Microsoft® .net™

# C# 3.0

## C# et le monde Objet

Key Consulting

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Niveaux de visibilité (encapsulation)

- **Un type est visible dans la classe où il est défini et :**
- **5 niveaux**
  - **public** : visible de partout
  - **private** : visible que dans la classe
  - **protected** : visible que dans
  - **internal** : visible partout dans l'assemblage où il est définit
  - **internal protected** : visible partout , mais dans un autre assemblage que par les classes dérivées

- **Par défaut un type est privé**

# Niveaux de visibilité (cas particuliers)

- « *protected* » et « *internal protected* » interdits pour les structures
- Un événement ne peut être déclenché que dans son type (interdit aussi pour les types encapsulés)
- Un type non encapsulé ne peut être que « *public* » ou « *internal* »
- Une property et un indexeur peuvent avoir un niveau de visibilité différent

```
public int Prop {
        public get {}
        private set{}
}
```

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Classe : Composition

## 6 sortes de membres :

- Les champs
- Les propriétés
- Les indexeurs
- Les méthodes
- *Les événements*
- Types encapsulés

```
class Foo {
   int _field = 0;
   int Field { get{return _field } }
   int this[int i] { get { return 0; }
   void method(){}
   event EventHandler _event;
   class Foo2 { }
  }
```

**« partial » : permet de définir une classe sur plusieurs fichiers**

# Champs

## Un champs doit être initialisé à sa déclaration ou dans son constructeur

```
class Foo {
  double field_1 = 2.0;
  string field_2;
  int i;
  public Foo()
  { field_2 = « value » }
  void Fct()
  { i++; // Ok car i est initialisé
    int j; j++; // erreur d
   compilation car j non initialisé
}
```

Valeurs par défaut :

- Type de valeur : 0
- Type référence : null

# Champs constants

- **const**
  - Le champs doit être affecté à l’initialisation et ne peux être changé

```
const double pi = 3.14;
```

- **readonly**
  - Le champs peut être affecté à l’initialisation et dans le constructeur et ne peut être changé

```
class Foo {
  readonly double pi = 0.0;
  public Foo() { pi = 3.14; }
}
```

# Properties

- **La syntaxe d’un champs avec les possibilités d’une méthode**
- **« value » du même type que la property**

```
private int _field;

public int Field
{
    get { return _field; }
    set { _field = value; }
}
```

**Property :**

- Lecture / Ecriture
- Ecriture seule
- Lecture seule

# Indexeur (this[ ])

- Considérer l’objet comme un tableau

```
class Foo {
      public int this[] { get {}; set{}; }
}
```

- Paramètre de n’importe quel type

```
public int this[int] { get {}; set{}; }
public int this[string] { get {}; set{}; }
public string this[Foo] { get {}; set{}; }
```

- Plusieurs indexeur par objet
- Paramètres multiples

```
public int this[int,string ]
                { get {}; set{}; }
```

# Methodes : passage par valeur/référence

- **Les arguments d'une méthode doivent être initialisés**
- **Passage par référence (type référence):**
  - pointent sur la même instance
- **Passage par valeur (type valeur):**
  - un clone de l'objet est utilisé par la méthode
- **ref / out : force le passage par référence**

```
static public void Fct(ref int i, out int j) {
    i = 3; j = 3;
}
static void Main() {
        int i = 0; int j;
        Fct(ref i, out j); // i vaut 3 , j vaut 3
}
```

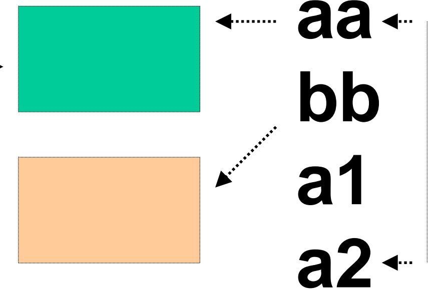
# Methodes : Référence d’une référence

- Utiliser le mot clé **ref** sur une référence permet d’agir directement sur la référence

```
class A{}

class Program  {
  static void Fct(A aa, ref A bb) {
    aa = new A();
    bb = new A();
  }

  static void Main() {
    A a1 = null;
    A a2 = null;
    Fct(a1,  ref a2); // a1 vaut null
                      // a2 référence un objet
  }
}
```



