



LES BASES DU C#

François-Xavier BAILLET



En vrac

- En V5, rien de plus vs. V4 (Majeur pour les méthodes asynchrones : await, async & les "caller information")
- C# est un langage orienté objet classique avec :
 - Données
 - Instructions
 - Et ... attributs et méthodes pour les objets
- Le C# 6
 - Révolution => Open Source (.NET Core)
 - Syntaxe allégée pour les initialiseurs et le constructeurs, dans le langage
 - Quelques gestions simplifiées des types et opérateurs.
- Le C# 7 (actuel / .NET Framework 4.7.1 / VS 2017 -15.5)
 - switch case sur des type objet
 - pattern

```
if (obj is int i || (obj is string s && int.TryParse(s, out i)) /* use i */)
```
 - tuple { var sum = (5,20) }
 - références sémantiques

```
var a = ref FourthElement(myArray)
a = 10; //MyArray[4] now equals 10
```

C#7 : Pattern, structures de contrôles, tuples

```
switch(shape){  
    case Circle c:  
        WriteLine($"circle with radius {c.Radius}");  
        break;  
    case Rectangle s when (s.Length == s.Height):  
        WriteLine($"{s.Length}x{s.Height} square");  
        break;  
    case Rectangle r:  
        WriteLine($"{r.Length}x{r.Height} rectangle");  
        break;  
    default:  
        WriteLine("<unknown shape>"); break;  
    case null:  
        throw new ArgumentNullException(nameof(shape));}
```

```
(string, string, string) LookupName(long id) // tuple return type  
{ ...  
// va retirer first, middle et last du stockage de données  
return (first, middle, last); // littéral du tuple  
}
```

Toujours en vrac

- Accepte les caractères accentués
- Reste standard avec C++
- On comprend l'anglais, on reconnaît les mots clefs 😊
- Un coup d'œil à Visual Studio

Les types de données

- C# utilise les types de données suivants :
 - Les nombres entiers
 - Les nombres réels
 - Les nombres décimaux
 - Les caractères et chaînes de caractères
 - Les booléens
 - Les objets

Les types de données prédéfinis

Type C#	Type .NET	Donnée représentée	Suffixe des valeurs littérales	Codage	Domaine de valeurs
char	Char (S)	caractère		2 octets	caractère Unicode (UTF-16)
string	String (C)	chaîne de caractères			référence sur une séquence de caractères Unicode
int	Int32 (S)	nombre entier		4 octets	[-2 ³¹ , 2 ³¹ -1] [-2147483648, 2147483647]
uint	UInt32 (S)	..	U	4 octets	[0, 2 ³² -1] [0, 4294967295]
long	Int64 (S)	..	L	8 octets	[-2 ⁶³ , 2 ⁶³ -1] [-9223372036854775808, 9223372036854775807]
ulong	UInt64 (S)	..	UL	8 octets	[0, 2 ⁶⁴ -1] [0, 18446744073709551615]
sbyte		..		1 octet	[-2 ⁷ , 2 ⁷ -1] [-128, +127]
byte	Byte (S)	..		1 octet	[0, 2 ⁸ -1] [0, 255]
short	Int16 (S)	..		2 octets	[-2 ¹⁵ , 2 ¹⁵ -1] [-32768, 32767]
ushort	UInt16 (S)	..		2 octets	[0, 2 ¹⁶ -1] [0, 65535]
float	Single (S)	nombre réel	F	4 octets	[1.5 10 ⁻⁴⁵ , 3.4 10 ⁺³⁸] en valeur absolue
double	Double (S)	..	D	8 octets	[-1.7 10 ⁺³⁰⁸ , 1.7 10 ⁺³⁰⁸] en valeur absolue
decimal	Decimal (S)	nombre décimal	M	16 octets	[1.0 10 ⁻²⁸ , 7.9 10 ⁺²⁸] en valeur absolue avec 28 chiffres significatifs
bool	Boolean (S)	..		1 octet	true, false
object	Object (C)	référence d'objet			référence d'objet

Ci-dessus, on a mis en face des types C#, leur type .NET équivalent avec le commentaire (S) si ce type est une structure et (C) si le type est une classe. On découvre qu'il y a deux types possibles pour un entier sur 32 bits : *int* et *Int32*. Le type *int* est un type C#.

On remarquera donc que pour les types commençant par une majuscule, on manipule des structures ou des objets, donc par référence

Les constantes

- `#define` différent du C++
 - Ne définit pas une constante mais déclare un symbole au préprocesseur (`#if`, `#else`,...)
- Mot clé « `const` »
 - `const type NOM = valeur;`
 - `const float VAL_PI = 3.1416;`
- **Ne pas négliger les constantes**
 - ➔ Pour la lisibilité du code
- Une règle simple : en majuscule

Les déclarations

- Déclaration de variables :
 - identificateur_de_type variable [= valeur];
 - int unEntier = 12;
 - int unEntier;
- Version .Net 3.5 : Mot clé « var »
 - La variable prend le type de la valeur
 - Initialisation obligatoire
 - var uneVariable = 12;
 - Utile quand le type n'est connu que du compilateur

```
var maReq = from clients in tournées  
            where clients.ville == "Tours"  
            select new {clients.ref, clients.phone};
```

