C#

C++

Very similar to Java

C-Based syntax (if, while,…)

object base class

no pointers, object parameters are references

All code in classes

Before we begin

You already know and have programmed with Java

You already know C/C++

You have at least basic knowledge of an OS

You are able to read basic english

known C# features

instruction set :

C-like operators (both arithmetic and logical) and sequence handling : if…else, switch…case (break), while, do…while, for

basic types : int, char, double

use '.' to access class members (attributes, methods)

known C# features

Java-like features :

use `new` when creating an object

the garbage collector (GC) is responsible for collecting unused memory segments

objects are references (hidden pointers), C# doesn't use pointers ☺

A sample class

```
class toto
{
   private int i;

   public toto()
   {
       i=0;
   }

   static void Main(string[] args)
   {
       toto t = new toto();
   }
}
```

JAVA ?

C# ?

```
public static void main(String args[])
```

attributes and methods

All code is written in classes :

variables are attributes

functions are methods

attirbutes and methods are generically called members

special methods : constructors

- no return type
- may be overriden

constructors

```
class foo
{
    int i;

    public foo()
    {
        i = 0;
    }

    public foo(int x)
    {
        i=x;
    }
}
```

```
class bar
{
    string s;

    public bar():this("default");
    {


    }

    public bar(string t)
    {
        s=t;
    }
}
```

static attributes and methods

static attributes are class attributes : shared by all objects

static methods are class methods

static methods can only access static members

access with class name rather than instance name.

static members and methods

```
class withS
{
    static int x;

    static void set(int i)
    {
        x =i;
    }
}
```

called using withS.set(…)

Accessibility

public, private, protected

private is the default for members

protected members can be accessed from derived classes (see inheritance later)

C# specific elements

how C# handles arrays : `System.Array` class

1-dimensional array : indexed by integers, index range from 0

Arrays may contain objects or variables

the size of an array must be defined before it is used

C# specific elements

declaring an array :

```
type/class [] array_name;
```

examples :

```
int [] tab1;
char [] message;
string [] tabS;
```

C# specific elements

creating an array :

```
array_name = new type/class[size];
```

examples (continued)

```
tab1 = new int[12];
float [] tabFl = new float[5];
```

C# specific elements

creating an array with implicit size definition :

by providing the values for the elements to be stored

```
char [] tabChar = {'a','j','k','m','z'};
```

or

```
char [] tabChar;
tabChar = new char {'a','j','k','m','z'};
```

C# specific elements

multi dimensional arrays

matrix : column size is the same for all rows (Delphi, Pascal)

split array : column size can be different for each row (Java)

C# specific elements

matrix : uses the [, ,] syntax

```
string [,] tab2s = new string[4,2];
```

same syntax for accessing elements

```
string myString = tab2s[2,0];
```

C# specific elements

split arrays : use the [][] syntax

example with a 2D array storing objects from a class named appClass

C# specific elements

```
class appClass
{
   private char c;

   public appClass()
   {
        c='a';
   }

   public affiche()
   {

   System.Console.Wri
   te(c.ToString());
   }
}
```

```
class prog
{
   static void Main(string [] args)
   {
        appClass [][] tab2d;

        int longueur = 8;

        tab2d = new appClass[4][];
        tab2d[0] = new appClass[3];
        tab2d[1] = new appClass[17];
        tab2d[2] = new
               appClass[longueur];
        tab2d[3] = null;

   }
}
```

C# specific elements

the `foreach` instruction : iterates in a collection (interface `ICollection`)

`System.Array` implements the `ICollection` interface

syntax :

```
foreach(type identif in collection)
{
  instructions;
}
```

C# specific elements

```
class test
{
  static void Main(string[] args)
  {
  string [] tab = new string[12];
  // initialize tab

  foreach (string s in tab)
  {
  System.Console.Writeline(s);
  }
  }
}
```

C# specific elements

for multidimensional arrays : elements are listed so that the indexes of the rightmost dimension are incremented first.