# Méthode : Liste d'arguments variables

- **Mot clé « params »**
  - Permet d'avoir un nombre d'arguments variables
  - Doit être le dernier paramètre
- **Le type des paramètres peut être spécialisé**

```
public void methode ( params Foo[] args );
```

- **L'utilisation de « object » permet d'avoir une liste d'arguments de type variable**

```
public void methode ( params object []args );
```

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Statique (static)

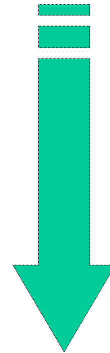
- **Champs, propriétés, méthodes et événements**
  - Membres de classe, membre partagés (!=membres d'instance)
- **Les membres statiques n'ont accès qu'au membres statiques de la classe**
- **Constructeur statique**
- **Classe statique**

```
static class Foo {
  public static Foo1() {}
  public static Foo2() {}
  public static Foo3() {}
}
```

# Statique (Initialisation)

## Ordre d'initialisation des champs d'une classe

1. Valeurs par défaut
2. Valeurs d'initialisation
   - Dans l'ordre de déclaration
3. Constructeurs de la classe



```
Class Foo {
    static int a=b+2;
    static int b=a+3;
    static Foo() {
        a = b+2;
        b = a+3;
    }
}
```

## Constructeur statique :

- Pas d'arguments
- Pas de niveau de visibilité
- Pas de contrôle sur son appel
  - `RunClassConstructor(type t)`

# Les méthodes d'extensions (1/2)

- Permet d'ajouter des méthodes à des types existants sans création de type dérivé.
- Un type particulier de méthode statique
- Définies comme méthodes statiques mais appelées en utilisant la syntaxe de méthode d'instance.

# Les méthodes d’extensions (2/2)

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

```
using ExtensionMethods;

// ---------------------------

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

# Opérateurs (1/3)

- **Opérateurs arithmétiques**
  - Unaires
    - `static Foo operator ++(Foo f)`
  - Binaires
    - `static Foo operator + (Foo f1, Foo f2)`
- **Opérateurs de conversion de type**
  - Implicite
    - `static implicit operator Foo2(Foo1 f)`
      - `Foo2 foo2 = foo1;`
  - Explicite
    - `static explicit operator Foo2(Foo1 f)`
      - `Foo2 foo2 = (Foo2) foo1;`

# Opérateurs (2/3)

- **Opérateur de comparaison liés :**
  - == et !=
  - <= et >=
  - < et >

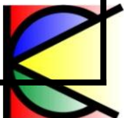
*Provoque une erreur de compilation si l’un existe sans l’autre*

- **Règles par défaut sur les opérateurs**
  - Pas de comparaisons de Structures par défaut
  - Types primitifs : transtypage et comparaison du contenu
  - Pas de comparaison entre un type valeur et un type référence
  - Pas de comparaison autre que l’égalité et l’inégalité sur une référence

# Opérateurs (3/3)

- Un opérateur n’est pas CLSCompliant

| | Binaire | | Unaire | | comparison |
|---|---|---|---|---|---|
| + | Add | + | Plus | == | Equals |
| - | Substract | - | Negate | != | Compare |
| * | Multiply | ++ | Increment | <= | Compare |
| / | Divide | -- | Decrement | >= | Compare |
| % | Mod | ! | Not | < | Compare |
| ^ | Xor | | Transtypage | > | Compare |
| & | BitwiseAnd | Implicite | ToXxx | | |
| \| | BitwiseOr | | FromXx | | |
| << | LeftShift | Explicite | ToXxx | | |
| >> | RightShift | | FromXxx | | |



# Agenda

- **Encapsulation et visibilité**
- Classe :
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Constructeur

- Le compilateur C# fournit un constructeur par défaut si aucun constructeur existe
- Un constructeur est forcément appelé à la construction d'un objet
- Possibilité de créer d'autres constructeurs