Les structures

- Quand utiliser les structures ?
 - Pour de petites quantités de données
 - Accès très rapide en mémoire
 - Pas de notion d'héritage
 - Initialisation implicite des champs impossibles
 - Copie par valeurs et non par référence

```
struct Personne  
{  
    public String Nom;  
    public int Age;  
}
```

```
Personne p = new Personne();  
p.Age = 5;  
Personne q = p;  
La valeur de q.Age est à 5  
Si on fait q.Age=7, p.Age toujours à 5.
```

Les énumérés

- Permet d'avoir une énumération de valeurs
`enum Mention {Passable, AssezBien, Bien, TrèsBien, Excellent};`
`Mention ma_mienne = Mention.Passable;`
`ma_mienne = 1 ; ma_mienne = 4;`
`Enum.GetValues(ma_mienne.GetType())=> "Excellent"`
`Enum.GetValues(typeof(Mention)) => les valeurs`
- Plusieurs valeur d'un enum pour une variable (bits indicateurs)
`[flags]enum Mention {Passable, AssezBien, Bien, TrèsBien, Excellent}`
`Mention A=Mention.Passable|Mention.Bien`

Conversion number↔string

- nombre – chaîne : nombre.ToString()
- chaîne – int : int.Parse(chaîne) ou System.Int32.Parse(chaîne)
- chaîne – long : long.Parse(chaîne) ou System.Int64.Parse(chaîne)
- chaîne – double : double.Parse(chaîne) ou System.Double.Parse(chaîne)
- chaîne – float : float.Parse(chaîne) ou System.Float.Parse(chaîne)
- Classe Convert (Convert.ToInt32, Convert.ToDouble...)
- Gestion des erreurs :

```
try {  
    appel d'une fonction de conversion  
} catch (Exception ex){  
    traitement de l'exception ex  
}
```
- Le code démo conversion

Les Tableaux

- Rassemblement de données de même type
 - Type[] nom_tableau = new Type[n];
 - Int[] entiers = new int[5];
 - Int[] entiers = new int[]{1,2,3,4,5};
 - Int[] entiers = {1,2,3,4,5};
 - entiers.Length : nombre d'éléments dans le tableau
- Tableau à 2 dimensions
 - Type[,] nom_tableau = new Type[n,m];
 - nom_tableau.Rank : nombre totale de dimensions du tableau
 - Nom_tableau.GetLength(i) : dimension correspondant à l'indice i
- Tableau de tableaux
 - Type[][] nom_tableau = new Type[n][];

Instructions élémentaires

- Affectation : variable = valeur;
- Opérateurs des expressions arithmétiques (+, -, *, /, %)
- $a=a+b \Leftrightarrow a+=b$; $i++ \Leftrightarrow i=i+1$;
- $i=1$; $a=++i$; a vaut 2 sinon, $i=1$; $a=i++$; a vaut 1 et i vaut 2;
- Fonctions mathématiques (Classe Math) :
 - double Sqrt(double x) : Racine carrée
 - double Cos(double x) : Cosinus
 - double Sin(double x) : Sinus
 - double Tan(double x) : Tangente
 - double Abs(double x) : Valeur absolue
 - double Exp (double x) : Exponentielle
 - double Pow (double x, double y) : x puissance y ($x>0$)
 - double Log (double x) : Logarithme népérien
 - etc.
- Expressions relationnelles :
 - Opérateurs : <, <=, ==, !=, >, >=
 - Priorité opérateurs :
 - » 1°) <, <=, >, >=
 - » 2°) ==, !=
- Expressions booléennes :
 - Opérateurs : AND (&&), OR (||;|)*, NOT (!)
 - Résultat true ou false

*{ x || y ; y évalué ssi x false}

Instructions élémentaires

- Comparaison de chaînes :

“Chat” < “Chien”

