

# C# Avancé

**POO - Génériques - Collections - Exceptions - Délégués - Linq**

# Sommaire

- Programmation Orientée Objet
- Définition de Classes
- Polymorphisme
- Héritage
- Interfaces
- Génériques
- Collections
- List
- Set
- Dictionary
- Exceptions
- Les Lambdas et Délégués
- LINQ



# Programmation Orientée Objet

# Qu'est-ce que la Programmation Orientée Objet ?

- La **POO** est un paradigme de programmation informatique. Elle consiste en la **définition** et l'**interaction** de briques logicielles appelées **objets**. Un **objet** représente un **concept**, une **idée** ou toute **entité** du monde physique (personne, voiture, dinosaure).
- Lorsque que l'on programme avec cette méthode, la première question que l'on se pose est :  
**« qu'est-ce que je manipule ? »**
- Alors qu'en programmation **Procédurale**, c'est plutôt :  
**« qu'est-ce que je fait ? »**

# Qu'est-ce que la Programmation Orientée Objet ?

- Elle permet de **découper** une grosse **application**, généralement floue, en une multitude d'**objets** interagissant entre eux
- La POO améliore également la **maintenabilité**. Elle facilite les **mises à jour** et l'ajout de **nouvelles fonctionnalités**.
- Elle permet de faire de la **factorisation** et évite ainsi un bon nombre de lignes de code
- La réutilisation du code fut un argument déterminant pour venter les avantages des langages orientés objets.

# Les paradigmes de la POO

La POO repose sur plusieurs concepts importants

- **Accessibilité (ou Visibilité)**
- **Encapsulation**
- **Polymorphisme**
- **Héritage**
- **Abstraction**
- **Interfaces**
- **Fonctions Anonymes**
- **Généricité**

*Nous les aborderons tous par la suite.*

# Qu'est-ce qu'un objet en programmation?

Commençons par définir les objets dans le mode réel:

- Ils possèdent des **propriétés propres** : Une chaise a 4 pieds, une couleur, un matériau précis...
- Certains objets peuvent **faire des actions** : la voiture peut rouler, klaxonner...
- Ils peuvent également **interagir entre eux** : l'objet roue tourne et fait avancer la voiture, l'objet cric monte et permet de soulever la voiture...

Le concept d'objet en programmation s'appuie sur ce fonctionnement.

# Qu'est-ce qu'un objet en programmation?

Il faut distinguer ce qu'est l'objet et ce qu'est la définition d'un objet

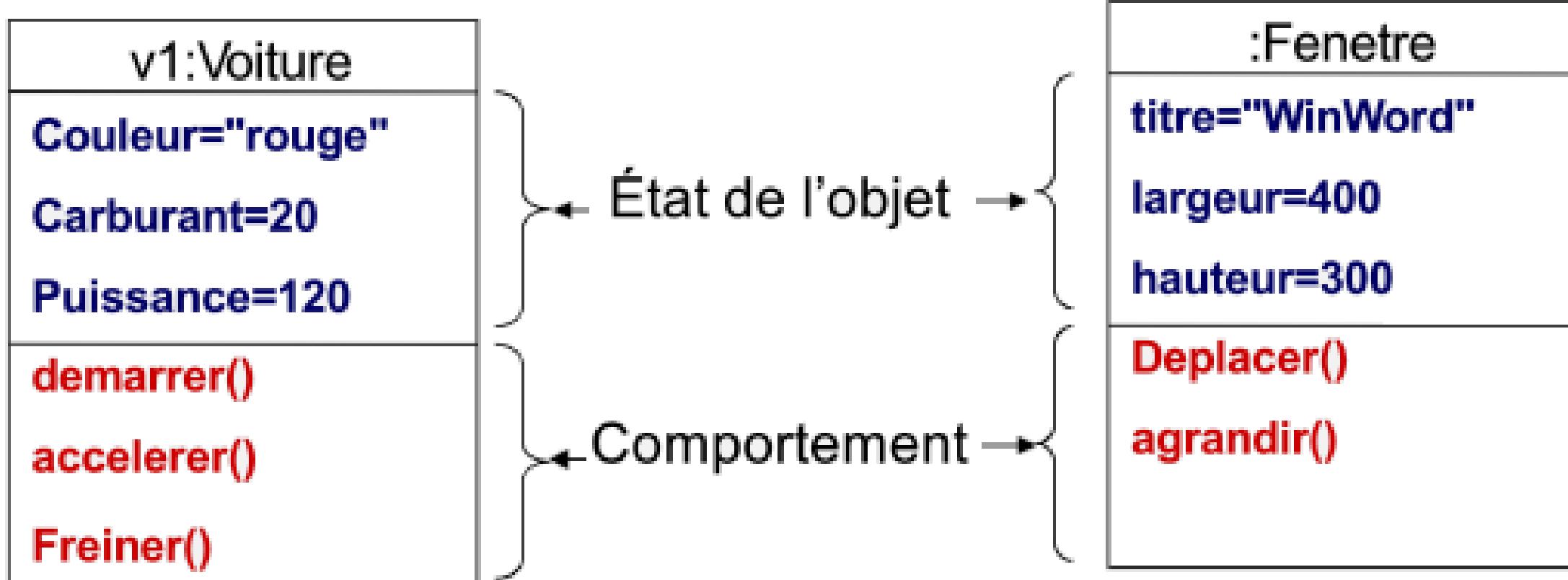
- **Le concept de l'objet** (ou définition/structure)
  - Permet d'indiquer ce qui compose un objet, c'est-à dire quelles sont ses propriétés, ses actions...
- **L'instance d'un objet**
  - C'est la création réelle de l'objet : *Objet Chaise*
  - En fonction de sa définition : *4 pieds, bleu...*
  - Il peut y avoir **plusieurs instances** : *Plusieurs chaises, de couleurs différentes, matériaux différents...*

# Qu'est-ce qu'un objet en programmation?

- Un objet est une structure informatique définie par un **état** et un **comportement**.
  - L'**état** regroupe les **valeurs instantanées** de tous les **attributs de l'objet**. Il peut changer dans le temps.
  - Le **comportement** décrit les **actions** et les réactions de l'objet. Autrement dit le comportement est défini par **les opérations que l'objet peut effectuer**. Généralement, c'est le comportement qui modifie l'état de l'objet.

# Exemple

*Objet = état + comportement*



*Il s'agit ici d'un diagramme d'objet*

## Identité d'un objet

- En plus de son état, un objet possède une **identité** qui caractérise son existence propre.
- Cette identité s'appelle également **référence** de l'objet
- En terme informatique de bas niveau, l'identité d'un objet représente son **adresse mémoire**.
- Deux objets **ne peuvent pas avoir la même identité**: c'est-à-dire que deux objet ne peuvent pas avoir le même emplacement mémoire

## Résumé

- La **POO** est un **paradigme de programmation** basé sur la manipulation d'**objets**, représentant des entités ou concepts du monde réel.
- Elle **découpe les applications** complexes en **objets**, améliorant ainsi la maintenabilité et favorisant la réutilisation du code.
- **Concepts clés** : Accessibilité, Encapsulation, Héritage, Polymorphisme, Abstraction, Interfaces, Fonctions Anonymes, Généricité.
- Un objet combine **état** (propriétés actuelles) et **comportement** (actions possibles), avec une **identité unique** (adresse mémoire).



# Définition de Classes

## Qu'est-ce qu'une Classe ?

Un **Classe** (`class`) permet de regrouper tous les éléments qui représenteront un Objet : ses **attributs**, ses **propriétés**, ses **méthodes**

On dit qu'une classe représente le concept de l'objet.

