from different namespace, we have to say the name of the namespace...
- ... or use keyword `using`.
- See the example in Studio - usings are of this type.

Comment

about C#

- Note that now we understand the "ritual" necessary to write arbitrary program in C#.
- Also note that now you are able to write in C# almost anything you were able to write in Pascal.
- What will be doing the rest of semester?
- Except of algorithms and theory we proceed with further parts of C#.

Input and Output

- Functions `Read` and `ReadLine` in `Console` we already know.
- Output was so far very simplified. We are missing something like...
- `writeln('Value of a is ',a,', value of b ',b,' and their sum',a+b);`
- `Console.WriteLine("Value of a is {0}, value of b {1} and their sum {2}",a,b,a+b);`
- Note that it may be dangerous to pass a string directly to the output!
- We may use these placeholders also for output formatting in much more sophisticated way than in Pascal.

Objects as structures

instanciation

- Note that inside a namespace we may define many classes.
- We will try some classes with not only static attributes and methods and see what happens.
- As an example, we implement complex numbers.

Example

class-definition

```
namespace nothing {
      class compl {
          int re,im;
          public void set_nr_up(int x,int y)
          {    re=x;
               im=y;
          }
      }
      class Program {
          static void Main(string[]x)
          {.........}
      }
}
```

Instanciation

example - better version the same occurs next week

```
namespace Nic {
      ........
      class Program {
            public void Main(string[]x)
            {    compl a=new compl(),b=new compl();
                 a.set_up(0,0);
                 b.set_up(10,5);
            }
      }
}
```

Complex numbers

further interesting methods

```
class compl {
      .........
      public void add(compl what)
      {     re+=what.re;
            im+=what.im;
      }
}
```

Question: Are `re` and `im` public, private or protected?

Complex numbers

passing the argument by result

```
class compl{
      .............
      public void value(out int x,out int y)
      {    x=re; y=im;}
}
```

- What if we were passing it by reference...
- ... and if we did not initialize values passed by reference (as those values get immediately overwritten)?
- Attention, please, modifiers `out` a `ref` have to be used even when calling the function: (`x.value(out a, out b);`)!
- What in the example seems like the biggest nonsence?
- Hint: Fact that value containing the value is named `re` and `x`. Can we avoid it?

Step aside

scope-resolution

- When looking for a variable (identifier), compiler tries first arguments of the recent function and local variables,
- after that it tries the attributes of own class.
- What happens if an attribute is named as an argument?
- Attribute is covered and we could not see it. Thus in any object there is a variable `this` referencing it. Beware in static context (this is invalid)!
- Better implementation:

```
public void value(out int re, out int im)
{    re=this.re; im=this.im;}
```

Constructors I/II

are filling the object in

- When implementing, e.g., binary tree in Pascal, after allocating the structure, we had to fill it (almost always in the same way) which is sometimes clumsy.
- Thus constructor was designed.
- Constructor is a function that is called when creating a new object.
- Syntactically we may observe it as an unnamed function returning object of its underlying type [class].
- Or we may imagine it as a function named as the class without the resulting data-type.

Constructors II/II

are normal overloadable functions

- There may be more constructors for one class, just they have to differ in the structure of arguments (number, data-types).
- **Remark: This is called overloading and in C# any function may be overloaded in this way.**
- If we define no constructor, a default constructor (with no parameters) gets generated.
- When we define any constructor, the implicit one is not generated!

Example on constructor

```
class compl {
      ..............
      public compl (int re,int im)
      {     this.re=re; this.im=im;}
      public compl()
      {     this.re=0; this.im=0;}
}
```

Can't we implement the argument-free constructor better?

Constructors

calling a different constructor

- To call a different constructor we use semicolon and say what should be called like this:
- `public compl():this(0,0){}`
- In the braces we may define further code that gets called after the "colleague".
- Step aside: When using inheritance, keyword `base` may be used for the parent in the same way as `this`. So we can (similarly) call the parent's constructor

Destructor

- Analogy to constructor, called when deallocating the object.
- This usually does garbage-collector, thus its use in C# is complicated.
- Destructor's name differ by the wave: (`~compl`).
- It takes no arguments and returns nothing.
- In C# not so widely used, in C++ it has better use.

Garbage collector

... in C# is another good reason why we taught you Pascal in the winter term

- After we allocate an object (`new typename();`), we are working with it as in Pascal...
- ... up to the moment when we want to deallocate it.
- Instead of deallocation we simply drop the reference at it (in Pascal $\Rightarrow$ memory leak).
- In C# garbage-collector takes effect after some time.
- Amount of available memory:
  `System.GC.GetTotalMemory(bool);`
- Explicit call of Garbage-collector:
  `System.GC.Collect();`

Arrays

they are completely simple although they behave differently than in Pascal

- Defining an array type variable we indicate by square brackets in front of the variable name:
- int [] intarray;
- Compared to Pascal, we have only arrays of previously unknown size, we have to initialize them - allocate a space for them.
- Again we use operator `new` for that:
- intarray=new int[10];
- Arrays get indexed from 0 (up to length - 1)!
- Arrays of some types may be initialized immediately:
- `int[]arr=new int[3]{1,2,3};` or
- `int[]arr=new int[]{1,2,3};`

Arrays II/III

- Arrays of type we defined (say `compl`):
- `compl[] arrc=new compl[10];`
- The array is uninitialized so far [full of `nulls`]!
- `arrc[0]=new compl();`
- Array-length – `length` attribute:
  `int len=arrc.Length;`
- Accessing element out of range causes IndexOutOfRangeException.
- Use of uninitialized array cause NullReferenceException.

Arrays III

- Multidimensional arrays: int [,]arr=new int[2,3];
- This array is rectangular and:
  `arr.Rank==2` and `arr.Length==6`
- Nonrectangular arrays (array of arrays):
  `int [][]arr=new int[3][];`
- Later we perform: `arr[0]=new int[2];`
  `arr[1]=new int[3];`...
- Construction `foreach`:
- `int []arr=int[]{1,2,3,4,5,6};`
  `foreach(int i in arr)`
  `    Console.WriteLine(i);`