



# Programmation orientée objet en C#

*Point sur le laboratoire 1*



# Point sur le labo 1

- Namespaces et using
- Types valeurs et types références
- Le type string

# NAMESPACES ET USING

Ce qui correspond aux packages de Java

# Namespaces et using

- Les classes sont regroupées en **namespaces** (ou espaces de noms).

```
namespace ExemplesOriginaux
{
    class Ville
    {
        ... définition de la classe ...
    }
}
```

La notion correspondante en Java est celle de **package**.



La commande Java **import** permet d'importer le contenu d'une classe dans un package ; par exemple : **import java.util.Date;**

- Pour permettre un accès aisé au contenu d'un package, on utilise la commande **using** :

Sans using : nom <u>qualifié</u>	Avec using
ExemplesOriginaux.Ville v = ...	using ExemplesOriginaux; Ville v = ...

# Namespaces et using

- On utilise souvent des namespaces imbriqués :

Définition	Utilisation
<pre>namespace Soute.Valise.Sac { ... classe Cadeau ... }</pre>	<pre>Soute.Valise.Sac.Cadeau c = ... ou using Soute.Valise.Sac; Cadeau c = ...</pre>

- Écriture équivalente à :

```
namespace Soute
{
    namespace Valise
    {
        namespace Sac
        { ... classe Cadeau ... }
    }
}
```

En Java, la structure des packages doit correspondre à celle des dossiers où les fichiers-classes se trouvent (pas de telle règle en C#).



C# permet de répartir la définition d'une classe (ou d'un namespace) sur plusieurs fichiers, voire de mélanger plusieurs classes/namespaces dans un même fichier (mais... Clean Code !) (plus de détails dans le labo 1)

# TYPES VALEURS ET TYPES RÉFÉRENCES

Scalaires ou objets accessibles via un pointeur ?

# Types valeurs et types références

- On manipule des valeurs et objets similaires à ce qu'on trouve en Java.
- Par exemple : attributs d'une classe

```
private int nbHabitants;  
private const int NB_PERSO = 5;  
private string[] noms = new string [NB_PERSO];
```

- Notes :
  - const : doit être initialisée, invariable, implicitement static (ne pas confondre avec readonly – voir labos suivants)
  - les attributs sont automatiquement initialisés au « zéro » de leur type

# Types valeurs et types références

- Il faut distinguer les types valeurs et les types références (qui peuvent être prédéfinis ou pas).

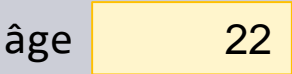
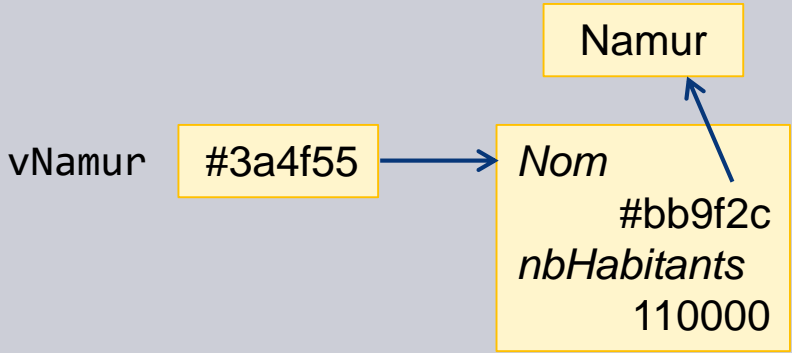
	Valeurs	Références
Prédéfinis	int, double, char, bool, ...	string, object, ...
Définissables	structures, énumérations (voir C)	classes

- Types valeurs et types références se comportent différemment...
  - en ce qui concerne leur **stockage en mémoire**,
  - pour l'**affectation**,
  - en ce qui concerne la **libération de la mémoire**  
(quand la variable devient inutile).



# Types valeurs et types références

- Les valeurs sont stockées et gérées différemment en mémoire.