Dans les langages fortement typés, la **création d'une classe** aboutira à la création d'un **nouveau Type**

# Instanciation

- Les objets qui sont **définis à partir** d'une classe **appartiennent à celle-ci**.
- Ce processus s'appelle **l'Instanciation**
- On passe du **concept** (classe) à l'objet **réel** (instance/objet)
- La **classe est unique** mais les **objets** qui en **dérivent** peuvent être nombreux

## Program

Nous avons déjà pu voir une Classe dans le code que nous avons utilisé précédemment qui a été généré par Visual Studio, la classe **Program**.

A partir du **.NET 6** cette classe apparaît par défaut de manière **tronquée** et nous **ne voyons pas la totalité de sa structure syntaxique**.

## Syntaxe de Program

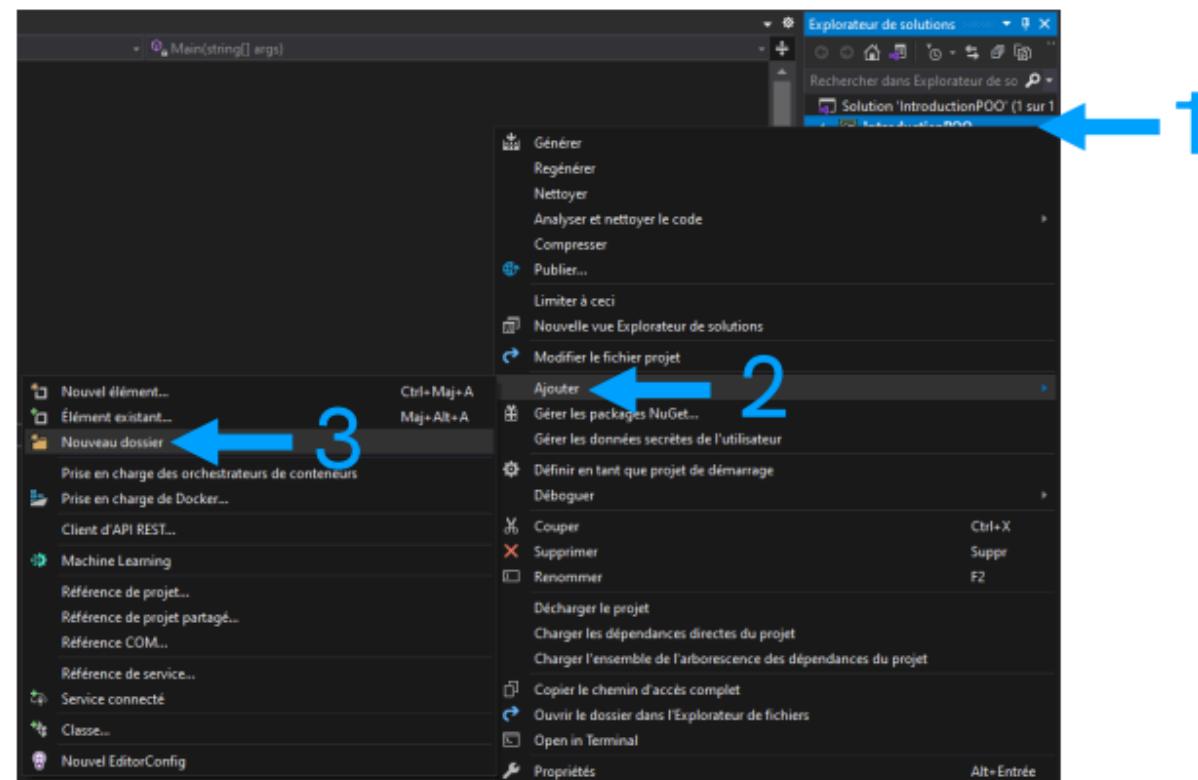
La class **Program** est une classe particulière car elle contient la méthode « **Main()** » qui est le **point d'entrée de notre application**

- Elle fonctionne comme toutes les classes
- La classe Program peut **faire des actions**, par exemple la **méthode Main()** en est une
- Notez la présence des **accolades {}** qui **délimitent la classe** ( le bloc d'instructions de celle-ci )
- Les **noms des classes** comme des méthodes s'écrivent en **PascalCase**.

Exemple: MaNouvelleClasse

# Création d'une nouvelle Classe

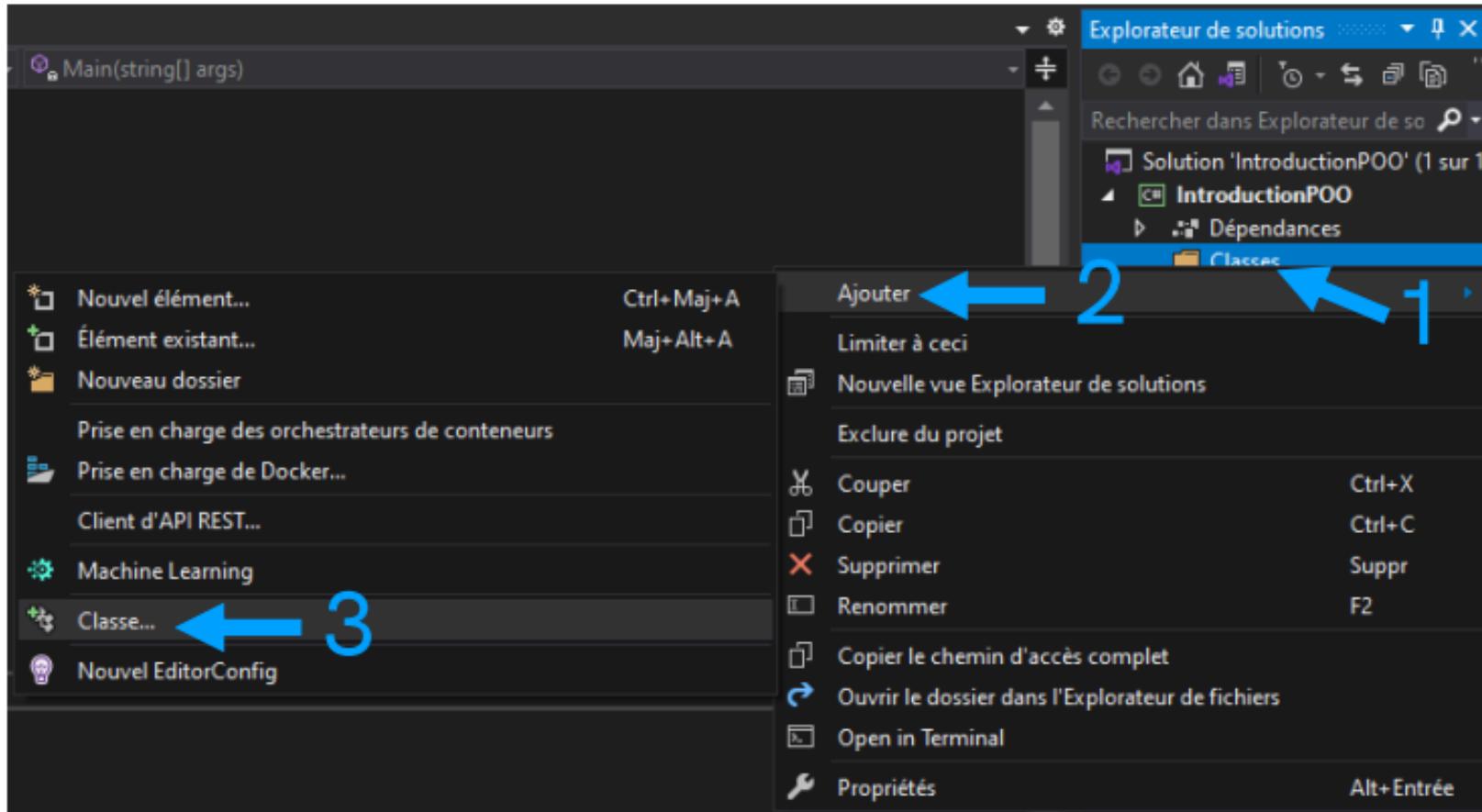
- Créer un nouveau projet console
- Par défaut, l'onglet Program.cs est ouvert
- Dans explorateur de solution créez un dossier nommé Classes en faisant un clic droit sur le nom de votre projet