Properties

- replaces accessors and mutators
- A property has a name, a type (or class) a `set()` and/or a `get()` method

- A property is :
  - accessed like a member
  - indeed, a method is called : data integrity is maintained

Properties

```
class booh
{
   private int _i;         // member

   public booh()  {_i=0;}  // method

   public int i            // property
   {
       get
       {
             return _i;
       }
       set
       {
             _i = value;
       }
   }
}
```

Properties

```
class arf
{
  public arf()
  {}
  public void f(booh b)
  {
      b._i = 2;   // no, _i is private
      b.i = 2; // i is a public property of b
      // this calls the set method from i
  }
  public void Main(string[] args)
  {
      arf a = new arf();
      a.f(new booh());
  }
}
```

Writing properties

`get` has no return type :

it is the type of the property

```
public type propertyname
```

`set` has no parameters : instead, uses the intrinsic `value` variable storing the value written when calling the property

`get` and `set` take no parenthesis !

Writing properties

`get` and `set` may contain C# code to ensure safe access / mutation

let `count` be a `private int` member of a sample `S` class.

constraint : `count` should be in the 0..100 range.

using a `Count` property

Writing the S class (1)

```
class S
{
  // members
  // properties & methods

  // GOOO !! (application entry point)
  [STAThread]
  static void Main(string[] args)
}
```

Writing the S class (2)

```
class S
{
  private int count;

  public S() {count=0;}

  public int Count // case sensitive fortunately !
  {
      get // no parenthesis, no parameters
      {
            return count;
      } …
  }
}
```

Writing the S class (3)

```
class S
{
  …
      // we are still inside public int Count
  set        // no parenthesis, one intrinsic parameter
  {
      if ((value <0) || (value >100)) // sounds
  familiar to you ??
      {
            count=0:
      }
      else
      {
            count = value;
      }
  }   // syntax highlighting is also done in VS !
}
```

Writing the S class : Main()

```
class S
{ // count and Count already written
   static void Main(string[] args)
   {

   }
}
```

this is not a safe place to test your code !

Main is a (static) method of class S : no privacy !

Writing the S class : Main()

```
class S
{ // count and Count already written
   static void Main(string[] args)
   {
       S myvar = new S();
       myvar.count = -3; // no problem !
       System.Console.Write(myvar.Count.ToString());
       System.Console.Read();
   }
}
```

The program runs and displays : -3

Writing the test Class

```
class S
{}
class Test
{
  static void Main(string[] args)
  {
      S myvar = new S();

      myvar.count = -4 ; // no, not even proposed by
      // the code completion tool !
  }
}
```

Writing the P Class

System calls can be written more quickly with the P class.

P uses only static methods

```
class P
{
  public static void ause()
  {
  System.Console.Read();
  }

  public static void rint(object o)
  {
  System.Console.Write(o.ToString());
  }
}
```

Writing the Main() with P

```
class S
{}
class Test
{
   static void Main(string[] args)
   {
       S myvar = new S();

       myvar.Count = -4 ;

       P.rint(myvar.Count); // Class method

       P.ause(); // Class method
   }
}
```

Operator overloading

C# allows operator overloading

operator is considered static

```
class complex
{
  double re,im;

  public static complex operator+(complex z1,
  complex z2)
  {
      return new complex(z1.re+z2.re,z1.im+z2.im);
  }
}
```

Operator overloading

```
class test
{
  public static void Main(string[] args)
  {
      complex z1,z2,z3;
      z1 = new complex(-1.3,4.2);
      z2 = new complex(2.4,1.0);

      z3 = z1+z2; // complex.operator+(z1,z2);

      P.rintln(z3); // if ToString() is overloaded
                    // in complex class

      z3 = z3+z2+z1; // (z3+z2)+z1;
      // ok, (z3+z2) is a complex object
  }
}
```

Operator overloading

operators `==` and `!=` must both be defined for a class

the following operators cannot be overloaded :

`&& || [] () += -=` …

[] property

[] operator overloading interesting

a special property is called indexer

allows to overload [] usage

example with PersonList class

```
public class PersonList : ArrayList
```

- `method Add(object o);`
- `property Count`
- `[] notation`

Indexer

```
public static PersonList operator+(PersonList l,
   Person p)
{
   l.Add(p);
   return l;
}
public override string ToString()
{
   // build a string from all the objects stored
   // in the PersonList (this)
   (for int i=0; i < this.Count;i++){…}
}
```

Indexer

```
public new Person this[int i]
{
   get{…}
   set{…}
}

get
{
   return (Person)base[i];
}
set
{
   base[i] = value;
}
```

masking

Indexer

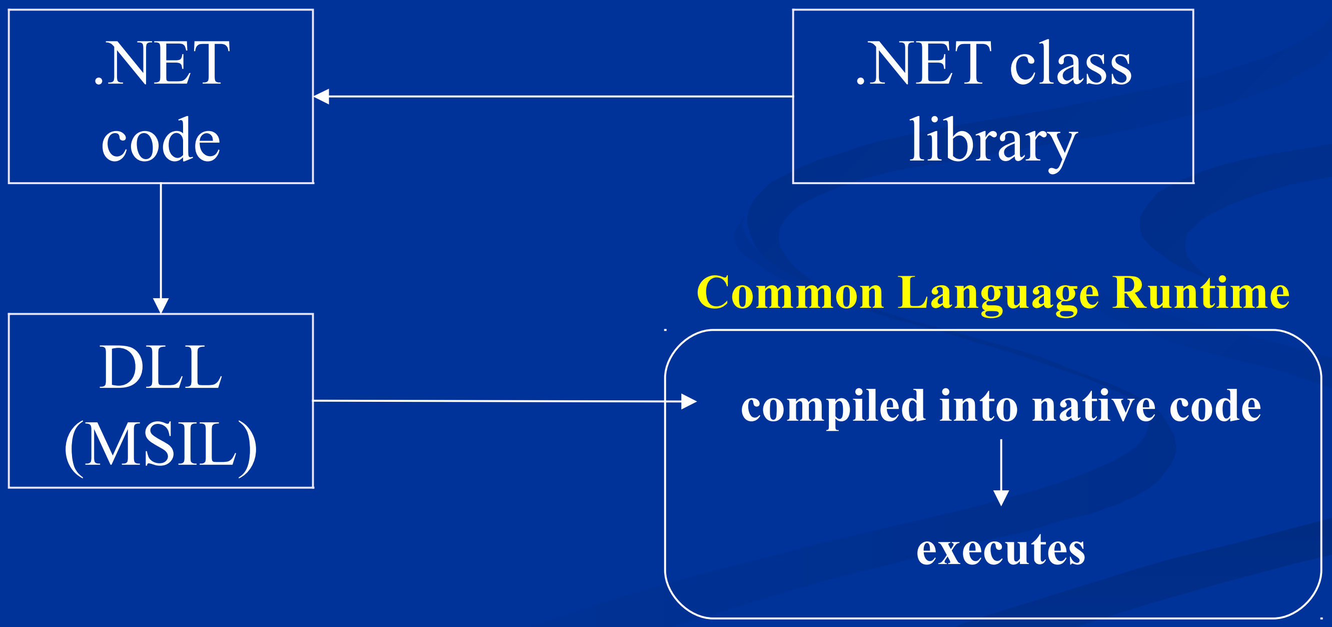
parameter and return types can change

what is the index of a Person having some `name` in the PersonList ?

```
public int this[string nom]
{
   get
   {
        int index=0;
        while((index <Count)&&((Person)base[index].name != name))
        {
                index++;
        }
        if (index == Count)
        {
                index = -1;
        }
        return index;
   }
}
```

.NET environment

2 parts : CLR (execution engine)
class library



.NET code

.NET class library

DLL (MSIL)

Common Language Runtime

compiled into native code

executes

.NET / Visual Studio

- Visual Studio enhances productivity (code behind edition)
- Code and/or graphical Page Design
- Direct HTML coding for Web pages (.aspx)
- DB connectivity through ADO.NET

  preferentially with ORACLE, OleDB, Odbc, Microsoft SQL Server.

Working with assemblies

Compiling a C# program :

console program : use the csc.exe (C Sharp Compiler) to generate an .exe file

not a "true" exe, needs the .NET CLR virtual machine in order to be translated from MSIL to binary (machine language)

this .exe is called an **assembly**

.DLL files

usage : suppose you wrote a test.cs file containing one or more classes

use the VS cmd.exe tool (that good old ugly DOS interface)

```
csc test.cs
```

generates test.exe

.DLL files

in order to generate a .DLL file :

in order to generate a .netmodule file

```
csc /t:library test.cs
```

```
csc /t:module test.cs
```

use .netmodule files to create assemblies containing files from different .NET languages