Types valeurs	Types références
<p>On stocke la valeur de la variable.</p> <pre>int âge = 22;</pre> 	<p>On stocke l'adresse où le contenu de la variable est stocké.</p> <pre>Ville vNamur = new Ville (...);</pre> 
<p>Aussi :</p> <pre>bool valide = false;</pre> <p>(tous les types prédéfinis simples sauf les <b>strings</b>)</p>	<p>Aussi :</p> <pre>string city = "Namur"; int [] nombres = new int[10];</pre> <p>(string, tableaux, classes, délégués...)</p>

# Types valeurs et types références

- L'affectation (ou le passage d'arguments) fonctionne différemment.

Types valeurs	Types références
<p>On copie la valeur de la variable.</p> <p>âge 22</p> <p>ailleurs en mémoire</p> <p>âge2 = âge;</p> <p>âge2 22</p> <p>âge2 = 19;</p> <p><i>ne modifie que la copie</i></p>	<p>On copie l'adresse (création d'alias).</p> <p>vNamur #3a4f55 → Nom → Namur nbHabitants 110000</p> <p>vNamur2 = vNamur;</p> <p>vNamur2 #3a4f55</p> <p>vNamur2.nbHabitants++;</p> <p><i>modifie la valeur pour les deux !</i></p>
Passage d'argument : la méthode travaille sur une copie	Passage d'argument : la méthode travaille sur le contenu de l'objet !

# Types valeurs et types références

- La libération de la mémoire est gérée différemment.

## Types valeurs

La mémoire est libérée (et la variable disparaît) dès la fin de son scope.

âge 22

## Types références

C'est le Garbage Collector qui gère la libération de la mémoire.

vNamur #3a4f55 → *Nom*  
→ Namur  
*nbHabitants*  
110000

- en fonction des besoins de mémoire
- calcul du nombre d'alias (pas de suppression tant que le contenu est accessible)

# LE TYPE STRING

... et ses bizarreries

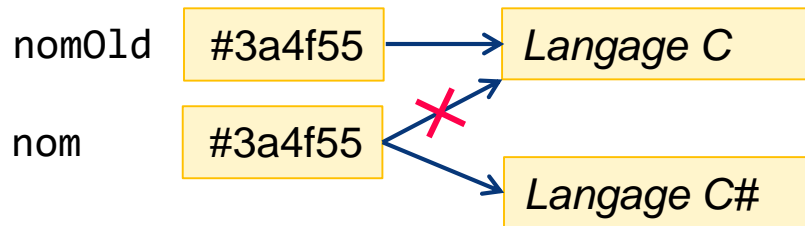
# Le type string

- C'est un **type référence**  
MAIS == est redéfini pour **comparer le contenu** !
  - Donc `chaîne1 == chaîne2` compare le contenu, pas les adresses !
  - Java : `chaîne1.equals(chaîne2)`
  - par contre, `<` et `>` ne sont pas redéfinis (il faut utiliser la méthode `compareTo`)
- « **String interpolation** » : les gabarits de chaînes en C#  
`string output = $"{name} a eu {âge} ans !";`

# Le type string

- Le type string est **immutable**.

= tout changement de la valeur entraîne la création d'une nouvelle zone de stockage en mémoire



```
string nomOld = "Langage C";
string nom = nomOld;
```

```
nom += "#";
```

- Code problématique :

```
public string ListingMaps () {
    string output = "";
    for (int iMap = 0 ; iMap < NB_MAPS_MAX ; iMap++)
        if (maps[iMap] != null)
            output += iMap + "-" + maps[iMaps] + "\n";
    return output;
}
```

# Le type StringBuilder

- Une solution : utiliser StringBuilder !

```
public string ListingMaps () {  
    StringBuilder output = new StringBuilder ();  
    for (int iMap = 0 ; iMap < NB_MAPS_MAX ; iMap++)  
        if (maps[iMap] != null)  
        {  
            output.Append(iMap);  
            output.Append("-");  
            output.Append(maps[iMaps]);  
            output.Append(Environment.NewLine);  
        }  
    return output.ToString();  
}
```