*Il est important de structurer un projet en dossiers et sous-dossiers*

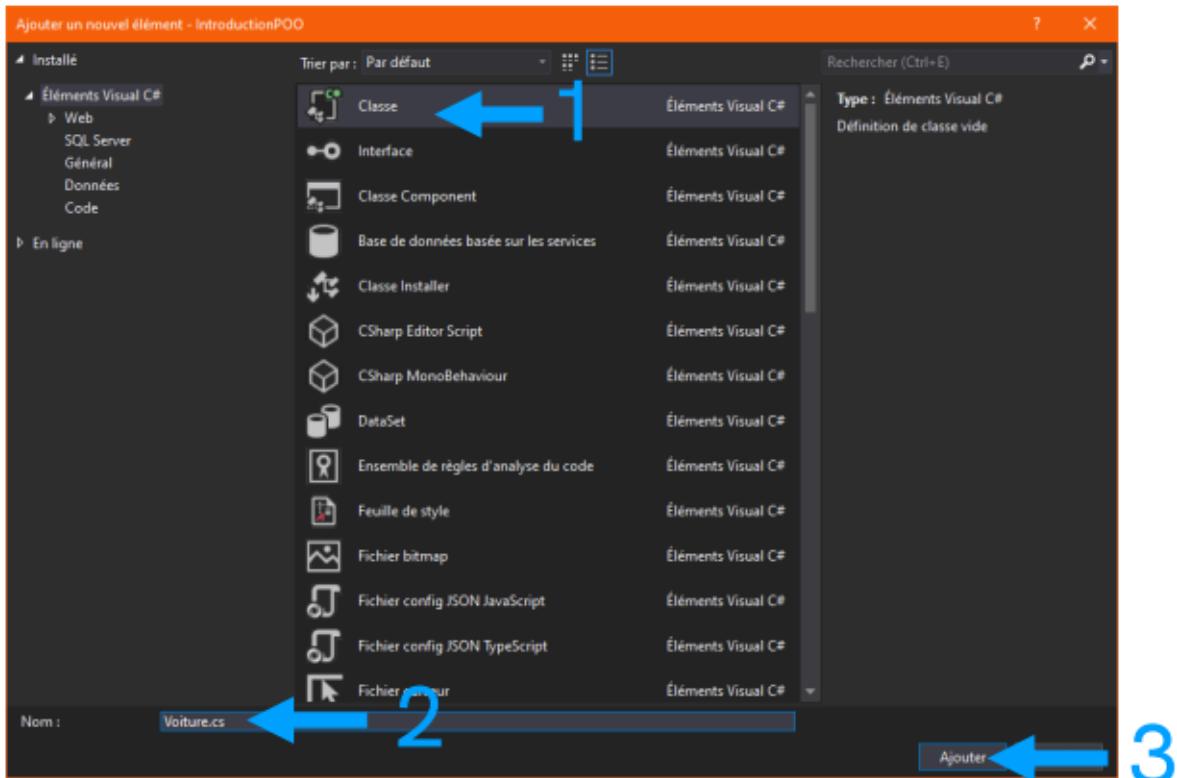
# Création d'une nouvelle Classe

Clic droit sur votre dossier «Classes» puis «Ajouter» puis «Classe»



# Création d'une nouvelle Classe

- Nommer cette nouvelle classe.
- Pour notre exemple nous l'appellerons « Voiture.cs »
- Dans cette fenêtre de Visual Studio on peut voir plusieurs **templates de fichier**, ils nous donnent **une base qu'il faudra retravailler**



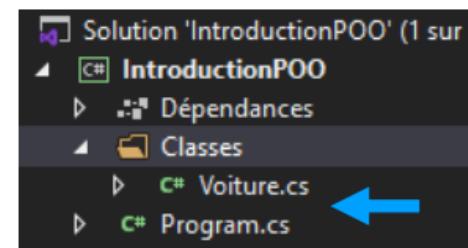
*Ces templates peuvent être d'autres types de fichiers (cshtml, razor, ...)*

# Création d'une nouvelle Classe

- Visual Studio Ouvre cette nouvelle classe, elle apparaît dans l'arborescence de votre application
- Maintenant, nous allons pouvoir commencer à développer notre première classe, la class **Voiture**. Elle définira **le concept de voiture** et on pourra **l'instancier** pour **créer plusieurs voitures distinctes**

```
using System;
using System.Collections.Generic;
using System.Text;

namespace IntroductionPOO.Classes
{
    0 références
    class Voiture
    {
    }
```



## Namespace

Notez la présence du mot clé `namespace`, il permet de **définir un ou plusieurs espace de nom**, ce qui correspondra au chemin d'espaces de nom qui permettra l'accès à la classe.

Il est possible de le définir avec une instruction unique pour le fichier `namespace MonNamespace ;` ou avec un bloc `namespace MonNamespace{ }`

**/!\ Attention, un espace de nom .NET n'est pas un dossier !**

Mais par convention cet **espace de nom** est censé **porter le même nom que le dossier** où l'on a mis le fichier avec le code de la classe.

Une erreur commune est d'oublier de changer le namespace lors du déplacement ou de la copie du fichier.

## La notion de visibilité/accessibilité

L'**indicateur de visibilité** est un mot clé qui sert à indiquer **depuis où** on peut **accéder** à l'**élément** qui le suit.

Visibilité	Description	Classe	Membres de classes
<b>public</b>	Accès non restreint	✓	✓
<b>private</b>	Accès uniquement depuis la même classe	✗	✓
<b>protected</b>	Accès depuis la même classe ou depuis une classe dérivée (cf héritage)	✗	✓
<b>internal</b>	Accès restreint à la même assembly (par défaut)	✓	✓

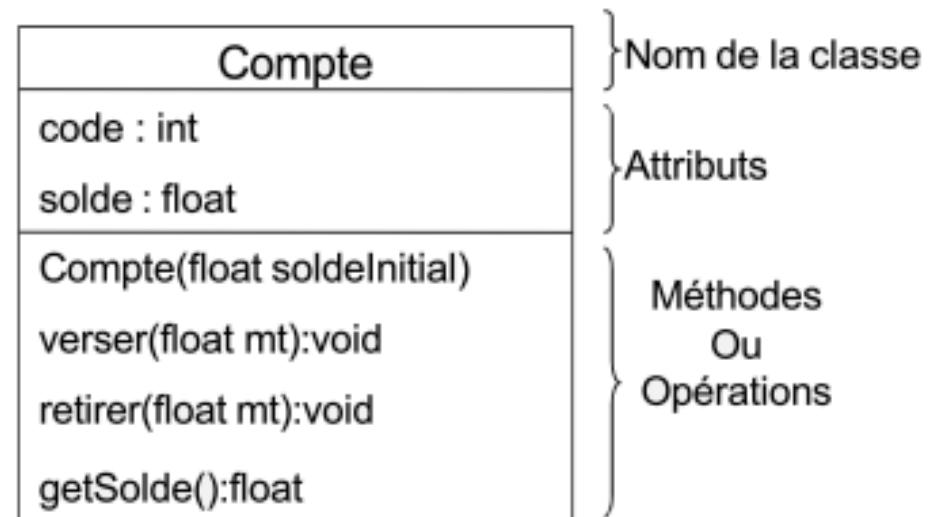
*Il existe aussi **protected internal** et **private protected** qui sont des cas spécifiques*

