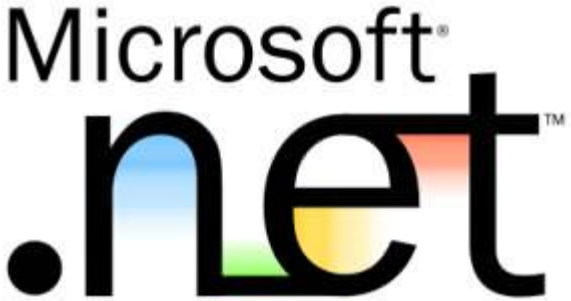

C# 2.0

Introduction à C# et au framework .Net

ISIMA ZZ2F2

Luc TOURAILLE

Source : "*C# 3.0 in a Nutshell*", Joseph & Ben ALBAHARI

Plan

I – Les bases du langage

II – Création de types

III – Fonctionnalités avancées

IV – Framework .Net

Classes

- Encapsulation de données et de méthodes agissant sur ces données.
- Doivent être vues comme des boîtes noires (l'utilisateur s'intéresse aux fonctionnalités de la classe, pas à son implémentation)

```
[modificateurs] class NomClasse[<T>]
[:BaseClass ou Interface][,Interface…]
{
    [membres]
}
```

Champs

- Champ = variable membre

- Peuvent être :
  - `static` : champ partagé par toutes les instances de la classe
  - `readonly` : non modifiable après son assignation ; doit être assigné dans sa déclaration ou dans le constructeur
  - `volatile` (threading), `new` (héritage), `unsafe` (code non managé (pointeurs…))

Champs

- `public`, `internal`, `private`, `protected` :

Modificateur	Visibilité
`public`	Visible partout
`internal`	Accessible uniquement dans l'assembly et les assemblies "amis"
`protected`	Visible à l'intérieur du type et dans les sous-classes
`private`	Accessible uniquement à l'intérieur du type
`protected internal`	Union de protected et internal

Méthodes

- Suite d'instructions
- Peut accepter des paramètres et renvoyer des valeurs (type de retour ou paramètres `ref/out`)
- Modificateurs :
  - `static`
  - D'accès (`public internal private protected`)
  - Héritage (`new virtual abstract override sealed`)
  - Code non managé (`unsafe extern`)

Méthodes

- La signature d'une méthode (nom et types des paramètres) doit être unique dans le type.

- Possibilité de *surcharger* les méthodes (même nom, signature différente)
  - ```
    void Foo(int x);
    void Foo(double x);   // OK
    char Foo(int x);      // Erreur de compil
    void Foo(ref int x); // OK
    void Foo(out int x)   // Erreur de compil
    ```
    ```

Constructeurs

- Initialisent l'instance
- Surcharge possible, éventuellement en appelant un autre constructeur

```
class Vin
{
    public decimal Prix;
    public int     Annee;
    public Vin(decimal prix) { Prix = prix; }
    public Vin(decimal prix, int annee) : this(prix)
    { Annee = annee; }
}
```

Constructeurs

- Dans une classe, si aucun constructeur n'est déclaré, le compilateur crée un constructeur par défaut
- Dans une structure, on ne peut pas définir son propre constructeur par défaut.
- Ordre de construction déterministe :
  - D'abord les initialisations, dans l'ordre des déclarations
  - Ensuite le corps du constructeur

Finaliseurs

- Appelé avant de collecter la mémoire ⇨ aucune garantie du moment où il va être invoqué
- Si on a des ressources à libérer, on préférera implémenter *IDisposable*, et définir la méthode *dispose()*

Propriétés

- De l'extérieur, on a l'impression de manipuler un champ, mais on effectue en fait des appels de méthode (accesseurs).

Propriétés

```
class Population
{
    int nbHommes;
    int nbFemmes;
    public int NbHommes { get { return nbHommes; }
                          set { nbHommes = value;}}
    public int NbFemmes { get { return nbFemmes; }
                          set { nbFemmes = value;}}
    public int Total    { get { return NbHommes + NbFemmes;}}
}
…
Population p = new Population();
p.NbHommes   = 12;       // appelle "set"
int total    = p.Total; // appelle "get"
```

Indexeurs

- Permet d'utiliser l'opérateur [] sur un objet

```
class Bibliotheque
{
    Livre[] livres;
    public Bibliotheque(int nbLivres)
      { livres = new Livre[nbLivres]; }
    public Livre this[int index]
      { get { return livres[index]; }
        set { livres[index] = value;}}
}
…
Bibliotheque bib = new Bibliotheque(10);
bib[0] = new Livre("D. Adams", "Starship Titanic");
```

Indexeurs

```
class Bibliotheque
{
   public IEnumerable<string> this[string nom]
   { get
     {
       List<string> titres = new List<string>();
       foreach (Livre l in livres)
         if (l.Nom == nom) titres.Add(l.Titre);
       return titres;
     }
   }
}
…
foreach (string titre in bib["D. Adams"]) Console.Write(titre);
```

Constantes

- Champ dont la valeur ne change jamais
  - `public class Math`
    `{ public const double Pi = 3.14159265; }`
- Différent d'un champ `static readonly`
  - Types prédéfinis et enum uniquement
  - Doit être initialisé à la déclaration
  - Valeur évaluée à la compilation et substituée à toute les occurrences de la constante

Constructeur statique

- Exécuté une fois par type, avant toute instanciation et tout accès à un membre statique
  - ```
    public class Math
    { static Math() {//init type}}
    ```
- Invoqué de manière non déterministe
- On peut avoir des classes statiques (qui ne peuvent contenir que des membres statiques)
    ```

Classes partielles

- Diviser la définition d'une classe (typiquement entre plusieurs fichiers)
- Scénario classique : une partie de la classe est généré automatiquement, l'autre est implémentée par le développeur

```
// TotoGen.cs - généré
partial class Toto { … }

// Toto.cs - manuel
partial class Toto { … }
```

Héritage

- Relation *est-un*
  - `public class Lion : Animal { … }`
- Classe dérivée possède tout les membres de la classe de base
- Pas d'héritage multiple (mais possibilité d'implémenter plusieurs *interfaces*)

Polymorphisme

- Références polymorphiques : une référence vers une classe de base peut pointer vers une instance d'une sous-classe.
- Utilisation des méthodes de la classe de base sans se soucier du type réel de l'objet

Transtypage

- Affecte uniquement les références
- Upcast implicite
  - ```
    Lion   l = new Lion(…);
    Animal a = l; // upcast
    ```
- Downcast explicite
  - ```
    Animal  a = new Gazelle(…);
    Lion    l = (Lion) a; // InvalidCastException
    Gazelle g = (Gazelle) a; // ok
    ```

Transtypage

- Opérateur `as`
  - `Lion l = a as Lion; // ok, l = null`

- Opérateur `is`
  - ```
    // teste validité du downcast
    if (a is Lion)
        Lion l = (Lion) a;
    ```
    ```

Méthodes virtuelles

- Permet de *redéfinir* des comportements dans les classes dérivées
- `virtual + override`

```
public class Animal
{
    public virtual void Manger() {…}
}
public class Lion : Animal
{
    public override void Manger() {… base.Manger(); }
}
```

Masquage

- Sans `override`, on *masque* la méthode ⇨ warning du compilateur
- Peut s'appliquer à tout les membres
- Si c'est volontaire, utiliser le mot-clé `new`

```
public class Animal
{ public int poids; }

public class Lion : Animal
{ public new int poids; }
```

Classes et membres abstraits

- Classe abstraite ⇨ ne peut être instanciée
- Peut contenir des membres abstraits (équivalent aux membres virtuels, mais sans implémentation)
- Doivent être implémentés par les sous-classes, sinon elles doivent être abstraites elles aussi

Classes et membres abstraits

```
public abstract class Animal
{ public abstract int Poids { get; }}

public class Lion : Animal
{
  public override int Poids
  { get { return …;}}
}

public abstract class Primate : Animal {}
// N'implémente pas Poids --> doit être abstract
```

Méthodes (et classes) scellées

- On peut empêcher la redéfinition de méthodes virtuelles dans les sous-classes

```
public class Animal
{ public virtual void Manger() {…}}
public class Lion : Animal
{ public sealed override void Manger() {…}}
```

- En scellant une classe, on empêche qu'elle soit dérivée

Constructeurs et héritage

- Constructeur par défaut des classes de base implicitement appelé (assure que l'objet est bien construit)
- `base` permet d'utiliser un autre constructeur

```
public class Animal
{ public Animal(string nom) {…}}

public class Lion : Animal
{ public Lion(string nom) : base(nom) {…}}
```

Ordre d'exécution

```
public class Base
{
    int x = 0;          // 3
    public Base(…)      // 4
    {…}                 // 5
}
public class Derivee : Base
{
    int y = 0;          // 1
    public Derivee(…)   // 2
    {…}                 // 6
}
```

Cas des structures

- Pas d'héritage (mais possibilité d'implémenter une interface)
- Pas de constructeur par défaut (le compilateur en génère un)
- Pas de finaliseur
- Pas de membres virtuels
- Pas d'initialisation des champs
- Tous les champs doivent être assignés à la construction

Interfaces

- Définit une spécification
- Similaire à une classe abstraite contenant uniquement des méthodes abstraites

```
public interface ICarnivore
{
  void Devorer(Animal a);
}
public class Lion : Animal, ICarnivore
{
  public void Devorer(Animal a); // forcément public
}
```

Interfaces

- Accessibilité uniforme des membres
- Une interface peut *étendre* une autre interface
  - ```
    public interface IUndoable { void Undo(); }
    public interface IRedoable : IUndoable
    { void Redo(); }
    ```
- L'implémentation d'un membre d'une interface peut être virtuelle
    ```

Type object

- Classe de base de *tout* les autres types
- Permet d'avoir des classes/méthodes très génériques (qui acceptent des instances de n'importe quel type)
- Même les types valeur peuvent être upcastés en `object` ⇨ unification de types, grâce au *boxing/unboxing*
  - ```
    int    x = 12;
    object o = x;
    int    y = (int) o;
    ```
    ```

Méthodes d'object

```
public class Object
{
  Object();

  Type GetType();
  virtual bool Equals(object o);
  static bool Equals(object o1, object o2);
  static bool ReferenceEquals(object o1, object o2);
  virtual int GetHashCode();
  virtual string ToString();

  protected virtual void Finalize();
  protected object MemberWiseClone();
}
```

Enums

- Enum = type valeur prenant uniquement certaines valeurs numériques nommées
  - `enum Couleur { Carreau, Pique, Trefle, Coeur }`
    `Couleur c = Couleur.Pique;`
- Possibilité de conversion entre la représentation entière et une instance de l'enum
  - `Couleur c = (Couleur) 1; // Pique`
- Enum peut représenter des flags (plusieurs valeurs combinées)