```
class Foo {
   public Foo() { }
   public Foo(int i) { }
}
```

- « new » appelle les constructeurs

```
Foo foo1 = new Foo();
Foo foo2 = new Foo(1);
```

# Initialisation rapide

- **Permet d'assigner des valeurs aux propriétés accessibles d'un objet à la création**

```
private class Cat
{
    public int Age { get; set; }
    public string Name { get; set; }
}

static void MethodA()
{
    Cat cat = new Cat { Age = 10, Name = "Sylvester" };
}
```

# Classes encapsulées

- **Définir un type à l'intérieur d'un type (nested type)**
  - Restreindre la visibilité de la classe
  - Classe qui a accès aux types privés de la classe encapsulante

```
public class A {
 private int _cpt = 0;
 private B _b = new B();
 public class B {
  public void m(A  a) {
   a._cpt++;
  }
 }
 private class C {}
}
```

```
A a = new A();
A.B b = new A.B();
b.m(a);  // a._cpt vaut 1
// Erreur de compilation
A.C c = new A.C();
```

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Héritage

- **Syntaxe**
  - `class FooA : FooB { }`
- **`Visibilité`**
  - `Les classes dérivées ont accès aux membres protégés des classes de base`
- **C# supporte**
  - Héritage d’implémentation simple
  - Héritage d’interface multiple
- **« `sealed` » permet d’interdire à une classe d’être une classe de base**

# Héritage : Constructeur

- **Appel au constructeur de la classe de base**
  - implicite
    - Constructeur par défaut

```
public Foo() { }
```

  - explicite
    - Constructeur par défaut
    - Constructeur surchargé

```
public Foo() : base()
 { }
public Foo(int i) : base(i)
 { }
```

- **Appel récursif**
- **Les constructeurs appelés doivent exister**

| class A { | class B : A { |
| --- | --- |
| public A(int i) { } | public B() { } // error |
| } | } |

# Classe : Méthode virtuelle (1/2)

- **Implémentation du concept de polymorphisme**
- **Une méthode virtuelle possède une implémentation et elle n'est pas privée**
- **Un constructeur ne peut être virtuel**
- **Appel à la méthode de la classe mère « base »**
  - « **virtual** »
    - Déclare une méthode comme virtuelle
  - « **override** »
    - Surcharge de la méthode virtuelle
  - « **override sealed** »
    - Arrête la chaîne de polymorphisme
  - « **new** »
    - Casse le mécanisme de polymorphisme

# Classe : Méthode virtuelle (2/2)

## Modèle

```
class A {
 public virtual void m () {
   System.Console.WriteLine(«A.m»);
 }
}

class B : A {
 public override void m () {
   base.m();
   System.Console.WriteLine(«B.m»);
 }
}
class C : B{
 public new void m () {
   base.m();
   System.Console.WriteLine(«C.m»);
 }
}
```

## Appel

```
A[] arrayA = new A[3];
arrayA[0] = new A();
arrayA[1] = new B();
arrayA[2] = new C();

foreach (A a in arrayA) {
  a.m();
}

((C)arrayA[2]).m();
```

```
A.m

A.m
B.m

A.m
B.m
```

## Résultats

```
A.m
B.m
C.m
```

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraite
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Classe abstraite : définition

- **Mot clé : « abstract »**
  - Ne peut être instanciée
  - Possède ou pas de méthodes implémentée
  - Possède ou pas de méthodes abstraites

- **Une classe avec une méthode abstraite doit être abstraite**

```
abstract Foo
{
  abstract public methode();
}
```

# Classe abstraite : dérivée

## Une classe qui dérive d'une classe abstraite est :

- **Abstraite**
  - si elle est déclarée abstraite
  - Si elle n'implémente pas toute les méthode abstraite de la classe mère
  - Si elle déclare de nouvelles méthodes abstraites
- **Concrète** si toutes les méthodes abstraites sont implémentées

```
class A {
}
abstract class B : A {
}
abstract class C : B
{

  abstract public void m();
}
```

```
abstract class D : C
{
  override public void m() {}
  abstract public void m2();
}
class E : D
{
  override public void m2() {}
}
```

# Classe abstraite : méthodes

- « **abstract** » = « **virtual** » **sans implémentation**
  - Suis les même contraintes
  - « override » pour l'implémentation

