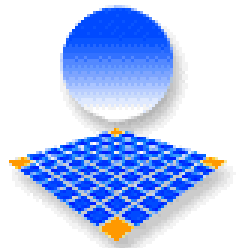

Introduction à C#

Clémentine Nebut
LIRMM / Université de Montpellier 2



Hello world !

Utilisation de l'espace de nom System de la bibliothèque

Main comme Java

Positionnement dans un espace de nom

```
using System;
namespace coursCs
{
    class HelloWorld
    {
        public static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Appel à la classe System.Console (E/S standard)

World.cs

Le nom du fichier peut être différent de celui de la classe

Possibilité de faire des classes partielles (une classe dans plusieurs fichiers)

Types de base

- sbyte, byte
- short, ushort
- int, uint
- long, ulong
- char
- float
- double
- bool
- decimal

Énumération

```
enum Couleur {Bleu, Blanc, Rouge};  
  
public void PrintBlanc(){  
    Couleur c=Couleur.Blanc;  
    Console.WriteLine(c.ToString());  
}
```

- Type sous-jacent : int
 - Bleu=0 ; Blanc =1 ; Rouge =2
 - conversion vers les entiers : int i = (int) c
 - arithmétique (+, -, ...)

- méthode ToString() de System.Object (en java : toString() ...)
- convention de nommage : noms de méthodes commencent par majuscule

Les structures

```
struct Cours{  
    public int VolumeCM;  
    public string Module;  
    public string Enseignant;  
  
    public Cours(int volume, string module, string enseignant){  
        VolumeCM=volume; Module=module; Enseignant=enseignant;  
    }  
}
```

- Même principe qu'une classe
- Est transmis par valeur

- C# est sensible à la casse
- Accessibilité : public, private, protected (classe ou sous-classes), internal (même assembly) , protected internal(protected ou internal)

Les tableaux

```
int[] tab1 = new int[3]; // tableau de rang 1 et de taille 3
tab1[0]=12;

int[,] tab2 ; /* déclaration d'un tableau de rang 2 */
tab2 = new int[2,3] {{1,2,3},{4,5,6}}; // alloc. et init.

for (int l=0;l<tab2.GetLength(0);l++){
    for (int c=0;c<tab2.GetLength(1);c++){
        Console.Write(tab2[l,c]+" ");
    }
    Console.WriteLine();
}
```

- Commentaires // ou /* */
- Console.Write(a+"et"+b+"et"+c) <=>
Console.Write("{0} et {1} et {2}", 'a', 'b', 'c');

- Les tableaux héritent de System.Array
- Indices à partir de 0

Les classes

- Membres = {propriétés, méthodes, champs, indexeurs, etc}
- Attribut <> champs

```
using System;  
namespace coursCs  
{
```

```
    public class Carre  
    {
```

```
        public double Longueur;
```

```
        public Carre(double longueur){  
            Longueur=longueur;  
        }
```

```
        public double Perimetre(){  
            return Longueur*4;  
        }
```

```
        public Carre Scale(double pourcentage){  
            Carre c=new Carre(Longueur*pourcentage);  
            return c;  
        }
```

```
    }  
}
```

Champs

constructeur

méthodes

création d'instance

Les classes partielles

```
partial class ClasseDeTest{  
    // une variable  
    private string maVariable;  
  
    public ClasseDeTest()  
    {  
        // mon constructeur  
    }  
}
```

```
partial class ClasseDeTest{  
    //une methode  
    public void maMethode()  
    {  
        MessageBox.Show(maVariable);  
    }  
}
```


Visibilité des classes

- **public**
 - classe visible par n'importe quel programme d'un autre namespace
- **protected (*)**
 - classe visible seulement par toutes les autres classes héritant de la classe conteneur de cette classe.
- **internal**
 - classe visible seulement par toutes les autres classes du même assembly.
- **Private (*)**
 - classe visible seulement par toutes les autres classes du même namespace
- Par défaut
 - si classe interne : private
 - sinon : public

(*) : pour les classes internes

Implémentation et héritage

- Pas de différence syntaxique entre l'héritage et l'implémentation (\neq Java)
- Héritage simple
- Implémentation multiple
- Classe non extensible : sealed (= final java)
- Référence à la superclasse : base (équivalent du super de Java)

Classes abstraites

- Contient au moins une méthode abstraite
- Peuvent contenir des attributs et des méthodes concrètes
- Ne peut pas être instanciée
- Méthode abstraite
 - méthode virtuelle, sans corps : mot clef `abstract`
 - `public` ou `protected`
 - définie lors d'un héritage, avec le mot clef `override`

Les classes abstraites

