Where we have finished

...there we shall continue...

```
namespace x{
     class y{
          public static void Main()
          {    System.Console.WriteLine("This is a
program doing something...");
               for(;;);
          }
     }
}
```

Constants

- As in Pascal, constants are supported, but in a bit indirect way.
- Step aside: Modifiers are modifying something (variable, function) and they behave in several different ways. We have seen some of them...
- `public`, `static`, there are many more, e.g., `virtual`, `unsigned`, `protected`,... we will see them incrementally.
- Constants are operated by modifier `const`. Example:

```
const int x=1; //nobody can modify x later!
const double pi=3.1415926535898;
//nobody wants to change pi
```

Further control structures

- We already know `if`, `while` and `for`,
- missing equivalents of `repeat ... until` and `case ... of ...`
- The former is replaced by `do body while(condition);`
- Compared to Pascal, cycle is iterated while the condition is true!
- `case ... of` is replaced by very similar construction

```
switch(expression)
{    case ....: body break;
     case ....: body break;
     ...
     default: body}
```

Example

do ... while(...)

```
int i=0;
do
      Console.WriteLine("I should have learnt
better!");
      i++;
while(i<1000);
```

Example switch

```
int i;...
switch(i){
     case 0: Console.WriteLine("It is zero!");
     break;
     case 1: case 2: Console.Write("It is 1");
          Console.WriteLine("or 2");
     break;
     default:
          Console.WriteLine("Neither of 1, 2, 3");
}
```

Functions and their parameters

- We already know how to define function:
  `static int add(int a, int b)...`
- By default, the argument is passed either by value or by reference (numeric- and reference-oriented types).
- We can enforce numeric-oriented type to be passed by reference using modifier `ref`:
- `static void sum(int a, int b, ref int c)...`
- Often we use passing by reference just to return a result. Thus C# supports passing an argument by result.
- Semantically identical to passing by reference, just at the beginning the variable does not need to (i.e., is not) defined.
- This time we use keyword `out`:
  `static void sum(int arg1, int arg2, out int res)...`

Further specialities of object programming I/II

instance-creating, a.k.a., what means word `static` in front of `Main`

- Classes are like data-types while objects resemble individual variables.
- Where are those objects by now?
- We are still ignoring possibility of their creation – we have only classes.
- Also a class exists and it may have attributes and methods!
- These methods don't have implicit access to attributes of a particular object!
- And the attributes are created only once for the whole class.
- So we obtain something like global variables.

Further specialities of object programming II/II

instance-creating, a.k.a., what means word `static` in front of `Main`

- Function `Main` is static, i.e., it "lives" in the class (not in particular object)!
- Thus the whole program "lives" in the class we defined (and not in some object).

Item protection

inside the object or class

- Some items (attributes) it is better to protect from incompetent use.
- Inside the object (class) we believe one another,...
- ... outside the object we may believe mainly our ancestors (later).
- Keywords `private`, `public` and `protected` define which items can be accessed only from the object (class), by everyone or by our ancestors (classes that inherited from us), respectively.
- Example:

```
public static void Main(){...}
private int x=1;//inaccessible from outside
```

Larger example

```
class x{
     public int a; private int b;
}
class y{
     public static void Main(){
          x.a=10;//this is OK,
          Console.WriteLine(x.b);
               //this is K.O.
}
```

Input processing

...pars prima

- Function names similar to Pascal, implementation completely different.
- These function are in (static) class `Console` which is in namespace `System`.
- `System.Console.Read()` and `System.Console.ReadLine()`.
- Functions for reading the input are returning a (read) value:
- `char x=Console.Read();`
- `ReadLine` returns string (up to the end of line/input).
- How to read something else?
- Write it yourself...

Or find some library functions...

conversion

- int x=int.Parse(Console.ReadLine());
- int y=Convert.ToInt32(Console.ReadLine());
- Class `Convert` can be used for converting into more data-types.

Writing the output

- Function System.Console.WriteLine
- is overloaded, i.e., may print out many data-types.
- At the moment we learn just to use the first argument,
- i.e., each piece of output we (temporarily) output separately.
- Later we learn something very similar to Pascal, but not yet.

Euclid's algorithm

```
using System;
static void Main(string[] args)
{    Console.WriteLine("Gimmi two numbers:");
     int a=int.Parse(Console.ReadLine());
     int b=int.Parse(Console.ReadLine());
     while(a!=b)
          if(a>b) a-=b;
          else b-=a;
     Console.Write("GCD is: ");
     Console.WriteLine(a);
}
```