```
class A {
 public abstract void m () {}
}

class B : A {
 public override void m () {
   System.Console.WriteLine(«B.m»);
 }
}
class C : A{
 public override void m () {
   System.Console.WriteLine(«C.m»);
 }
}
```

```
A[] arrayA = new A[2];
arrayA[0] = new B();
arrayA[1] = new C();

foreach (A a in arrayA) {
   a.m();
}
```

```
B.m
C.m
```

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Interfaces : Définition

```
public interface A {
  void fct1();
}
public interface B : A {
  int Value { get; set; }
}
```

- **Syntaxe : « interface »**
- **Ne peut être instanciée**
- **Visibilité :**
  - « public » ou « internal » pour l'interface
  - « public » pour ses membres
- **Possède ou non :**
  - Méthodes, propriétés, indexeurs ou événements
- **Une interface peut étendre d'autres interfaces**
- **Ne dérive pas de la classe « object »**

# Interfaces : Implémentation

- **Une classe peut implémenter une ou plusieurs interfaces**
- **Une classe qui dérive d'une interface est**
  - ***Concrète*** si tout les membres des interfaces sont implémentées
  - ***Abstraite*** sinon et les champs doivent être abstraites
- **L'implémentation d'un membre peut être virtuelle**

# Interfaces : Implémentation explicite

- **Forcer l'utilisation d'une référence sur l'interface**

```
public interface A {
 void fct1();
}
public class B : A {
 void A.fct1() { }
}
```

```
B b = new B();
b.fct1();  // ne compile pas
( (A) b).fct1();
```

- **Conflits de noms de méthodes sur une interface**
  - Si une classe implémente plusieurs interfaces ayant des signatures de membre identiques :
    - Soit une implémentation unique
    - Soit une implémentation explicite pour chaque interface

# Interfaces : Structure

- Une structure « peut » dériver d'une interface
- Déconseiller à cause du boxing / unboxing

```
interface I {
  void set(int i); int get();
}
struct Struct : I {
  private int _i;
  int get() { return _i; }
  void set(int i) { _i = i; }
}
```

```
Struct s = new Struct();
I i = (I) s;
i.Set(1);
s.Set(2);
```

« I » est sur le tas   « s » est sur la pile

Le boxing fait une « copie » de la structure sur le tas managé

# Is / As

- **Is : détermine si le « Cast » est possible**
  - Permet de déterminer si un transtypage est possible
  - Retourne « true » si le transtypage est possible
  - Si une référence est nulle : retourne « false »
- **As : effectue un « Cast »**
  - Effectue le transtypage si possible
  - Sinon renvoie « null »

# Agenda

- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# enum : Définition

- **Définit un ensemble valeurs**

```
enum Product {Swap, Fra, Cap}
```

- **Peut être définit**
  - Dans une classe , structure ou méthode
  - En dehors d'une classe ou d'une structure
- **Une variable d'un « enum » est un « int »**
  - Par défaut commence à Zéro
  - Peut être redéfinit

```
enum Product {Swap = 10, Fra = 20 , Cap = 30 }
```

  - Permet des opérations arithmétiques

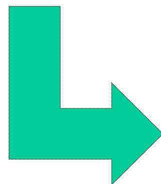
# enum : Indicateur binaire

```
[System.Flags()]
enum FlagFormat {
  Bold   = 0x01,
  Italic = 0x04,
  Under  = 0x10
}
```

Permet de stocker plusieurs informations dans une seule variable

```
FlagFormat flag = FlagFormat.Bold | FlagFormat.Italic;

System.Console.WriteLine(flag.ToString());
if ((flag & FlagFormat.Bold) > 0)
{
        System.Console.WriteLine("Is Bold");
}
```



```
Bold, Italic
Is Bold
```

# enum : Classe Enum

- Possède des méthodes statique utilitaires
- Ex :
  - `string Format(Type type ,object value, string format)`
    - Convertit une valeur d'un « enum » en string
  - `object Parse(Type type ,string value, bool ignoreCase)`
    - Convertit une « string » en une valeur « enum »
  - `string[] GetNames(Type type)`
    - Renvoi la liste de tout les noms de l'énumération
  - `object[] GetValues(Type type)`
    - Renvoi la liste des valeus de l'énumération

# Structures

- **Mot clé `struct`**
- **Les différences avec une classe :**
  - Est de type valeur
  - Ne peut être dérivée et ne dérive pas (sauf interface)
  - Pas de constructeurs par défaut
    - Possibilité de redéfinir un constructeur autre
  - Création d’un objet sans le mot clés « new »
    - Les champs sont initialisés à 0

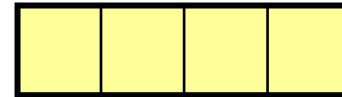
```
struct Foo {
        int field;
        void method()
        ….
}
```

# Agenda

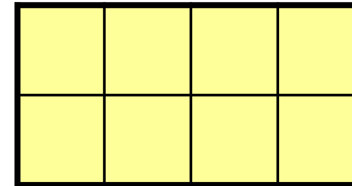
- **Encapsulation et visibilité**
- **Classe :**
  - Définition et membres
  - Statique
  - Constructeur
- **Héritage**
  - Polymorphisme
  - Classe abstraites
  - Interfaces
- **Enum et Structures**
- **Tableaux et collections**

# Les tableaux : définition

- **Tableaux à une dimension**
  - Ex : int [] a = new int[10];



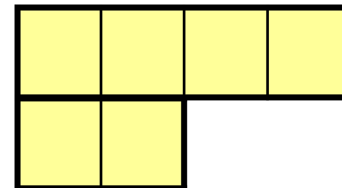
- **Tableaux à plusieurs dimensions**
  - Ex : int [,] a = new int [10,2];



- **Tableaux irréguliers**
  - Ex : int [][] a = new int [2][ ]

    a[0] = new int[10];

    a[1] = new int[5];



**Héritent de « `System.Array` »**

**Initialisé par zéro ou null**

**Initialisé par l'utilisateur**

```
int[,] a1 = { { 1, 2 }, { 2, 1 } };
```

# Les tableaux : covariances

- **Vérification de la validité d'un « cast » pour chaque élément du tableau**

```
interface I{} ; class A : I {} ; class B : I {};
A[] a = { new A(), null };
```

  - Conversion explicite sans problème

```
I[] i = a;
```

  - Conversion explicite : Vérification des « cast »

```
B[] i = a;
```

    – Compile mais InvalidCastException sera lancé à l'exécution

# Les tableaux : BitArray

- **Permet de stocker un tableau unidimensionnel de booléens**
  - Optimise le stockage
  - Constructeurs d'initialisation
  - Méthodes de logique sur des tableaux :
    - `BitArray Not()`
    - `SetAll(bool)`
    - `BitArray And(BitArray)` …

```
BitArray ba1 = new BitArray(10,true);
BitArray ba2 = new BitArray(10,false);
BitArray ba3 = ba1.And(ba2);
```

# IEnumerable / IEnumerator

- Un objet est énumérable si il implémente l'interface `IEnumerable`

- Une classe qui est énumérable peut être énumérée par `foreach`

```
public interface IEnumerable {
   IEnumerator GetEnumerator();
}
public interface IEnumerator
{
   object Current { get; }
   bool MoveNext();
   void Reset();
}
```

```
int[,] a1 = { { 1, 2 }, { 3, 4 } };
foreach (int i in a1)
{
   System.Console.WriteLine(i);
}
```

```
1
2
3
4
```

# Collection / bibliothèque de classes

- **Interface de base : ICollection<T> : IEnumerable<T>**
  - Add, Clear, Remove
- **Interface de la liste : IList<T> : ICollection<T>**
  - IndexOF, Insert, RemoveAt , this[]
- **Interface du dictionnaire : IDictionary<K,V> : ICollection<KeyValuePair<K,V>>**
- **Implémentation :**
  - List<T>, Dictionary<K,V>