```
public abstract class IForme2{
    public string Nom;
    public abstract double Perimetre();
    public abstract double Aire();
    public void Print(){
        Console.WriteLine("Mon nom est {0}, mon aire est {1}", Nom, Aire());
    }
}
```

```
public class Rectangle : IForme2{
    public double Longueur, Largeur;
    public Rectangle(string nom, double longueur, double largeur){
        Longueur=longueur;
        Largeur=largeur;
        Nom=nom;
    }
    public override double Perimetre(){
        return 2.0*(Longueur+Largeur);
    }
    public override double Aire(){
        return Longueur*Largeur;
    }
}
```

Les interfaces

```
public interface IForme
{
    double Perimetre();
    double Aire();
}
```

```
public class CarreImplemente : IForme
{
    double Longueur;
    public CarreImplemente(double longueur)
    {
        Longueur=longueur;
    }
    public double Perimetre(){
        return Longueur*4;
    }
    public double Aire(){
        return Longueur*Longueur;
    }
}
```

- Convention : les noms d'interface commencent par I
- Que des signatures de méthode

Appel aux éléments de la superclasse

- Constructeur
 - `public void Etudiant(int age):base(age)`
- Appel de méthode, accès aux membres et propriétés
 - `base.meth()`
 - `base.prop`

Masquage : new (mot-clef facultatif)

```
class BaseClass{  
    public void Method1(){  
        Console.WriteLine("Base -  
Method1");  
    }  
    public void Method2(){  
        Console.WriteLine("Base -  
Method2");  
    }  
}
```

```
class DerivedClass : BaseClass{  
    public new void Method2(){  
        Console.WriteLine("Derived  
- Method2");  
    }  
}
```

```
class Program{  
    static void Main(string[] args) {  
        BaseClass bc = new BaseClass();  
        DerivedClass dc = new DerivedClass();  
        BaseClass bcdc = new DerivedClass();
```

```
        bc.Method1(); // Base – Method 1  
        bc.Method2(); // Base – Method 2  
        dc.Method1(); // Base – Method 1  
        dc.Method2(); // Derived – Method 2  
        bcdc.Method1(); // Base – Method 1  
        bcdc.Method2(); } } // Base – Method 2
```

Redéfinition d'une méthode virtuelle : override

```
class BaseClass{  
    public virtual void Method1(){  
        Console.WriteLine("Base -  
Method1");  
    }  
    public void Method2(){  
        Console.WriteLine("Base -  
Method2");  
    }  
}
```

```
class DerivedClass : BaseClass{  
    public override void Method1(){  
        Console.WriteLine("Derived - Method1");  
    }  
    public new void Method2(){  
        Console.WriteLine("Derived -  
Method2");  
    }  
}
```

```
class Program{  
    static void Main(string[] args) {  
        BaseClass bc = new BaseClass();  
        DerivedClass dc = new DerivedClass();  
        BaseClass bcdc = new DerivedClass();
```

```
        bc.Method1(); // Base – Method 1  
        bc.Method2(); // Base – Method 2  
        dc.Method1(); // Derived – Method 1  
        dc.Method2(); // Derived – Method 2  
        bcdc.Method1(); // Derived – Method 1  
        bcdc.Method2(); } } // Base – Method 2
```


Visibilité des membres et méthodes

- **par défaut (aucun mot clef)**
 - private
- **public**
 - visibles par toutes les classes de tous les modules.
- **private**
 - visibles que dans la classe.
- **protected**
 - visibles par toutes les classes incluses dans le module, et par les classes dérivées de cette classe.
- **Internal**
 - visibles par toutes les classes incluses dans le même assembly.

Les membres

- Peuvent être :
 - constants (const)
 - en lecture seule pour les clients (readonly)
 - associés avec des accesseurs (-> propriété)
 - partagés par toutes les instances (static)

Les propriétés

- Même syntaxe de définition qu'un attribut
- Fonctionnement par invocations de 2 méthodes d'accès internes : get et set

```
public class Accesseurs{  
    public double Longueur; //champs  
    public double Aire {    // propriété  
        get {return Longueur*Longueur;}  
        set {Longueur=Math.Sqrt(value);}  
    }  
    public void Test(){  
        Aire=9; // appel de set  
        Console.WriteLine(Longueur); // écrit 3  
        Longueur=2;  
        Console.WriteLine(Aire); // écrit 4  
    }  
}
```

Propriétés et constructeurs