## Elements d'une classe

Élément	Caractéristiques	Détails
<b>Attributs :</b> Variables d'instance	- Nom - Type - Valeur initiale (optionnelle)	<b>État</b> de l'objet
<b>Méthodes :</b> Fonctions liées à l'instance	<b>Signature :</b> - Nom - Type de retour - Paramètres	<b>Comportement</b> de l'objet
<b>Constructeurs</b>	- Pas de type de retour - Même nom que la classe - Paramètres	Appelés à la <b>création</b> de l'objet
<b>Destructeur</b>	Rarement utilisé, varie selon les langages	Méthode particulière appelée par le <b>Garbage Collector</b> à la <b>suppression</b>

# Représentation UML d'une classe

- Une classe est représenté par un rectangle à 3 compartiments :
  - Un compartiment qui contient le nom de la classe
  - Un compartiment qui contient la déclaration des attributs
  - Un compartiment qui contient les méthodes



# Les Attributs

Les **attributs** sont un **ensemble de variables** permettant de définir les caractéristiques de notre objet (aussi appelés **variables d'instance**). Ils doivent être déclarés par convention **au début de notre classe**.

- **Tous les types de variables sont utilisables pour la déclaration des attributs y compris des objets** ( int, float, string, List<>, Voiture, Personne ... )
- Ils se déclarent comme suit et **peuvent être initialisés** ou non en fonction des besoins de votre application (norme "\_xx" => **private**)

```
private string _model;
```

```
private string _model = "Tesla";
```

## Les Propriétés

- Le principe de l'**encapsulation** de la POO a pour bonne pratique de laisser **les attributs en privé (private)**, c'est-à-dire **uniquement accessibles depuis l'intérieur de cette classe**
- Dans une majorité des langages, on pourra y accéder par des **méthodes publiques** nommées en général GetXXX() et SetXXX().
- En C#, l'**encapsulation** est simplifiée par le principe de **propriétés**, elles regroupent le **getter** et le **setter** en un seul élément/membre de la classe

# Les Propriétés

Voici la syntaxe pour une **propriété** liée à un **attribut** en C#

```
public string Model { get => _model; set => _model = value; }
```

Équivalent en syntaxe longue :

```
public string Model
{
    get
    {
        return _model;
    }
    set
    {
        _model = value;
    }
}
```

# Les Propriétés

Si l'on veut **définir** un **comportement spécifique** à la **modification (setter)** ou à la **récupération (getter)** d'un attribut, il faudra donc **changer le bloc d'instruction** du set ou du get en fonction de nos besoins.

```
public double Poids
{
    get
    {
        Console.WriteLine(
            "_poids a été récupéré, il vaut "
            + _poids);
        return _poids;
    }
    set
    {
        if (value <= 0)
        {
            Console.WriteLine(
                "La valeur passée au poids est invalide !!!"
                + "Je le met donc à 100 kg.");
            _poids = 100;
        }
        else
            _poids = value;
    }
}
```

# Les Propriétés

- Plus généralement une propriété est en fait **le regroupement de 2 méthodes** (getter et setter) qui ont une **signature bloquée**
- C'est une des **particularité du C#**, dans d'autres langages comme le **Java**, les **propriétés n'existent pas** et sont remplacées par 2 méthodes `getAttribut()` et `setAttribut(valeur)`. Exemple:
  - **Getter (get)**

```
public string GetModel() {return _model;}
```

- **Setter (set)**

```
public void SetModel(string value) {_model = value;}
```

# Les Propriétés en lecture seule

Si l'on veut **empêcher la modification d'un attribut**, on peut décider de **bloquer le setter** de la propriété de 2 manières :

- Propriété **avec setter en privé (lecture seule extérieure)**  
Il est toujours possible d'utiliser le **set** à l'intérieur de la classe

```
public string Model { get => _model; private set => _model = value; }
```

- Propriété **sans setter (lecture seule totale)**  
La propriété n'a plus de setter, il n'est plus possible de l'affecter via la propriété

```
public string Model { get => _model; }
```

## Les Propriétés composées (en lecture seule)

Lorsque l'on veut faire **une propriété qui dépend d'autres Propriétés et Attributs**, il est possible d'avoir une propriété **sans setter** avec le **getter** qui **retourne une valeur** le plus souvent **calculée** à partir de ces propriétés/attributs.

3 syntaxes marchent pour les propriétés en lecture seule :

```
public string NomComplet { get => Nom + " " + Prenom; }
```

```
public string NomComplet { get { return Nom + " " + Prenom; } }
```

```
public string NomComplet => Nom + " " + Prenom;
```

La dernière ne définit aussi qu'un Getter mais sa syntaxe est simplifiée au maximum.

# Les Propriétés Automatiques (auto-property)

- Il existe des propriétés **sans attribut visible** dont **le getter et setter n'ont pas d'instructions**, elle s'appelle des **auto-properties**
- Ces propriétés correspondent à **des propriétés basiques d'encapsulation pour un seul attribut** mais cet attribut est **caché**, **il n'est pas accessible**. On les utilise quand on n'a **pas de comportement particulier** à ajouter au **get** et au **set**

```
public string Model { get; set; } // pas d'attribut _model visible
public string Model { get; set; } = "Fiat multipla"; // avec initialisation

// property classique
private string _model;
public string Model { get => _model; set => _model = value; }
```

## Les attributs et propriétés d'une classe

Voici notre class Voiture après la déclaration de quelques attributs et de leurs propriétés

```
internal class Voiture
{
    private string _model;
    private string _couleur;
    private int _reservoir;
    private int _autonomie;

    public string Model { get => _model; set => _model = value; }
    public string Couleur { get => _couleur; set => _couleur = value; }
    public int Reservoir { get => _reservoir; set => _reservoir = value; }
    public int Autonomie { get => _autonomie; set => _autonomie = value; }
}
```

## Le constructeur

Maintenant que notre **concept de Voiture (class)** a des **attributs** et des **propriétés**, il nous faut un outil pour pouvoir créer **des nouvelles voitures spécifiques (instances/objets)**, on parle de **construction**.

- Cet outils s'appelle donc le **constructeur**, il définit la manière de **créer une nouvelle instance**
- Il est **similaire à une fonction** et **prendra des paramètres** en entrée
- Lors de son **appel** il faudra utiliser le mot-clé **new** (instanciation/construction d'un **nouvel** objet/instance)

## Le constructeur

Voici la syntaxe d'un constructeur en C# pour notre class Voiture  
(Notez sa visibilité en public)

```
public Voiture(string model, string couleur, int reservoir, int autonomie)
{
    _model = model; // avec l'attribut
    Model = model; // avec la propriété
    Couleur = couleur;
    Reservoir = reservoir;
    Autonomie = autonomie;
}
```

Il est souvent préférable d'**utiliser les propriétés** pour passer par les setters et ainsi réutiliser leurs instructions

## Mot-clé this

Lorsque l'on génère le constructeur avec les **actions rapides** de Visual Studio (alt+Entrée), par défaut il ajoute le **mot-clé this**.

```
public Voiture(string model, string couleur, int reservoir, int autonomie)
{
    this._model = model;
    this.Couleur = couleur;
    this.Reservoir = reservoir;
    this.Autonomie = autonomie;
}
```

Ce mot-clé représente **l'instance sur laquelle on travaille**, dans le constructeur il s'agit donc de **celle que l'on construit**. Il est le plus souvent **facultatif** en C# (si l'on respecte les conventions de nommage `_nom`)

## Constructeur par défaut (sans paramètres)

Lorsque l'on crée une **nouvelle classe vide**, on **pourrait penser** qu'il est **impossible de l'instancier si aucun constructeur n'est défini**

En réalité, **il existe un constructeur vide par défaut** (implicite/invisible) dans toute classe qui **n'a pas encore de constructeur**

Voilà à quoi il correspond :

```
public Voiture() { }
```

Dès le moment où l'on en ajoute un nous-même, ce constructeur **disparaît**

# Vue d'ensemble de notre class Voiture à présent

```
public class Voiture
{
    // Attributs
    private string _model;
    private string _couleur;
    private int _reservoir;
    private int _autonomie;

    // Propriétés
    public string Model { get => _model; set => _model = value; }
    public string Couleur { get => _couleur; set => _couleur = value; }
    public int Reservoir { get => _reservoir; set => _reservoir = value; }
    public int Autonomie { get => _autonomie; set => _autonomie = value; }

    // Constructeurs
    public Voiture() { }
    public Voiture(string model, string couleur, int reservoir, int autonomie)
    {
        Model = model;
        Couleur = couleur;
        Reservoir = reservoir;
        Autonomie = autonomie;
    }
}
```

## L'instanciation d'un objet

Maintenant que notre **class Voiture** a des **attributs**, des **propriétés** et des **constructeurs**, nous allons pouvoir créer des voitures depuis notre class Program

- Voici la syntaxe pour **l'instanciation d'un objet** en C# (utilisation du constructeur sans-paramètres)

```
// type nomVariable = new Classe();
Voiture autoDeGuillaume = new Voiture();
```

- Attention, la class Voiture n'est **pas reconnue** tant que nous n'avons pas fait l'**import de notre namespace**

```
using Namespace.SousNamespace.Classe;
```

# L'instanciation d'un objet avec paramètres

Instantiation avec l'autre constructeur que nous avons défini

```
// Voiture(string model, string couleur, int reservoir, int autonomie)  
Voiture autoDeGuillaume = new Voiture("Fiat multipla", "Rouge", 63, 733);
```

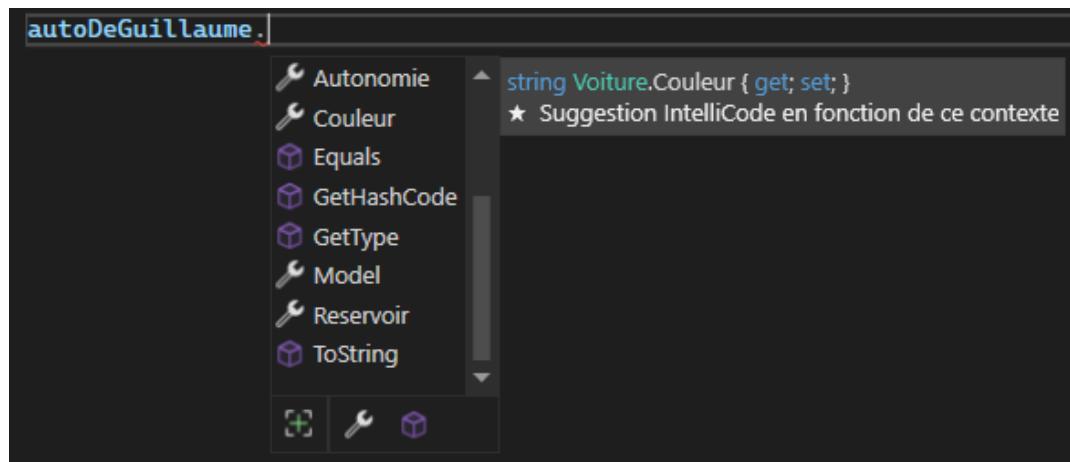
Ici, les attributs auront les valeurs définies à l'appel du constructeur.

Cependant si vous avez défini un comportement spécifique dans le constructeur ou les propriétés, ces valeurs peuvent changer.

# La modification d'un objet instancié

Maintenant que nous avons instancié notre objet Voiture pour pouvons **accéder à ses propriétés** via l'**auto complétion** de l'IDE (Ctrl+Espace ou Alt+Enter le plus souvent)

Il suffira ensuite de les **assigner** pour les modifier pour notre instance depuis la variable autoDeGuillaume



```
autoDeGuillaume.Model = "Clio" ;
autoDeGuillaume.Couleur = "Noir" ;
autoDeGuillaume.Reservoir = 45;
autoDeGuillaume.Autonomie = 900;
```

# Affichage de notre objet Voiture dans la console

Maintenant que nous avons instancié notre objet Voiture, nous pouvons l'utiliser. Essayons de l'**afficher dans la console** :

```
Console.WriteLine(autoDeGuillaume);
// résultat : Namespace.Voiture
```

Ce résultat est la **représentation textuelle de l'objet**. Nous verrons comment le changer par la suite (cf .ToString()).  
Voilà comment nous aurions pu l'afficher :

```
Console.WriteLine($"Notre première voiture est une {autoDeGuillaume.Model} de couleur {autoDeGuillaume.Couleur}");
Console.WriteLine($"Elle à un réservoir de {autoDeGuillaume.Reservoir} litres pour une autonomie de {autoDeGuillaume.Autonomie} km.");
```

Pour les attributs non définis, ils auront leur valeur par défaut `default`.

## Les méthodes d'une classe

Une **méthode** est **une fonction liée à une classe**, elle est définie dans le bloc de la classe, depuis celle-ci on peut **accéder** aux **attributs**, **propriétés** et autres **méthodes** de la classe.

Pour faciliter l'affichage de nos objets **Voiture** , nous pouvons mettre le bout de code précédent dans une **méthode** pour en faciliter le réemploi

```
public void Afficher()
{
    Console.WriteLine($"Notre première voiture est une {Model} de couleur {_couleur}");
    Console.WriteLine($"Elle à un réservoir de {this.Reservoir} litres pour une autonomie de {this._autonomie} km.");
}
```

## Les méthodes d'une classe

Il est possible d'ajouter autant les méthodes que nous souhaitons, leur **nom** donnera **une idée de leur utilité pour la classe**.

Faisons ensemble une méthode `Demarrer()`.

- Nous ajoutons une Propriété booléenne `Demaree` et pour indiquer si le moteur tourne. Nous pourrons utiliser celle-ci afin de vérifier si le moteur tourne avant de le démarrer.
- **Si** elle est **éteinte** nous afficherons un message dans la console pour informer l'utilisateur que **la voiture démarre**
- **Sinon** nous indiquerons que **le moteur tourne déjà**

# Les méthodes d'une classe

Voici la Méthode Demarrer().

```
public bool Demarrer()
{
    if ( !Demarree )
    {
        Demarree = true;
        Console.WriteLine( "La voiture est démarée... le moteur tourne !" );
    }
    else
        Console.WriteLine("La voiture est déjà démarée !");

    return Demarree;
}
```

# Commentaires de documentation dans une classe

Le **commentaire de documentation** se fait avant **un membre d'une classe, une classe** ou beaucoup d'autre éléments du C#.

Il permet d'**expliquer l'élément en question** et cette explication sera affichée par visual studio au survol de l'élément.

```
/// <summary>
/// Fait l'addition de 2 entiers
/// </summary>
/// <param name="a">Premier entier</param>
/// <param name="b">Deuxième entier</param>
/// <returns>Addition des entiers</returns>
public int Add(int a, int b)
```

## Notion de static

Il est possible via l'utilisation du mot clé **static** de **créer des membres** (attributs, propriétés et méthodes) qui seront **liés à la classe** et non aux instances.

```
private static int _nombreDeVoitures = 0;  
public static int NombreDeVoitures{ get => _nombreDeVoitures; }  
public static int NombreDeVoitures { get; } = 0; // en auto-property
```

Ici nous avons un **attribut de classe** et non d'instance, il est **partagé entre toutes les instances** et accessible directement depuis la classe avec cette syntaxe :

```
Console.WriteLine("Total :" + Voiture.NombreDeVoitures);
```

# Notion de static

Un autre exemple avec des **méthodes static** (méthode de classe):

```
public static void AfficherTotalVoitures()
{
    Console.WriteLine("Voitures créées avec le constructeur : " + NombreDeVoitures);
}
public static void AfficherVoituresParlantes()
{
    Console.WriteLine("Les voitures qui parlent ça n'existe pas...");
```

Elles serviront en général à travailler avec des notions relatives à toutes nos voitures en non une en particulier

# Notion de static

## Utilisations des statics dans un constructeur

```
public Voiture()
{
    _nombreDeVoitures++;
    AfficherTotalVoitures();
}
```

!\ Attention, pour des raisons évidents, un constructeur **ne peut pas être static**, il permet de créer **une** instance

# Constructeur dépendant d'un autre constructeur

À l'aide du `: this()` on vient préciser que à l'appel d'un constructeur, on en appelle aussi un autre, cela permet d'éviter les répétitions

```
public Voiture()
{
    _nombreDeVoitures++;
    AfficherTotalVoitures();
}
public Voiture(string model, string couleur, int reservoir, int autonomie) : this()
{
    // réutilise le premier constructeur
    Model = model;
    Couleur = couleur;
    Reservoir = reservoir;
    Autonomie = autonomie;
}
public Voiture(int reservoir, int autonomie) : this("fiat multipla", "rouge", reservoir, autonomie)
{
    // réutilise le deuxième constructeur avec des valeurs prédéfinies
} // attention aux conflits (2 constructeurs avec le même nombre de paramètres)
```



# Le Polymorphisme

# Rappel sur les signatures

```
public bool AjouterVoiture(Voiture voiture)
```

- La **signature** de la fonction/méthode nous renseigne sur le **nom**, les **paramètres** et **type de retour**
- Lorsque l'on parle de méthodes, le mot **clé de visibilité / accessibilité** vient s'ajouter
- **2 méthodes** portant le même nom mais avec **des paramètres** et **un type de retour différents** donnent bien **2 éléments distincts**, c'est **un premier cas de polymorphisme** (polymorphisme paramétrique)

# Le concept de Polymorphisme

- Le mot **polymorphisme** suggère qu'un élément **définit par son nom (identificateur/symbol)** possède **plusieurs formes**
- Il aura ainsi la capacité de faire **une même action** avec **différents types d'intervenants**
- En POO, ce concept s'applique principalement aux **méthodes**, mais aussi aux **propriétés** et aux **constructeurs**

# Les types de Polymorphisme

Il y a plusieurs types possibles de **polymorphisme** en POO:

- Les polymorphismes **avec signatures différentes**
  - par **Surcharge / Overload** (aussi nommé « **ad hoc** »)
  - **Paramétrique**
- Les polymorphismes de l'**Héritage**
  - par **Masquage / Shadowing**
  - par **Substitution / Override**

# Le polymorphisme par surcharges / overloading (ad hoc)

C'est le cas où l'on utilise le même nom de méthode mais **un nombre de paramètres différents**

Prenons le cas d'une class **Concessionnaire** possédant une **List<Voiture>** dans laquelle **on ajoutera des voitures**

- Ici notre méthode prend un objet en paramètre

```
public bool AjouterVoiture(Voiture voiture)
```

- Ici notre méthode prend 3 paramètres (on instanciera la voiture)

```
public bool AjouterVoiture(string model, string couleur, int reservoir, int autonomie)
```

# Le polymorphisme paramétrique

C'est le cas où l'on utilise le même nom de méthode, le même nombre de paramètres mais avec **une signature différente au niveau des types**

- Ici notre méthode est signée int

```
public static int Additionner(int a, int b)
```

- Ici notre méthode est signée string

```
public static string Additionner(string a, string b)
```

# Les polymorphismes de l'Héritage

Les polymorphismes par **Masquage** et par **Substitution / Override** interviennent **dans la notion d'Héritage** (*chapitre suivant*)

Ils permettent de faire de la **spécialisation** sur nos **méthodes**



# Héritage

## Le concept de l'héritage

L'**héritage** est un mécanisme fortement utilisé dans la POO

- Une classe peut **hériter** d'une **autre classe**, dans ce cas elle en possédera **les membres** (méthodes / attributs / paramètres / constructeurs), on dit aussi qu'elle **dérive** de l'autre classe
- On parle alors de **classe fille/enfant** (spécialisé) et de **classe mère/parent** (général)
- Pour **réaliser un héritage** en C# il suffit d'**ajouter le caractère :** après le nom de la classe que l'on créé et d'**ajouter la classe dont l'on souhaite hériter à la suite**

```
public class Homme : Mammifere {...}
```

## Exemples réels

Afin de comprendre cette notion d'héritage, rien de tel que quelques exemple basés sur le réel

- Chien est une sorte de la classe Mammifère
- La classe Mammifère est une sorte de la classe Animal
- La classe Animal est une sorte de la classe ÊtreVivant

Chaque **parent** est un plus **général** que son **enfant**

Et inversement, chaque **enfant** est un plus **spécialisé** que son **parent**

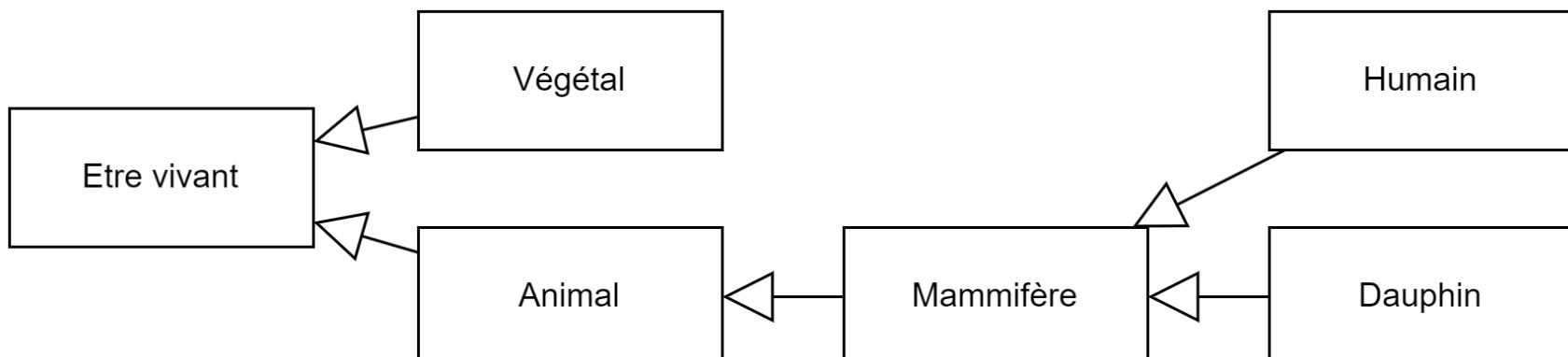
L'**enfant** aura donc **les caractéristiques du parent** auxquelles s'ajoute ses **spécificités**

## Non-multiplicité de l'héritage

Il est possible pour un **parent** d'avoir **plusieurs enfants**

Par contre, **l'inverse est impossible**, un **enfant ne peut pas avoir plusieurs parents** -> **L'héritage multiple est interdit en C#**

On peut définir une sorte de **hiérarchie** entre les objets, un peu comme on le ferait avec **un arbre généalogique**



## Mot clé base

- Lors d'un **héritage**, il est possible d'**accéder aux attributs et aux méthodes de la class mère**

Si l'on souhaite **accéder à un membre de la classe mère pour s'en servir dans la classe enfant**, on doit utiliser le mot-clé **base**

Exemples: **base.\_attr** **base.Prop** **base.Meth()**

- Le mot clé **base()** est également utilisé au niveau d'un **constructeur** pour faire **appel au constructeur de la classe parent**

```
public Mammifere(string nom, int age, string genre) : base(nom, age)
```

- Il est similaire au mot clé **this** qui concerne l'instance

# Les polymorphismes de l'Héritage

Les polymorphismes par **Masquage** et par **Substitution / Override** permettent de faire de la **spécialisation** sur nos **méthodes**

En effet, si on veut **modifier** ou **remplacer** le **comportement de méthodes** d'une **classe mère** dans une **classe fille** cela sera possible avec ces concepts

Ainsi, ces méthodes auront **plusieurs formes** en fonction du **type de l'instance** que l'on utilisera

[Savoir quand utiliser les mots clés override et new](#)

# Les polymorphisme de l'Héritage

## Exemple:

Prenons une class **Mammifere** qui aura la méthode **SeDeplacer()**

**Tout les Mammifères** se déplacent mais de manière **spécifique** (nager, voler, marcher, sauter, ...)

Ce type de polymorphisme permettra de définir des **formes différentes** pour **SeDeplacer()** en fonction du mammifère

Un **Dauphin** se déplace **différemment** d'un **Humain** pourtant se sont tout les deux des **Mammifère**

# Masquage / Shadowing (polymorphisme d'héritage)

Lors du **Masquage**, on aura des **méthodes** dans les classes **mère et fille** de **même nom** mais celle de la fille viendra **remplacer** celle de la mère. En C# il n'est **PAS RECOMMANDÉ** dans une majorité des cas

```
internal class Animal
{
    public string Nom { get; set; }
    public bool EstVivant { get; set; }
    public Animal(string nom, bool estVivant)
    {
        Nom = nom;
        EstVivant = estVivant;
    }
    public void Respirer()
        => Console.WriteLine("L'animal respire");
}
```

```
public class Mammifere : Animal
{
    public string Genre { get; set; }
    public Mammifere(string nom,
                     bool estVivant, string genre)
        : base(nom, estVivant)
    {
        Genre = genre;
    }
    public void Respirer()
        => Console.WriteLine("Le mammifère respire");
}
```

Il est recommandé d'utiliser le mot clé `new` pour le masquage

# Substitution / Override (polymorphisme d'héritage)

Lors de la **Substitution**, on aura des **méthodes** dans les classes **mère et fille de même nom** mais celle de la fille viendra **redéfinir** celle de la mère en ayant la possibilité de la réutiliser.

On utilisera les mots clés **virtual**, **override**, **abstract** et **sealed**.

```
internal class Animal
{
    public string Nom { get; set; }
    public bool EstVivant { get; set; }
    public Animal(string nom, bool estVivant)
    {
        Nom = nom;
        EstVivant = estVivant;
    }
    public virtual void Respirer()
        => Console.WriteLine("L'animal respire");
}
```

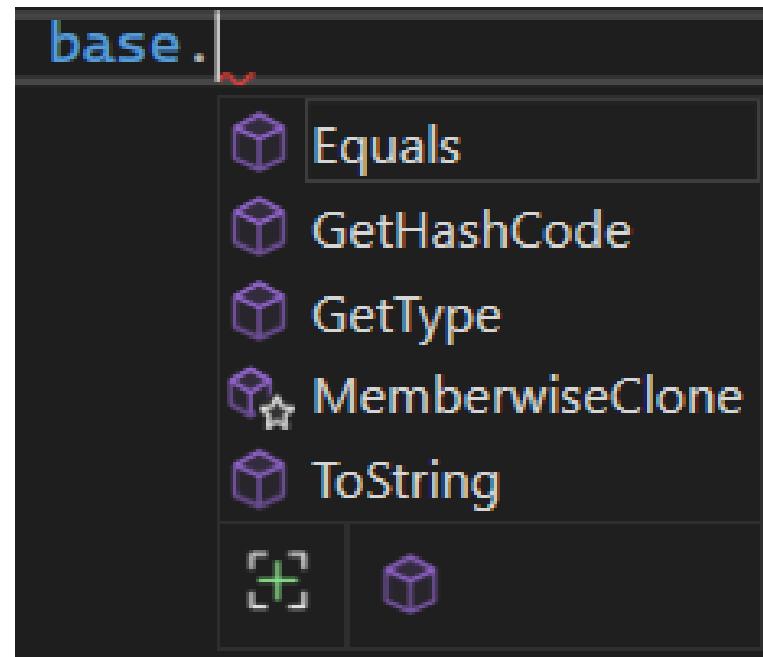
```
public class Mammifere : Animal
{
    public string Genre { get; set; }
    public Mammifere(string nom,
                     bool estVivant, string genre)
        : base(nom, estVivant)
    {
        Genre = genre;
    }
    public override void Respirer()
    {
        base.Respirer(); // appeler une méthode du parent
        Console.WriteLine("Le mammifère respire");
    }
}
```

## La classe object

Chaque classe du C# va **automatiquement hériter** d'une classe qui se nomme « **object** ».

Cette classe comporte **une série de méthodes** qui seront ainsi automatiquement hérités par les classes enfants.

- **ToString** = représentation textuelle de l'objet
- **Equals** = comparaison d'égalité
- **GetType** = récupération du type
- **GetHashCode** = Hash de l'objet
- **MemberwiseClone** = clone de l'objet avec attributs à l'identique



## La méthode `.ToString()`

L'exemple le plus courant est sans doute celui de l'héritage de la méthode `.ToString()` qui est la méthode utilisée lorsque l'on souhaite récupérer la représentation textuelle de l'objet.

```
// Animal
public override string ToString()
{
    return this.GetType().Name
        + $" : Nom = {Nom}"
        + $" , EstVivant = {EstVivant}";
}
```

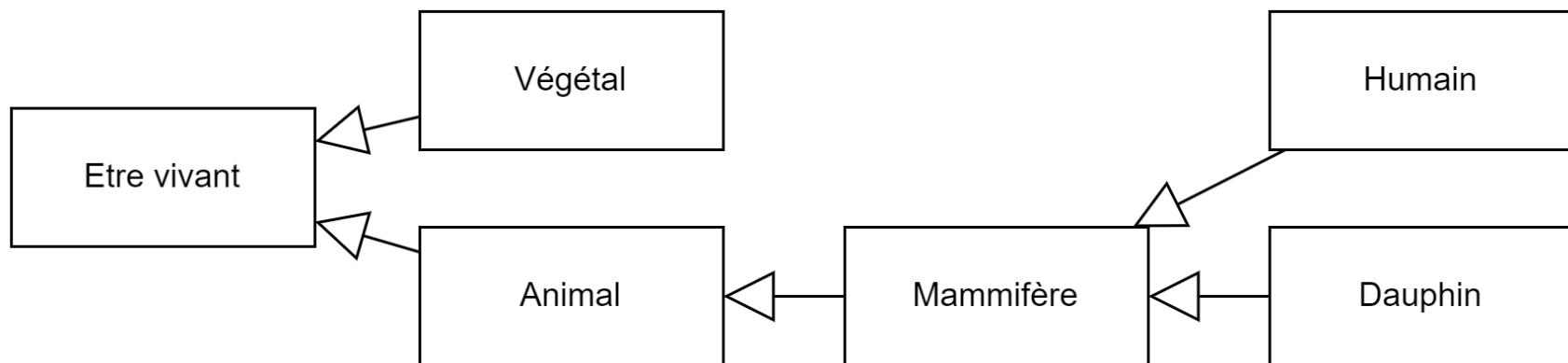
```
// Mammifère
public override string ToString()
{
    return base.ToString()
        + $" , Genre = {Genre}";
}
```

## Les classes abstraites (`abstract`)

- Une classe `abstract` est une classe particulière qui **ne peut pas être instanciée**
- **Impossible d'utiliser les constructeurs et l'opérateur new**
- Pour être **utilisables**, les **classes abstraites** doivent être **héritées** et leurs méthodes abstraites redéfinies
- Elle servent à **représenter** un **concept** ou un **objet** qui **n'a pas de sens tel quel car trop général**, ce seront ses **spécialisation / enfants** qui seront **instanciées** (possiblement indirectement)

## Les classes abstraites (abstract)

Dans notre exemple précédent, nous pourrions avoir `EtreVivant`, `Vegetal`, `Animal` et `Mammifere` en abstrait car par la présence de leurs spécialisations, leur **instanciation devient incohérente, abstraite.**



Autre exemple, si nous avions supprimer les classes `Humain` et `Dauphin` et ajouté un attribut `Especie` à `Mammifere`. celui-ci pourrait ne plus être abstrait.

## Les classes et les méthodes abstraites (Abstract)

De la même façon, une **méthode abstraite** est une méthode qui ne contient **pas d'implémentation**

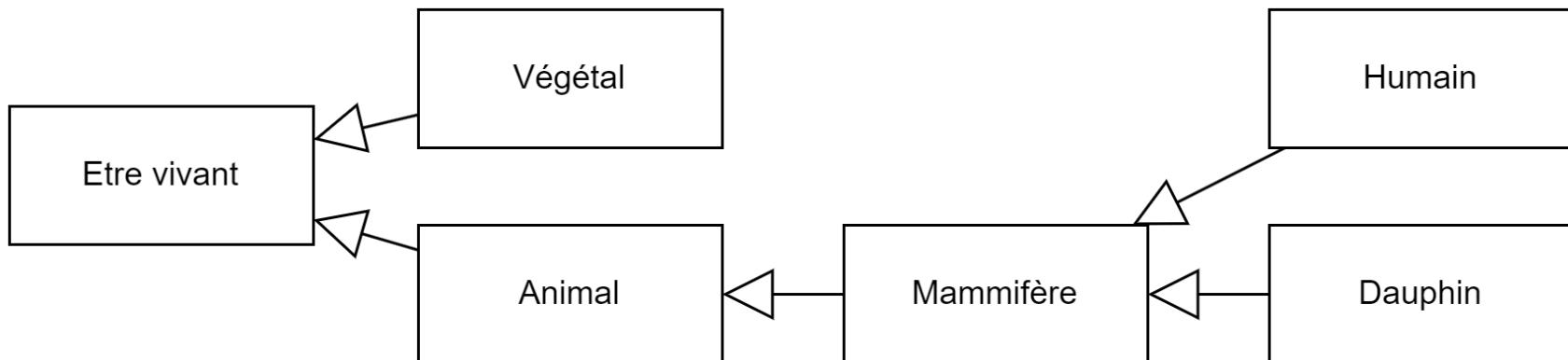
- Elle n'a **pas de corps** (pas de block de code)
- Une méthode **abstract** sera toujours dans une class **abstract**
- Pour être utilisables, les méthodes **abstract** doivent être redéfinies avec un **override**

## Les classes et les méthodes sealed (scellée)

Nous utilisons le mot-clé `sealed` quand une **classe** ne devra **plus être héritée** ou qu'une **méthode** ne devra **plus être substituée/override**

- La classe sera la **dernière** de la lignée
- La méthode ne pourra **plus être substituée**

Dans notre exemple on pourrait avoir `Humain` et `Dauphin` en `sealed`



# Type de variables et type d'instance

Si on reprends notre exemple, **un Dauphin EST un Animal**, on pourra alors faire :

```
Animal dph = new Dauphin();
```

Ici la **variable** sera de **type** `Animal` mais pourra aussi **référencer** des **instances** de **classes dérivées** de `Animal`.

Autre exemple :

```
List<Animal> animaux = new List<Animal>()
{
    new Baleine(), new Dauphin(), new ChauveSouris(), new Pigeon(), new Humain()
};
```

# Cast et Opérateurs is et as dans l'héritage

Précédemment, nous avons vu les **cast implicites** et **explicites** et les **opérateurs de cast** `is` et `as`. Ils prennent tout leur sens dans le cadre de l'héritage.

Si nous prenons l'exemple précédent avec la liste, nous pourrons ainsi itérer sur la liste puis **convertir chaque élément si besoin**.

```
foreach (Animal animal in animaux)
{
    if (animal is Dauphin dauphin)
    {
        Console.WriteLine(dauphin.GetType().Name + " => Cet Animal est bien une baleine.");
        dauphin.Nager();
    }
}
```



# Interfaces

# Pourquoi les interfaces existent-elles ?

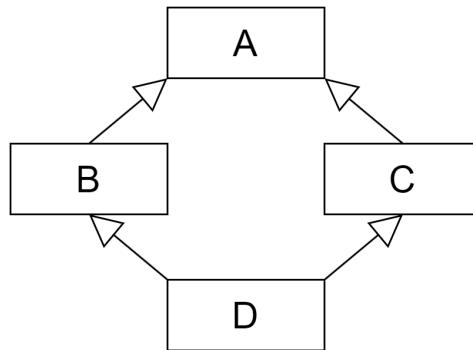
- Imaginons que nous cherchions à **regrouper** plusieurs **classes** qui ont un **comportement commun** sous **un même type**
- Nous pourrions être tenté d'utiliser l'**héritage**
- Cependant celui-ci n'est **valide** que lorsque l'on peut dire:  
**"ClasseB est un cas spécifique de ClasseA"**
- Il existe certains cas qui **ne correspondront pas** et où cette affirmation sera **invalidé**

## Exemple Concret

- Un `Avion` peut voler, il aura les méthodes `décoller()` et `atterrir()`
- Un `Oiseau` peut voler, il aura les **mêmes méthodes**
- Si nous voulions créer une liste d'**objets volants** pour les faire **décoller** successivement, il nous faudrait un type `Volant`
- Cependant, l'`Oiseau` est un `Animal` et l'`Avion` est une `Machine`
- On aurait donc besoin de pouvoir **hériter de plusieurs classes simultanément**, par **Héritage Multiple**
- Or, en C#, c'est **Impossible**
- Le Python le permet mais un problème complexe en résulte

## Héritage en diamant impossible

- C# ne supporte pas l'**héritage multiple** pour éviter le problème de **l'héritage en diamant**.
- Si **B** et **C** héritent de **A** et **D** hérite de **B** et **C**, **quelle version de A doit être utilisée** par **D** ?



- Les interfaces offrent une **solution**, permettant à une classe d'**implémenter plusieurs interfaces**.

# Pourquoi les interfaces existent-elles ?

- Les **interfaces** permettent de définir des **contrats** que les classes doivent respecter.
- Elles facilitent la **modularité** et la **réutilisabilité** du code.
- Elles permettent d'implémenter une forme de **polymorphisme** sans héritage multiple.

## Qu'est-ce qu'une interface ?

- Une interface est un **type** en C# qui ne contient que des **méthodes abstraites**.
- Les classes qui implémentent une interface doivent **redéfinir** toutes ses méthodes.
- Les interfaces permettent de définir des **comportements communs** sans imposer une hiérarchie d'héritage.

## