ESIEE – Paris L. Buzer & B. Perret Oct 2011 v1.04

INTRODUCTION AU LANGAGE C# ET A LA PROGRAMMATION OBJET
Résumé de cours

Programme du module (30H)

Connaissances

- CLR et .net
- Présentation bibliothèque .net - namespace
- Langage & syntaxe
- Type valeur & type objet référencé
- String
- Tableaux 1D, 2D, nD
- Clonage/duplication
- POO : classe & instance
- POO : variables et méthodes de classe et d'instance
- Connaissance des différents « Class members » du C#
- Constructeur par défaut / sans argument / paramétrique
- Mécanisme des boucles (while, for, foreach)
- Passage et retour de paramètres
- Notion de bloc & sous bloc / conflit de nom et masquage

Compétences :

- Organisation d'un projet C# avec l'outil Visual
- Mise en place d'une application fenêtrée
- Utilisation des composants IHM de .net
- Gestion des évènements
- Savoir utiliser et comprendre la documentation MSDN
- Manipulation basique des bitmaps et des images
- Les outils de dessin « drawing » de .net

SONT HORS-PROGRAMME pour ce module

- Destructeur
- Switch case / break / do while
- Variables et fonctions globales – classe utilitaire
- Formatage d'affichage
- Structure
- Entrée / Sortie fichier
- Mot clef : out
- Déclaration des propriétés (getter / setter)
- Déclaration des méthodes et variables de classe
- Libération de ressources – dispose

VOC : symbole, mot, acronyme, mot-clef ou syntaxe à connaître pour l'examen.

Introduction

Petite généalogie des langages : FORTRAN, BASIC, ALGOL, COBOL, PASCAL, C, SIMULA, ADA, Objective C, C++, Visual C++, DEPLHI, JAVA, .net, C#, Kernighan Ritchie, Stroustrup

VOC : IDE (EDI), RAD, DOD, POO, IBM, ATT, GUI, IHM, MSDN

Les fichiers de projets

VOC : extensions csproj, cs et sln

Le Framework .net

VOC : CLR, IL, CTS, JIT, CLS

Principaux packages

System.IO, System.Data, System.Drawing, System.Math, System.Windows.Form, System.SQL, System.SQL, System.Printing, System.Speech, System.Text, System.Timers

Espace de noms

VOC : namespace, using

```
namespace Projet
{
  …
}
```

```
using System;
Console.Write("Bonjour");
```

VOC: espace de nom, instruction, type, constante, operateurs, identificateurs (ID), variable, fonction, classe, qualificatif, expression

Un **identificateur** correspond au nom donné à une fonction, à une variable ou à une classe. Une **expression** est un regroupement d'opérateurs, de constantes et d'identificateurs.

VOC : opérateur unaire, binaire

Syntaxe autorisée et interdite pour les identificateurs

Les deux types de variables en C #

Type - **valeur** - nombre entier, nombre flottant, caractère, structure et les références
Type - **objet référencé** – tous les autres

Syntaxe pour la création d'un type valeur :

TYPE ID; ou **TYPE ID = val_init ;**

Exemple de création de type valeur :

```
int a ;
float t ;
int b = 3 ;          // inclut la déclaration de la variable et son initialisation
Bitmap Img ;         // pas d'objet Bitmap créé, juste une référence vers Bitmap
```

Une variable de type valeur peut être créée sans être initialisée. CEPENDANT, elle ne peut être utilisée sans avoir été initialisée précédemment.

Syntaxe pour la création d'un type objet référencé :

new ID_CLASSE(<parametres>) ;

L'instruction new retourne une adresse sur l'objet créé qui doit être stockée dans une référence. La référence peut aussi être déclarée au même moment – ce n'est pas obligatoire. Les paramètres fournis dépendent du constructeur choisi, il peut n'y avoir aucun paramètre comme plusieurs.

ID_CLASSE ID_ref = new ID_CLASSE(<paramètres>) ;

Exemple : Bitmap IMG = new Bitmap(400,200);

Dans cet exemple, il y a création d'une référence (type valeur) recevant la position en mémoire d'un nouvel objet de type Bitmap. Il y a deux créations : la référence et l'objet référencé puis association de la référence à cet objet par l'opérateur « = ».

Les variables d'instance (les champs) d'un objet sont AUTOMATIQUEMENT initialisés lors de sa création. Les numériques prennent ainsi la valeur 0 par défaut.

Remarque : toute création d'objet référencé obéit à ces règles. Cependant, pour les objets très usités comme les tableaux et les strings, des écarts ont pu être effectués dans la norme du langage.

Remarque : pour un numérique, la valeur d'initialisation par défaut est zéro. Pour un booléen, cette valeur est à false. Pour une référence, un mot clef spécial a été créé pour l'occasion, il s'agit du mot clef : **null** indiquant que la référence ne pointe vers aucun objet.