```
public class Jouet{  
    private double _prix; //champs  
    public double prix {    // propriété  
        get {return _prix;}  
        set {_prix=value;}  
    }  
}  
  
public class Test{  
    public void Test(){  
        Jouet buzz=new Jouet(prix=12);  
    }  
}
```

Propriétés auto-implémentées

```
using System;
```

```
public class Customer  
{  
    public int ID { get; set; }  
    public string Name { get; set; }  
}
```

```
public class UtilisationCustomer  
{  
    static void Main()  
    {  
        Customer cust = new Customer();  
  
        cust.ID = 1;  
        cust.Name = "Jean Dupond";  
  
        Console.WriteLine(  
            "ID: {0}, Name: {1}",  
            cust.ID,  
            cust.Name);  
  
        Console.ReadKey();  
    }  
}
```

Le passage de paramètres

- Passage :
 - par valeur (pour les types de base)
 - par référence (pour les objets, ou mot clef ref)

```
public void Echanger(ref int a, ref int b)
{
    int temp=a;
    a=b;
    b=temp;
}

public void Echanger2(int a, int b)
{
    int temp=a;
    a=b;
    b=temp;
}
```

```
public void Test(){
    int a=1;
    int b=2;

    Echanger2(a, b);
    Console.WriteLine("a={0} et b={1}", a, b); //a=1 et b=2

    Echanger(ref a, ref b);
    Console.WriteLine("a={0} et b={1}", a, b); //a=2 et b=1
}
```

Paramètres de sortie

```
public void Foo(out int x){  
    x=1900;  
}  
  
public void Test2(){  
    int a;  
    Foo(out a);  
    Console.WriteLine(a); // affiche 1900  
}
```

Les opérateurs

```
public class Coordonnees2D{
    public double X;
    public double Y;

    public Coordonnees2D(double x, double y){
        X=x; Y=y;
    }

    public static Coordonnees2D operator + (Coordonnees2D c1,
                                             Coordonnees2D c2){
        return new Coordonnees2D(c1.X+c2.X, c1.Y+c2.Y);
    }
}
```

```
Coordonnees2D c1=new Coordonnees2D(1,2);
Coordonnees2D c2= new Coordonnees2D(3,4);
c1=c1+c2;
Console.WriteLine("C1 = ({0}, {1})", c1.X, c1.Y); // c1=(4,6);
```


Les indexeurs

```
public double this[int index]{  
    get {  
        if (index==0) return X;  
        else if (index==1) return Y;  
        else return 9999.9999;  
    }  
    set {  
        if (index==0) X=value;  
        else if (index==1) Y=value;  
    }  
}
```

```
Coordonnees2D c1=new Coordonnees2D(1,2);  
Console.WriteLine("C1 = ({0}, {1})",c1[0], c1[1]); // c1=(1,2);
```

- Ne peuvent pas être static (membres de classe)

Les conditionnelles

- `if (cond) {...} else {...}`
- `switch (expr) {`
 - `case valeur1 : ... ; break;`
 - `case valeur2 : ... ; break;`
 - `...`
 - `default : ...;``}`
- Rq : `break` obligatoire

Les boucles

- `while(cond) {...}`
- `do {...} while (cond);`
- `for (init ; tantQueCond ; incrément){...}`
- `break;`
- `continue;`

Foreach

- Enumération de collections et de tableaux
- `foreach (type identificateur in expression)
instructions`
 - `type` : type de l'identificateur
 - `identificateur` : la variable d'itération
 - `expression` : collection d'objets ou tableau. Le type des éléments de la collection doit pouvoir être converti en le type de l'identificateur. Implémente l'interface `IEnumerable` ou déclare une méthode `GetEnumerator`.
 - `instructions` : les instructions à exécuter

Foreach : exemple

```
public int[] QuatreEntiers=new int[4]{10,8,6,4};
public Carre[] TroisCarres=new Carre[3]{new Carre(1),
                                         new Carre(2),new Carre(3)};

public void Test(){
    foreach (int c in QuatreEntiers){
        Console.WriteLine(c);
    }

    foreach (Carre c in TroisCarres){
        Console.WriteLine(c.Longueur);
        c.Longueur=c.Longueur+1;
    }

    foreach (Carre c in TroisCarres){
        Console.WriteLine(c.Longueur);
    }
}
```

Yield return

- pour retourner un itérable
- [http://msdn2.microsoft.com/en-us/library/65zzykke\(VS.80\).aspx](http://msdn2.microsoft.com/en-us/library/65zzykke(VS.80).aspx)

```
public System.Collections.IEnumerator GetEnumerator()  
{  
    for (int i = 0; i < max; i++)  
    {  
        yield return i;  
    }  
}
```

```
public System.Collections.IEnumerator GetEnumerator()  
{  
    yield return "With an iterator, ";  
    yield return "more than one ";  
    yield return "value can be returned";  
    yield return ".";  
}
```

Yield

- Utilisé dans un bloc itérateur pour fournir une valeur à l'objet énumérateur ou signaler la fin de l'itération.
 - yield return <expression>;
 - yield break;

```
// yield-example.cs
using System;
using System.Collections;
public class List
{
    public static IEnumerable Power(int number, int exponent)
    {
        int counter = 0;
        int result = 1;
        while (counter++ < exponent)
        {
            result = result * number;
            yield return result;
        }
    }
    static void Main()
    {
        // Display powers of 2 up to the exponent 8:
        foreach (int i in Power(2, 8))
        {
            Console.Write("{0} ", i);
        }
    }
}
```

Les exceptions

```
class MonException : Exception {}

class ExceptionTest {
    public static void Main (String [] args){
        int i = 0;
        try{
            while (true){afficheA10 (i++);}
            Console.WriteLine ("jamais exécuté ici");
        } catch (MonException e){
            /* e.StackTrace : pile des appels de méthodes
             * e.Message : le message associé à l'exception
             * e.InnerException : en cas d'exception imbriquée */ }
            //si on n'attrape pas MonException, une erreur est levée à l'exec
        finally{Console.WriteLine ("Instr. du finally tjs exécutées");}
        Console.WriteLine ("bloc exécuté que si l'erreur est attrapée");
    }
    private static void afficheA10 (int n){
        if (n > 10) throw new MonException();
        Console.WriteLine (n);
    }
}
```

On ne déclare pas les exceptions pouvant être levées (pas de throws)

Conversion de type et typage

- Conversion de types comme en java :

Type2 y = ...;

Type1 x = (Type1) y;

- Conversion avec instruction as :

Type1 x = y as Type1;

– affecte *null* si conversion impossible

- Test de l'appartenance à un type : instruction *is*
if (x is Type1) ...

Boxing / unboxing

- Boxing : prendre un type primitif et le faire tenir dans un objet (int dans Integer en Java)
- En C# le boxing et unboxing est automatique

int i;

Object o_i =i;

Console.WriteLine((int) o_i);

La documentation

- Même principe que la javadoc :
 - introduction de commentaires spéciaux dans le code
 - ex : `/// <summary>`
 - outil de génération de doc
 - `csc /doc:maDoc.xml maClasse.cs`
- La doc est générée en XML et pas en HTML
- Microsoft fournit une feuille de style par défaut

Les tags de documentation

- `<summary>` petite description `</summary>`
- `<remark>` grosse description `</remark>`
- `<param name="xxx">`descrip. param`</param>`
- `<returns>`desc. de ce qui est retourné`</returns>`
- `<exceptions cref="xxx">`desc. `</exception>`
- `<example>`ex`</example>`
- `<c>` ou `<code>` code C# `</c>` ou `</code>`
- `<see cref="url"></see>`
- `<seealso cref="url"></seealso>`

NDOC

- Projet NDOC, ndoc.sourceforge.net
- Permet de générer aux formats :
 - javadoc
 - LaTeX
 - HTML
 - ...

Les métadonnées (attributs)

- Similaire aux annotations Java
- Informations ajoutées en tête d'un élément de code
- Syntaxe : [attribut1, attribut2, ...]
- Exemples :
 - [WebMethod] indique que la méthode est un service web
 - [StaThread] indique qu'une méthode s'exécute dans le même espace mémoire en cas de thread

Définition d'attributs personnalisés

```
using System.Reflection;
public class Author: Attribute
{
    using System;
    public readonly string name;
    public Author(string name)
    {
        this.name = name;
    }
    public override String ToString()
    {
        return String.Format("Author: {0}", name);
    }
}
```

```
[Author("Damien Watkins")]
public class CPoint: Point
{
    public static void Main()
    {
        MemberInfo info = typeof(CPoint);
        object[] attributes =
            info.GetCustomAttributes();
        Console.WriteLine("Custom Attributes are:");
        for (int i = 0; i < attributes.Length; i++)
        {
            System.Console.WriteLine("Attribute "
                + i + ": is " + attributes[i].ToString());
        }
    }
}
```

Délégation

```
public class C{ // on déclare le type de délégué où l'on veut
    public delegate void Le_delegate (int a, string b); // déclaration
}
public class delegation{ //classe utilisant le délégué
    // association entre le type de délégué et le délégué effectif
    public static C.Le_delegate Affichage=new C.Le_delegate(D2.Meth2);
    public static void Main (String [] args){
        Affichage(7,"toto");Console.ReadLine();
    }
}
public class D1{
    public static void Meth1(int x, string y){
        Console.WriteLine("l'entier : {0} puis la chaine : {1}",x,y);}
}
public class D2{
    public static void Meth2(int x, string y){
        Console.WriteLine("la chaine : {0} puis l'entier : {1}",y,x);}
}
```

- Permet de passer en paramètre l'adresse d'une méthode

Les événements

- Mécanisme abonnement/notification
- Le producteur d'événements
 - déclare et produit des événements
- Le consommateur
 - s'abonne à des événements
- Quand un producteur produit un événement
 - tous les abonnés sont notifiés

Et bien d'autres choses ...

- Classiques
 - Introspection
 - Threads
 - Nunit
 - Contrats
- Moins classiques
 - Linq
 - Pointeurs et code unsafe
 - Typage dynamique
- ...

Contrats

Pré-conditions

Post-conditions

Invariants

Intégration possible à VS

Contrats

```
using System;
using System.Diagnostics.Contracts;

// An IArray is an ordered collection of objects.
[ContractClass(typeof(IArrayContract))]
public interface IArray
{
    // The Item property provides methods to read and edit entries in the array.
    Object this[int index]
    {
        get;
        set;
    }

    int Count
    {
        get;
    }

    // Adds an item to the list.
    // The return value is the position the new element was inserted in.
    int Add(Object value);
}
```

Contrats - suite

// Removes all items from the list.

```
void Clear();
```

// Inserts value into the array at position index.

// index must be non-negative and less than or equal to the

// number of elements in the array. If index equals the number

// of items in the array, then value is appended to the end.

```
void Insert(int index, Object value);
```

// Removes the item at position index.

```
void RemoveAt(int index);
```

```
}
```

Contrats – suite suite

```
[ContractClassFor(typeof(IArray))]  
internal abstract class IArrayContract : IArray  
{  
    int IArray.Add(Object value)  
    {  
        // Returns the index in which an item was inserted.  
        Contract.Ensures(Contract.Result<int>() >= -1);  
        Contract.Ensures(Contract.Result<int>() < ((IArray)this).Count);  
        return default(int);  
    }  
    Object IArray.this[int index]  
    {  
        get  
        {  
            Contract.Requires(index >= 0);  
            Contract.Requires(index < ((IArray)this).Count);  
            return default(int);  
        }  
        set  
        {  
            Contract.Requires(index >= 0);  
            Contract.Requires(index < ((IArray)this).Count);  
        }  
    }  
}
```

Contrats suite suite suite

```
public int Count
```

```
{  
    get  
    {  
        Contract.Requires(Count >= 0);  
        Contract.Requires(Count <= ((IArray)this).Count);  
        return default(int);  
    }  
}
```

```
void IArray.Clear()
```

```
{  
    Contract.Ensures(((IArray)this).Count == 0);  
}
```

```
void IArray.Insert(int index, Object value)
```

```
{  
    Contract.Requires(index >= 0);  
    Contract.Requires(index <= ((IArray)this).Count); // For inserting immediately after the end.  
    Contract.Ensures(((IArray)this).Count == Contract.OldValue(((IArray)this).Count) + 1);  
}
```

Contrats - fin

```
void IArray.RemoveAt(int index)
{
    Contract.Requires(index >= 0);
    Contract.Requires(index < ((IArray)this).Count);
    Contract.Ensures(((IArray)this).Count == Contract.OldValue(((IArray)this).Count) - 1);
}
```


Linq – Aperçu

```
class IntroToLINQ
{
    static void Main()
    {
        // The Three Parts of a LINQ Query:
        // 1. Data source.
        int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

        // 2. Query creation.
        // numQuery is an IEnumerable<int>
        var numQuery =
            from num in numbers
            where (num % 2) == 0
            select num;

        // 3. Query execution.
        foreach (int num in numQuery)
        {
            Console.WriteLine("{0,1} ", num);
        }
    }
}
```

Quelques références

- Documentation C# de Microsoft
- Cours C# :
<http://rmdiscala.developpez.com/cours/livres/LivreBases.html#csharp>