Car ‘C’=‘C’; ‘h’=‘h’; ‘a’<‘i’ → selon ordre Unicode

- Traitements des bits

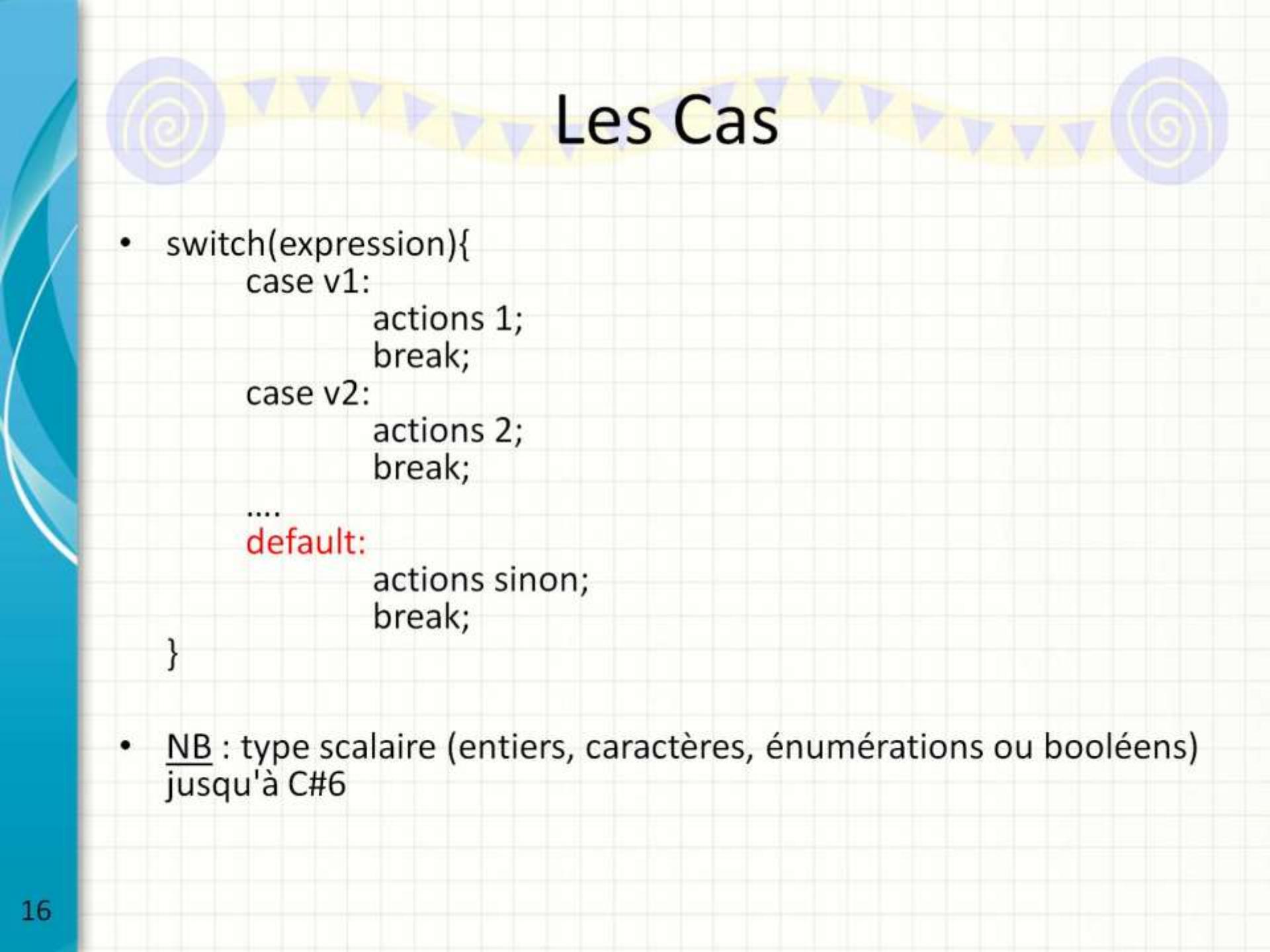
Soient i et j deux entiers.

- $i \ll n$ décale i de n bits sur la gauche. Les bits entrants sont des zéros.
- $i \gg n$ décale i de n bits sur la droite. Si i est un entier signé (signed char, int, long) le bit de signe est préservé.
- $i \& j$ fait le ET logique de i et j bit à bit.
- $i | j$ fait le OU logique de i et j bit à bit.
- $\sim i$ complémente i à 1
- $i \wedge j$ fait le OU EXCLUSIF de i et j

Choix simple et opérateur ternaire ?

```
if (condition 1) {  
    action(s) condition 1 vraie;  
}  
else if (condition 2) {  
    action(s) condition 2 vraie;  
}  
else {  
    action(s) condition 1 et 2 fausses;  
}
```

- `expr_cond ? expr1:expr2` est évaluée de la façon suivante :
 - ✓ l'expression `expr_cond` est évaluée. C'est une expression conditionnelle à valeur vrai ou faux
 - ✓ Si elle est vraie, la valeur de l'expression est celle de `expr1` et `expr2` n'est pas évaluée.
 - ✓ Si elle est fausse, c'est l'inverse qui se produit : la valeur de l'expression est celle de `expr2` et `expr1` n'est pas évaluée.
- L'opération `i=(j>4 ? j+1:j-1);` affecte à la variable `i` : `j+1` si `j>4`, `j-1` sinon.
- C'est la même chose que d'écrire `if (j>4) i=j+1; else i=j-1;` (plus concis)



Les Cas

- ```
switch(expression){
 case v1:
 actions 1;
 break;
 case v2:
 actions 2;
 break;

 default:
 actions sinon;
 break;
}
```
- NB : type scalaire (entiers, caractères, énumérations ou booléens)  
jusqu'à C#6

# Les répétitions

- ```
for (index=nb; index<nb2; index++){  
    actions;  
}
```
- ```
foreach (Type variable in collection){
 actions;
}
```
- ```
while (condition){  
    actions;  
}
```
- ```
do {
 actions;
} while(conditions)
```
- Code boucle
- break : fait sortir des boucles → **A éviter sauf Foreach**
- continue : fait passer à l'itération suivante

# Les exceptions

- ```
try {  
    instructions pouvant générer exception;  
}  
catch (TypeException1 e1){  
    gestion de l'exception de type 1;  
}  
catch (TypeException2 e2){  
    gestion de l'exception de type 2;  
}  
catch (Exception ex){  
    gestion des autres exception;  
}  
finally { // toujours exécutée  
    exécution après try ou catch  
}
```
- exception.Message : message de l'exception
exception.StackTrace : Pile d'exécution de l'exception
- Code exception

Les filtres d'exceptions : C#6

```
try
{
    using ( car stream = new FileStream ( path ,
                                         FileMode.OpenOrCreate,
                                         FileAcces.Write ) )
    {
        ...
    }
}
catch ( IOException ex )
{
    if ( ex.HResult == 0x80070020 )
    {
        // File is in use by another process
        // Handle that case...
    }
    else if ( ex.HResult == 0x80070035 )
    {
        // Network path was not found
        // Handle that case...
    }
    else
    {
        // Something else happened
        // Handle that case...
    }
}
catch ( Exception ex )
{
    // Something else happened
    // Handle that case...
}
```

```
try
{
    using ( car stream = new FileStream ( path ,
                                         FileMode.OpenOrCreate,
                                         FileAcces.Write ) )
    {
        ...
    }
}
catch ( IOException ex ) if ( ex.HResult == 0x80070020 )
{
    // File is in use by another process
    // Handle that case...
}
catch ( IOException ex ) if ( ex.HResult == 0x80070035 )
{
    // Network path was not found
    // Handle that case...
}
catch ( Exception ex )
{
    // Something else happened
    // Handle that case...
}
```

Intérêt :

C'est plus clair

On ne rentre que dans le catch concerné et
on peut relancer une exception
"interceptable" par les catchs suivants.

Passage de paramètres

- Public static void Main (String[] args)
 → Tableau de chaînes en paramètre
- Pas de pointeurs (en mode géré, il faut passer en mode unsafe) → Pas/Peu utilisé en C#
- Passage par valeurs (défaut)
 - public static void Change_age(int age) {}
- Passage par références
 - public static void Change_age(**ref** int age) {}
- Out : pas de valeur initiale, juste en sortie
 - public static void Change_age(out int age) {}
- Code tableau de références

Passage de paramètres (C#4)

- Arguments par défaut :
 - void fonction (int num, int var=12){}
 - void type “vide”, “nul”
 - fonction (15,16);
 - fonction (15);
- Arguments nommés :
 - Donner le nom à l’argument pour changer sa position ➔ fonction (var:58, num:102);

Programmation Objet (POO)-Rappels

- Un objet est une entité qui contient
 - des données qui définissent son état (on les appelle les champs, **attributs**)
 - des fonctions (on les appelle les **méthodes**)
- Un objet est créé selon un modèle que l'on appelle une **classe**
- A partir d'une classe on peut instancier (créer) de nombreux objets

Programmation Objet (POO)-Rappels

- Définition d'une classe \equiv Définition d'un nouveau Type de données
- Accès :
 - private : champ privé, accessible uniquement par les méthodes internes de la classe
 - public : champ public, accessible par toute méthode définie ou non dans la classe
 - protected : champ protégé, accessible uniquement par les méthodes internes de la classe ou d'une classe dérivée (héritage)
- En général, les données d'une classe sont déclarées privées alors que ses méthodes et propriétés sont déclarées publiques. Cela signifie que l'utilisateur d'un objet (le programmeur)
 - n'aura pas accès directement aux données privées de l'objet
 - pourra faire appel aux méthodes publiques de l'objet et notamment à celles qui donneront accès à ses données privées.
- Création d'un objet : new
- Désignation de l'objet courant : this

La classe C# : syntaxe par l'exemple

```
public class Personne {  
    // attributs de classe  
    private static long nbPersonnes;  
    // méthode associée  
    public static long Nb_Personnes {  
        get{ return nbPersonnes; }  
    }  
  
    // attributs d'instance  
    private string prenom;  
    private string nom;  
    private int age;  
  
    // constructeurs = méthode avec nom de la classe  
    // appelée à la création de l'objet  
    public Personne (String p, String n, int age ) {  
        Initialise(p, n, age);  
        nbPersonnes++;  
    }  
    public Personne(Personne p) {  
        Initialise(p);  
        nbPersonnes++;  
    }  
  
    // méthode pour initialiser l'objet Personne dont les  
    // attributs sont privés  
    public void Initialise(string p, string n, int age) {  
        this.prenom = p;  
        this.nom = n;  
        this.age = age;  
    }  
    // légal car paramètres différents  
    public void Initialise(Personne p) {  
        prenom = p.prenom;  
        nom = p.nom;  
        age = p.age;  
    }  
}
```

```
// propriétés  
public string Prenom {  
    get{ return prenom; }  
    set{ // prénom valide ?  
        if(value== null || value.Trim().Length == 0) {  
            throw new Exception("prénom ("+ value+ ") invalide");  
        } else{  
            prenom = value;  
        }  
    } //if  
} //prenom  
  
public string Nom {  
    get{ return nom; }  
    set{ // nom valide ?  
        if(value== null || value.Trim().Length == 0) {  
            throw new Exception("nom ("+ value+ ") invalide");  
        } else{ nom = value; }  
    }  
} //nom  
  
public int Age {  
    get{ return age; }  
    set{ // age valide ?  
        if(value<= 0) {  
            age = value;  
        } else  
            throw new Exception("âge ("+ value+ ") invalide");  
    } //if  
} //age  
  
// méthode  
public void Identifie() {  
    Console.WriteLine("[{0}, {1}, {2}]", prenom, nom, age);  
}
```

La classe C# : syntaxe par l'exemple

- Personne p1;
`p1.Initialise("Jean","Dupont",30); //légal mais....`
- L'objet n'existe pas donc :
`Personne p1=new Personne();`
- On peut aussi :
`Personne p1=new Personne("Jean","Dupont",30)`
- p1 référence l'objet ("Jean","Dupont",30)
- Si p1=null; l'objet est perdu
- `New Personne(new Personne("Jean","Dupont",30)).Identifie();`
→ Objet temporaire, mémoire récupérée par le garbage collector (besoin évaluation d'expression par exemple)



NB : le `p1.Dispose()` c'est parfois mieux... (class IDisposable)

La classe C# : syntaxe par l'exemple

- Accès aux attributs privés ?

```
// accesseurs  
public String GetPrenom() {return prenom;} ...  
  
// modifieurs  
Public void SetPrenom(String P){this.prenom=P;}...
```

→ Lourd non ?

C#6 :
public String Prenom{get;set} = "Pierre";

- Utilisation des propriétés

```
public string Prenom {  
    get{ return prenom; }  
    set{ // prénom valide ?  
        if(value== null || value.Trim().Length == 0) {  
            throw new Exception("prénom (" + value + ") invalide");  
        } else {  
            prenom = value;  
        }  
    } //if  
} //prenom
```

- Console.Out.WriteLine("p=("+ p.Prenom + "," + p.Nom + "," + p.Age + ")");
p.Prenom → get
- p.Age = 56; → set
- Attributs de classe, dans l'exemple permet de compter le nb d'instance => déclaration en **static**

Héritage

- Principe identique au C++
- Pas d'héritage multiple
- Syntaxe :

```
class ClasseFille : ClasseMere
```

```
{  
}
```

- La classe fille n'hérite pas des constructeurs
=> les redéfinir
- Code héritage

Les méthodes d'extension C#3.0

- Possibilité d'ajouter des fonctionnalités à des classes sans avoir accès au code source
 - Déclarer une classe **static** « **NomClasse** »
 - Déclarer fonction static d'extension
 - Premier paramètre = objet courant
 - Impossible si les classes sont « **sealed** »
(Héritage interdit..)
- Pose pas mal de problèmes dans le versionning

Les méthodes d'extension C#3.0

- Extension de la classe System.String :

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str) /* indique l'extension */
        {
            return str.Split(new char[] { ' ', '.', '?' },
                            StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

- Mise à portée par `using ExtensionMethods;`
- Utilisée :

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

- Conclusion : peu conseillé, relecture de code difficile, perte de compatibilité avec .NET 2.0 (conséquent)

Les comparaisons d'objets

- Définir le sens de la comparaison...
- “==” → compare les références d'objets
- Méthode Equals() hérité de la classe mère
Object compare par défaut... les références
- ⇒ **spécialiser la méthode dans la classe fille**

Les comparaisons d'objets : exemple

- Soit une classe voiture :

```
public class Voiture {  
    public string Couleur { get; set; }  
    public string Marque { get; set; }  
    public float Conso { get; set; }  
}  
  
Voiture A_Car = new Voiture();  
Voiture A_Car .Couleur= "Bleue";  
Voiture B_Car = A_Car;  
Voiture B_Car .Couleur= "Vert";  
if (A_Car == B_Car) // C'est vrai  
    { Console.WriteLine("Les objets référencent la même instance"); }  
  
Voiture A_Car = new Voiture();  
Voiture A_Car .Couleur= "Bleue";  
Voiture B_Car = new Voiture();  
Voiture B_Car .Couleur= "Bleue";  
if (A_Car == B_Car) // C'est faux  
    { Console.WriteLine("Les objets référencent la même instance"); }  
else {  
    // Les deux objets ne référencent pas la même instance  
}
```

Il faut définir la comparaison pour les voitures sur les attributs

La classe voiture devient :

```
public class Voiture {  
    public string Couleur { get; set; }  
    public string Marque { get; set; }  
    public float Conso { get; set; }  
  
    public override bool Equals(object obj) {  
        Voiture v = obj as Voiture;  
        if (v == null) return false;  
        return Conso == v.Conso &&  
               Couleur == v.Couleur &&  
               Marque == v.Marque;  
    }  
}
```

Utilisation :

```
Voiture A_Car= new Voiture { Conso = 5.5F, Marque =  
                           "Peugeot", Couleur = "Grise" };  
Voiture B_Car = new Voiture { Conso = 5.5F, Marque =  
                           "Peugeot", Couleur = "Grise" };  
if (A_Car.Equals(B_Car)) {  
    Console.WriteLine("Les objets ont les  
    mêmes valeurs de propriétés");  
}
```

YES!



Classe Abstraite

- Ne peut pas être instanciée. Il faut créer des classes dérivées qui elles pourront être instanciées.
- On peut, par exemple, utiliser ces classes pour factoriser le code d'une lignée de classes.
- Déclaration :

```
abstract class MaClasseAbstraite{...}
```
- Dérivation :

```
public class maClasse : MaClasseAbstraite{...}
```

Classe Abstraite : exemple

```
class Exception UtilisateurInconnu:Exception{  
    public ExceptionUtilisateurInconnu(string message) : base(message){  
    }  
  
    abstract class Utilisateur {  
        // champs  
        private string login;  
        private string motDePasse;  
        private string role;  
  
        // constructeur  
        public Utilisateur(string login, string motDePasse){  
            // on enregistre les informations  
            this.login = login;  
            this.motDePasse = motDePasse;  
            // on identifie l'utilisateur  
            role=identifie();  
            // identifié ?  
            if (role == null) {  
                throw new  
                    ExceptionUtilisateurInconnu(String.Format("[{0},{1}]",  
                        login, motDePasse));  
            }  
  
            // toString  
            public override string ToString(){  
                return String.Format("Utilisateur[{0},{1},{2}]", login,  
                    motDePasse, role);  
            }  
  
            // identifie  
            abstract public string identifie();  
        }  
    }
```

→ On laisse la main pour la maîtrise des règles d'identification et donc de l'exception gérée

On dérive ...

```
class Administrateur : Utilisateur {  
    // constructeur  
    public Administrateur (string login, string motDePasse)  
        : base(login, motDePasse) {  
    }  
    // identifie  
    public override string identifie() {  
        // override pas nécessaire (classe abstraite)  
        // identification via un SGBD  
        ...  
        return "admin";  
    }  
}
```

Idem du style

```
class Operateur : Utilisateur {  
    ...  
}  
class Inconnu : Utilisateur{  
    ...  
    public override string identifie() {  
        // identification LDAP  
        // ...  
        return null; ← Peut être bien ;-)  
    }  
}
```

Utilisation

```
static void Main(string[] args) {  
    Console.WriteLine(new Observateur("observer","mdp1"));  
    Console.WriteLine(new Administrateur("admin","mdp2"));  
    try{  
        Console.WriteLine(new Inconnu("xx","yy"));  
    }  
    catch(ExceptionUtilisateurInconnue){  
        Console.WriteLine("Utilisateur non connu : "+ e.Message);  
    }  
}
```

Rq : C'est la méthode .ToString qui sert dans la méthode WriteLine

Interface

- Une interface est un ensemble de prototypes de méthodes ou de propriétés qui forment un contrat.
- **Une classe qui décide d'implémenter une interface s'engage à fournir une implémentation de toutes les méthodes définies dans l'interface.**
- Les propriétés et méthodes d'une interface ne sont définies que par leurs signatures. Elles ne sont pas implémentées.
- Déclaration :

```
public interface IMonInterface{...}
```
- Implémentation :

```
public class maClasse : IMonInterface {...}
```
- Utile pour laisser la main mise sur l'implémentation des méthodes
- Une classe dérivée d'une classe interface peut avoir des méthodes statiques, mais sans rapport avec l'interface considérée.
- Schématiquement : Classe abstraite ≈ Interface + une partie du codage

Interface

- Méthode de swap d'entier =>

 Public static void Swap_int (ref int I, ref int J)

- Méthode swap de personne =>

 Public static void Swap_Pers (ref Personne p1, ref Personne p2)

- Idée : classe générique :

```
class Generic1<T> {  
    public static void Swap (ref T value1, ref T value2){...}  
}
```

Utilisée :

```
Generic1<int>.Swap(ref i1, ref i2);  
Generic1<Personne>.Swap(ref p1, ref p2);
```

NB : C'est aussi possible pour la méthode

```
class Generic {  
    public static void Swap<T> (ref T value1, ref T value2){...}  
}
```

Utilisée :

```
Generic1.Swap<int> (ref i1, ref i2);  
Generic1.Swap<Personne> (ref p1, ref p2);
```

Interface

En implémentant une interface :

```
class Generic {  
    public static void Swap<T>(ref T el1, ref T el2) where T : Interface1 {  
        // on récupère la valeur des 2 éléments  
        intvalue1 = el1.Value();  
        intvalue2 = el2.Value();  
        // si 1er élément > 2ième élément, on échange les éléments  
        if (value1 > value2) {  
            T temp = el2;  
            el2 = el1;  
            el1 = temp;  
        }  
    }  
}
```

Avec :

```
Interface Interface1  
{  
    int Value()  
}
```

Interface : plus simple ;-)

```
public interface IAffichable {  
    /* Liste des méthodes que doivent posséder toutes les classes  
       implémentant l'interface IAfffichable : */  
    public void Afficher();  
    public void Afficher(string message);  
}
```

Implémentation :

```
public class Personne : IAfffichable {  
    private string nom, prenom;  
    public void Afficher() { Console.WriteLine(nom+ " "+prenom); }  
    public void Afficher(string message) {  
        Console.WriteLine(nom+ " "+prenom+ " :" +message);  
    }  
}
```

Utilisation :

```
public void Montrer(IAfffichable affichable, string message) {  
    affichable.Afficher(message); }
```

Espaces de Noms et utilisation du FrameWork .NET

- Créer un classe dans un espace de noms

```
Namespace X
```

```
{ class Program ...
```

```
}
```

utilisé en using X;

- Utilisation des assemblys stockées via l'OS dans le GAC (Global Assembly Cache)
- Celles utilisées sont dans le folder "References" sous VisualStudio
- Import de l'assembly : System.Drawing pour les fonctionnalités graphiques.
 - Using System.Drawing; // en entête de fichier
 - Color color = new Color();
 - Evite le System.Drawing.Color()

Le type DELEGATE

- Un délégué (**delegate**) est un type référence qui définit la signature d'une méthode
- Quand il est instancié, un délégué peut faire référence à une ou plusieurs méthodes
- De manière intuitive :
 - **delegate** = un “pointeur” sur une fonction / méthode dans le modèle objet ou une série de fonctions variables
 - analogie avec une callback
- Sert de base pour la gestion des évènements
 - Indispensable pour la programmation d'interface (permet d'être averti d'un évènement)

Le type DELEGATE

- Permet de passer une méthode en paramètre
- On peut affecter un “Delegate” à une variable
- Remarques :
 - Il n'y a pas de distinctions entre “pointeur” de méthode et “pointeur” de fonction
 - Un instance de ‘Delegate’ définit une ou plusieurs méthodes pouvant être ou non statique et/ou retourner une valeur

```
Delegate double Del(double x); // Déclaration
```

```
static void DemoDelegate()
{
    Del Inst_del = new Del(Math.Sin); // Instantiation
    double x = Inst_del(1.0); // Invocation
}
```

Le type DELEGATE

- Chaque 'Delegate' possède sa propre liste d'appel de méthodes
- Opérateur += et -= pour constituer les listes

```
delegate void Evt(int x, int y);

static void F1(int x, int y) {
    Console.WriteLine("F1");
}

static void F2(int x, int y) {
    Console.WriteLine("F2");
}

public static void Main() {
    Evt ma_ft = new Evt(F1);
    ma_ft += new Evt(F2);
    ma_ft(1,2); // F1 et F2 sont appelées
    ma_ft -= new Evt(F1);
    ma_ft(2,3); // Seul F2 est appelée
}
```

Méthodes anonymes

- Définir directement la méthode au niveau du paramètre d'appel → Pas de nom, vie locale
- Pas de variable de type delegate, mais utilisation du mot clé
- Exemple :

```
public void DemoTri ( int [] tableau )
{
    TrierEtAfficher ( tableau , delegate ( int [] leTableau )
                      { Array.Sort(leTableau); }
                    );
}
```

```
Console.WriteLine();
```

```
TrierEtAfficher ( tableau , delegate ( int [] leTableau)
                    {
                        Array.Sort(tableau);
                        Array.Reverse(tableau);
                    }
                );
}
```

Expression Lambda

- Méthode simplifiée pour écrire un délégué
- Exemple :

```
DelegateTri tri = delegate ( int[] mon_tableau) {  
    Array.Sort(mon_tableau)  
};
```

- Devient

```
DelegateTri tri = (mon_tableau)=>{Array.Sort(mon_tableau)};
```

- **Action** (C# 3.5): Déclaration de delegate qui ne retourne rien
 - Action<int> del = n => {DoSomething }; del(5);
- **Func** (C# 3.5): Déclaration de delegate qui retourne une valeur
 - Func<int, String> del = n=>{return n.ToString();} del(5);
- Evite la définition du delegate pour passage via lambda expression

Les évènements : Events

- Permet à un objet de prévenir un ensemble d'objets en attente d'un évènement
- **Très utilisé pour les IHM**
- Mémo de mise en place:
 1. Définir la classe qui génère l'évènement et appelle le délégué associé (contiendra l'ensemble des réactions)
 2. Définir la signature de la fonction associée à l'événement comme un délégué
 3. Définir la variable qui contiendra les délégués ou l'évènement
 4. Définir la logique d'émission par appel du délégué
 5. Définir la classe qui traite l'évènement
 6. Définir dans cette classe le traitement de l'évènement
 7. Et ne pas oublier de s'abonner à l'évènement...
- Code associé
- Plus simple : le Feu ☺

Les Threads

- Utilisé pour la programmation asynchrone
- Un programme peut lancer plusieurs unités de traitement, threads en "parallèle"
- Une application est déjà un Thread nommé « Main »
- **A retenir :**
 - Un thread possède sa propre pile (mémoire)
 - Deux threads ne peuvent pas accéder directement aux mêmes éléments graphiques
 - Un thread hérite de la classe Thread
(System.Threading)

Les Threads : Gestion

- Créer un objet Thread, qui prend en argument du constructeur un delegate de type ThreadStart (fonction d'exécution du thread)
 - newThread(new ThreadStart(maFonction))
- **Lancer** le thread : myThread.Start()
- **Attendre** la fin du thread : myThread.Join()
- **Annuler** l'exécution : myThread.Abort()
- Mettre en **pause** : myThread.Suspend()
- **Reprendre** l'exécution : myThread.Resume()
- Mise en **sommeil** programmée myThread.Sleep(ms)

Les Threads : Propriétés

- CurrentThread : propriété statique de l'objet thread indiquant le thread dans lequel le code s'exécute.
- CurrentContext : propriété statique de l'objet thread donnant le contexte du thread en cours d'exécution.
- CurrentUICulture, CurrentCulture : Langue du thread courant
- IsAlive : thread est en cours d'exécution
- IsBackground : thread en arrière plan
- Priority : Priorité d'exécution (Scheduler)
- ThreadState : Etat du thread

Les Threads : Principes

```
public class Action {  
    public Action () {  
        thread = new Thread (new  
            ThreadStart (Activité));  
    }  
    public void Demarrage () {  
        thread.Start();  
    }  
    public void Attente () {  
        thread.Join();  
    }  
    public void Activité() {  
        // code de l'activité  
    }  
}
```

```
public class Action implements Runnable {  
    public Action () {  
        thread = new Thread(this);  
    }  
    public void demarrage () {  
        thread.start();  
    }  
    public void attente () {  
        thread.Join();  
    }  
    public void run(){  
        // code de l'activité  
    }  
}
```

- En C#, utilisation d'un delegate (qui ne prend aucun paramètre et ne rend aucune valeur)

Les Threads : Principes

```
class ClasseUtilisatrice {  
    public static void Main(string[] args) {  
        Action act = new Action ();  
        act.Demarrage ();  
        // D'autres activité peuvent être démarrées ici  
        act.Attente ();  
    }  
}
```

Avec le ""Runnable""

```
public class ClasseUtilisatrice {  
    public static void Main(String[] args) {  
        Action act = new Action ();  
        act.demarrage ();  
        // D'autres activité peuvent être démarrées ici  
        act.attente ();  
    }  
}
```

La classe BackgroundWorker

- Cette classe appartenant à l'espace de noms [System.ComponentModel]
=> permet de simplifier la gestion des Threads
- Elle émet les événements suivants :
 - DoWork : un thread a demandé l'exécution du BackgroundWorker
 - ProgressChanged : l'objet BackgroundWorker a exécuté la méthode ReportProgress. Celle-ci sert à donner un pourcentage d'exécution.
 - RunWorkerCompleted : l'objet BackgroundWorker a terminé son travail. Il a pu le terminer normalement ou sur annulation ou exception.
 - ...
- Peut être utile dans les interfaces graphiques

La classe BackgroundWorker

- Les propriétés associées :
 - **DoWorkEventArgs.Argument** : Argument de type « Object » passé au thread par l'utilisateur dans la fonction RunWorkerAsync
 - **DoWorkEventArgs.Result** : Résultat de type « Object » retourné par le thread par l'utilisateur dans la fonction RunWorkerAsync
 - **DoWorkEventArgs.ProgressPercentage** : Pourcentage d'avancement donné par la fonction ReportProgress
 - ...

BackgroundWorker : Exemple

```
using System;
using System.Threading;
using System.ComponentModel;

namespace BK {
    class Prg {
        // threads
        static BackgroundWorker[] tâches = new BackgroundWorker[5];

        public static void Main() {
            // init Thread courant
            Thread main = Thread.CurrentThread;
            // on fixe un nom au Thread
            main.Name = "Main";

            // création de threads
            for (int idx = 0; idx < tâches.Length; idx++) {
                // on crée le thread n° idx
                tâches[idx] = new BackgroundWorker();
                // on l'initialise
                tâches[idx].DoWork += Sleep;
                // s'exécute à la fin du thread
                tâches[idx].RunWorkerCompleted += End;
                // le thread peut être annulé
                tâches[idx].WorkerSupportsCancellation = true;
                // on le lance
                tâches[idx].RunWorkerAsync(new Data{
                    Numéro = idx,
                    Début = DateTime.Now,
                    Durée = idx + 1 });
            }

            // on annule le dernier thread
            tâches[4].CancelAsync();

            // fin de main
            Console.WriteLine("Fin du thread {0}, tapez [entrée] pour terminer...", main.Name);
            Console.ReadLine();
            return;
        }
    }
}
```

```
// Tous les Threads exécute la méthode Sleep
// Signature standard des évts : sender => émetteur;
// infos->infos sur évènent DoWork (transmission et récupération résultats)
public static void Sleep (object sender, DoWorkEventArgs infos) {
    // on exploite le paramètre infos
    Data data = (Data)infos.Argument;
    // exception pour la tâche n° 3
    if(data.Numéro == 3) {throw new Exception("Exception tâche n° 3");}

    // mise en sommeil pendant Durée secondes avec un arrêt toutes les secondes
    for ( int idx = 1;
          idx <= data.Durée && !tâches[data.Numéro].CancellationPending; idx++) {
        // attente d'1 seconde
        Thread.Sleep(1000);
    }
    // fin d'exécution
    data.Fin = DateTime.Now;
    // on initialise le résultat
    infos.Result = data;
    infos.Cancel = tâches[data.Numéro].CancellationPending;
}

// Exécuté à la fin des threads
public static void End (object sender, RunWorkerCompletedEventArgs infos) {
    // on exploite le paramètre infos pour afficher le résultat de l'exécution
    // info.Error de type Exception renseigné si exception produite
    if (infos.Error != null) {
        Console.WriteLine("Le thread {1} a rencontré l'erreur suivante : {0}",
                          infos.Error.Message, sender);
    } else
        // info.Cancelled (boolean) est à true si thread annulé
        if(!infos.Cancelled) {
            Data data = (Data)infos.Result;
            // données disponibles uniquement qd exécution normal sinon exception
            Console.WriteLine("Thread {0} terminé : début {1:hh:mm:ss}, durée programmée {2} s, fin {3:hh:mm:ss}, durée effective {4}", data.Numéro, data.Début, data.Durée, data.Fin, (data.Fin - data.Début));
        } else{
            Console.WriteLine("Thread {0} annulé", sender);
        }
}

internal class Data{
    // Informations diverses
    public int Numéro { get; set; }
    public DateTime Début { get; set; }
    public int Durée { get; set; }
    public DateTime Fin { get; set; }
}
```

Programmation asynchrone

- Section critique :
 - Bloquer l'accès simultané de threads à une ressource
 - Utilisation du mot clé : lock{}
 - Utilisation de la classe Interlocked, pour effectuer des opérations arithmétiques
 - Utilisation de la classe Monitor
 - Gestion avancée de la section critique par verrous

Programmation asynchrone

- Mutex pour gérer la section critique
- Synchronisation de threads
- Pas forcément du même processus (mutex nommé)

```
new Mutex(flag de blocage,[ « nom du mutex »])
```

```
myMutex.WaitOne();
```

```
myMutex.ReleaseMutex();
```

Programmation multi-cœurs C# v4

- Linq : utilisation de Plinq, extension parallèle
 - Ajout de la fonction .AsParallel()
 - « from n in myCollection.AsParallel()... »
- Module TPL (Task Parallel Library)
 - Lancement de tâches en parallèle avec la classe « Task »
 - Task t = new Task(() => myFunc(« arg »));
 - T.Start();

Programmation multi-cœurs C# v4

- Classe « Parallel »
 - Fournit des méthodes statiques en remplacement de for et foreach
 - Remplacer le mot clé « for » par « Parallel.For »
 - Fournit aussi une surcharge du Invoke pour l'exécution parallèle de fonctions

→ *Quand vous en serez là, vous n'aurez plus besoin de cette présentation*



- Suite du cours :

En vi'a une question...
qu'elle est bonne !



- Quelques notions existantes dans DotNet :
 - Accès aux données
 - WCF & Internet
 - WPF : XAML
 - ➔ des fans, beaucoup en reviennent 
 - ➔ vous verrez en TP, il y a plus performant