Les différents types valeurs

Entiers : byte, short, int, long
    Equivalence avec System.Int32 …
Booléen : bool
Réels : float, double, decimal
Caractère : char

Opérateurs de conversion : int a = **(int)** b ;

Les types non-signés ne sont pas compatibles CLS.

Type par défauts des constantes : 3 – 3.0 – 1.2f

Les objets string

Exception à la règle de déclaration des objets référencés :

string t = «Bonjour » ; et non string t = new string(« bonjour ») ;

Les objets string sont IMMUTABLE, c'est-à-dire qu'ils ne peuvent pas être modifiés après leur création. Tout appel d'une méthode d'un objet string (ToLower par exemple) retourne la référence sur un nouvel objet portant le résultat.

Opérateurs de concaténation +

La console

Différence entre application console et application fenêtrée.

Fonctions :

```
Console.Write(« Bonjour ») ;
Console.WriteLine(« Bonjour ») ;
Console.Write(« resulats {0} {1} {0} »,a,b) ;
```

Les objets tableaux

Création : **int [] T = new int [10] ;**
Création & initialisation : **int [] T = { 1, 5, 6, 7 } ;**

Tableaux multidimensionnels : **int [,] T = new int [3,4] ;**
Int [,,] T = new int [2,4,7] ;

Accès aux valeurs : T[2]

Les tableaux sont de type objets référencés, la valeur de chaque cellule est initialisée par défaut.

Possibilité de faire une initialisation dynamique à partir d'un paramètre :

int [] T = new int [taille] ;

L'accès en dehors d'un tableau déclenche automatiquement une erreur à l'exécution.

Logique de fonctionnement des différents types

Type : objet référencé	**Type : valeur**
int [] T1 = new int[100] ; T1[0] = 7 ; int [] T2 = T1 ;	T2[0] = 8 Aff(T1[0]) => 8

```
Aff(T2[0])  => 8
```

```
int M ;
M = 1 ;
int P = M ;
```

```
P = 2 ;
```

```
Aff(M)   => 1
Aff(P)   => 2
```

Duplication (clonage,recopie) des tableaux

Choix 1 : **int [] T2 = new int[T1.Length] ;**
T1.CopyTo(T2,0) ;

Choix 2 : int [] T2 = (int []) T1.Clone()

La gestion de la mémoire

VOC : ramasse-miettes, garbage collector

Un objet n'ayant plus aucune référence associée devient inaccessible et donc inutilisable. Cet objet est alors inscrit dans la liste d'attente des objets à détruire, mais il n'est pas encore détruit. Le système, lorsqu'il le décide, va détruire les objets dans cette liste et enfin libérer les ressources associées, notamment la place occupée en mémoire. Les stratégies choisies par les garbage collector sont diverses et variées. Certains, si la mémoire le permet, attendent la fin du programme pour détruire tous les objets.

HP : Libération de ressource – dispose

Remarque : par convention, un objet qui détient des ressources doit implémenter une fonction **dispose**(). En particulier, les objets ayant des ressources non managées, c'est-à-dire non gérées par le mécanisme du garbage collector, doivent implémenter la fonction dispose(). Par exemple, l'ouverture en lecture d'un fichier par un objet bloque l'accès en écriture à ce fichier. Il faut implémenter la fonction dispose() si vous êtes le développeur de cet objet et dans tous les cas, il faut appeler explicitement la méthode dispose() dans le programme. En effet, la méthode dispose() n'est pas appelée par le garbage collector. De toute façon, l'objet peut avoir un besoin momentané de lire un fichier et il n'est pas concevable qu'il bloque l'accès au fichier après cela. Ce n'est pas acceptable.

Les classes et les objets (instances)

Une classe est une description, un modèle, un plan, un moule ou encore un schéma de construction pour les objets à créer. Une classe n'existe pas physiquement en mémoire (exception faite des statics). Seuls les objets allouent des ressources à chacune de leur création.

Syntaxe :

```
Class Voiture
{
        public int couleur ;
        public int puissance ;
        public void Demarrer() ;
        public void Arreter() ;
}
```

```
Voiture Car1 = new Voiture() ;
Voiture Car2 = new Voiture() ;
```

La classe est unique. Il peut cependant y avoir entre 0, 1 ou n objets créés durant l'exécution du programme.

Variable et fonctions de classe et d'instance

Une variable propre à un objet (la couleur, l'immatriculation de la voiture) sont qualifiées de variable d'instance. Entre deux objets voitures, ces paramètres peuvent être différents. Une fonction ayant un effet sur un seul objet uniquement (demarrer) est qualifiée de fonction/méthode d'instance. Cette fonction est appelée depuis une instance particulière Car1.Demarrer(). Elle n'influe que sur le comportement de Car1, les autres objets ne subissent aucune modification. Elle peut cependant accéder aux variables de classe car ces variables sont accessibles par tous les objets de la classe.

Une variable partagée par toutes les instances (le nombre de voitures créées au total, la date de mise en service de la première voiture, la TVA sur les vehicules), est alors qualifiée de variable de classe. Il serait très maladroit de positionner cette variable comme variable d'instance, car elle serait dupliquée autant de fois que ce qu'il y a d'instances et sa valeur serait identique dans chaque instance, sa place est donc dans un niveau supérieur, celui de classe.

Une méthode de classe est une méthode propre à la classe. Elle n'a pas accès aux paramètres des différentes instances. Elle ne les voit normalement pas. Une méthode de classe peut sembler être un concept abstrait pour les débutants. L'exemple type est la fonction « constructeur » des classes. Cette fonction est appelée lors de la création d'un objet, elle le « fabrique », retourne une référence au programme et oublie complètement l'existence de cet objet.

Les variables et les méthodes de classe existent dès le lancement du programme, c'est-à-dire, avant la création du premier objet, même si aucun objet est crée ou même si aucune instance n'existe au moment de l'appel. Les variables et les méthodes d'instance sont fortement associés au mot clef « static » car ce qualificatif indique dans plusieurs langages (C++, Java, C#) la déclaration d'une variable de classe ou d'une méthode de classe.

Remarque : les clases sont très proches des types (int float) conceptuellement. On peut d'ailleurs dire que la définition d'une classe correspond à la définition d'un nouveau type. Ainsi, « int » ne sert pas à représenter une variable. Dans l'écriture « int a », la variable « a » est instance d'un « int ». Le type « int » indique le modèle choisi pour la variable « a ». Ainsi la variable « a » va modéliser un entier. Cette proximité se retrouve dans la syntaxe du langage où beaucoup de similarités apparaissent lors de l'utilisation d'un type ou d'une classe, par exemple lors de la création d'un tableau.

Les membres de classes (class members)

Dans la MSDN, il faut savoir lire la documentation sur une classe. Les différents membres d'une classe sont regroupés en plusieurs catégories

VOC :

- variable de classe (champ statique, static field)
- méthode de classe (méthode statique, static method)
- variable d'instance (champ d'instance, instance field)
- propriétés d'instance
- méthode d'instance (méthode d'instance, instance method)
- constructeurs
- opérateurs (+ entre deux vecteurs)
- indexeurs (implémente l'opérateur [])

- évènements (events) exemple : les composants IHM gèrent des events

Une propriété masque généralement l'accès à une variable d'instance que l'on cherche à protéger. Par exemple, il serait maladroit de mettre une longueur à -1 directement en modifiant la variable size, length, height… Un couple de fonctions getter/setter sert d'intermédiaire et sont appelées indirectement. Elles vérifient alors l'intégrité du paramètre et effectuent les modifications nécessaires dans la variable d'instance associée. Pour faciliter l'écriture des programmes, les propriétés se manipulent comme des variables d'instance. Les appels au getter / setter sont ainsi implicites.

HP : destructeur, syntaxe getter, setter

Constructeur

```
class Cercle
{
        public int x = 0 ;              // init explicite       // constructeur par défaut
        public int y ;                  // init implicite

        public Cercle()                                      // constructeur sans argument
        {
                x = 10 ;
                y = 10 ;
        }

        public Cercle(int _x, int _y)                        // constructeur paramétrique
        {
                x = _x ;
                y = _y ;
        }
}
```

Le constructeur sans argument et les constructeurs paramétriques sont OPTIONNELS. De toute façon, un constructeur par défaut est fourni par C# en tenant compte des initialisions explicites et en fournissant des initialisations implicites pour tous les autres variables d'instance.

Cas 1 :

Cercle C = new Cercle() ;

- ⇨ Appel du constructeur par défaut
- ⇨ Appel du constructeur sans argument – **S'IL EXISTE**

Cas 2 :

Cercle C = new Cercle(10,10) ;

- ⇨ Appel du constructeur par défaut
- ⇨ Appel du constructeur paramétrique

Le constructeur par défaut est TOUJOURS appelé. Il n'est pas obligatoire de fournir un constructeur sans argument même s'il semble explicitement appelé lors du new. Le langage considère que le constructeur par défaut est alors suffisant. L'utilisation d'un constructeur paramétrique implique par contre qu'il soit explicitement programmé dans le code.

Le constructeur doit porter le même nom que la classe et ne doit pas être précédé d'un type de retour (surtout pas de void).

Dans la littérature, plusieurs définitions pour le constructeur par défaut et le constructeur sans argument se mélangent. Vous pouvez donc trouver des différences par rapport à ce qui est présenté ici.

Instructions

Instructions & mécanismes : if, while, for, foreach

Passage de paramètres

Cas1 : type valeur

```
void fnt1()
{
        int a = 7 ;
        fnt2(a) ;
        Aff(a) ; // -> 7
}
```

```
fnt2(int u)
{
        u++ ;
        Aff(u);  // u -> 8
}
```

Lors de l'arrivée dans fnt2, une variable « u » dite « locale » (car locale à cette fonction) est créée. La valeur de « a » (nous rappelons que « a » est du type valeur, les choses sont bien faites !) est recopiée dans « u ». Les variables « a » et « u » sont indépendantes et désignent physiquement deux cellules mémoires distinctes. A la fin de fnt2(), « u » est détruite. Les modifications effectuées sur « u » n'ont en rien affectées « a ».

Cas 2 : type objet référencé

```
void fnt1()
{
        int [] T = { 1, 4 ,5 } ;
        fnt2(T) ;
        // à ce niveau T : { 8, 4, 5}
}
```

```
fnt2(int [] u)
{
        u[0] = 8 ;
}
```

Nous rappelons que « T » est une référence et donc un type valeur. Ainsi, lors de l'arrivée dans fnt2, une nouvelle variable locale « u » est créée et le contenu de « T » est recopié dans « u ». Ainsi, ces deux références, qui correspondent à deux variables distinctes en mémoire, désignent le même objet tableau. Elles ont toutes deux pour valeur l'adresse du tableau en mémoire. Ainsi la modification faite à partir de « u[0] = 8 » s'applique au tableau déclaré dans fnt1. La variable « u » est détruite à la fin de fnt2 mais cela n'implique pas la destruction de l'objet tableau car il existe toujours la référence « T ».

Remarque : dans fnt2, si l'on veut faire des modifications dans un tableau initialisé à partir des valeurs de T sans modifier le contenu de T, il faut alors passer par une procédure de clonage de T.

Paramètres de retour

Cas 1 : mot clef – ref –

```
void fnt1()                          void fnt2(ref int b)
{                                    {
        int a = 7 ;                          b++ ;
        fnt2(ref a) ;                        aff(b) ; // ->8
        aff(a) ;   // -> 8            }
}
```

En mettant le qualificatif « ref » devant la déclaration de « b », cela interdit la création d'une nouvelle variable locale. Ainsi, la variable « b » dans fnt2 correspond à la variable « a » de fnt1 qui a juste été renommée.

Cas 2 : mot clef – return –

De manière tout à fait équivalente à la précédente :

```
void fnt1()                          int fnt2(int b)
{                                    {
        int a = 7 ;                          b++;
        a = fnt2(a) ;                        return b;
        aff(a) ;   // -> 8           }
}
```

Remarque : dans le cas où plus d'un paramètre doit être retourné (par exemple x et y), il faut alors utiliser la syntaxe « ref » du cas 1.

HP : out

Portée des variables

Variable locale :
- variable passée en paramètre d'une fonction
- variable déclarée à l'intérieur d'une fonction
- variable déclarée dans le corps d'un for( )

Une variable locale a sa portée d'utilisation qui commence à sa déclaration et qui finit à la fin de son bloc. Elle ne peut donc être utilisée à l'extérieur de son bloc, d'où le caractère « local ». Dans le corps d'une boucle for : « for (int i = 0 ; i < 10 ; i++) », la variable "i" appartient au bloc sous-jacent associé au « for ». Une fois sortie de la boucle for, cette variable n'existe plus.

Variable d'instance (instance field) : toutes les méthodes d'instance (pas les méthodes de classe) ont accès à ces variables. Ces variables naissent avec l'objet et disparaissent à sa destruction.

Variable de classe (static field) : tous les membres de classes et d'instance ont accès à ces variables, à tous les niveaux de leurs blocs. Ces variables naissent en début de programme (lancement) et sont détruites à la fin.

Règle sur les conflits de noms

Aucune variable ne peut être utilisée sans être une variable d'instance ou une variable locale déclarée dans ce bloc ou un bloc supérieur.

Lorsqu'il y a conflit de nom entre une variable locale et une variable d'instance, la variable locale l'emporte et masque la variable d'instance. Pour contourner le masquage et utiliser la variable d'instance, il faut utiliser le mot clef this qui désigne l'instance courante : **this.variable_dinstance**

Règle du C# : il ne doit pas y avoir risque de masquage et de confusion entre deux variables de même nom dans des blocs de niveaux DIFFERENTS. Cette règle ne tient pas compte de la position de la déclaration dans le bloc.

```
void fnt1()
{
        int a ;
        {
                int a ;
        }   // ERREUR
}
```

```
void fnt2()
{
        {
                int a ;
        }
        int a ;  // erreur
}
```

```
void fnt3()
{
        {
                int a ;
        }
        {
                int a ;  // OK
        }
}
```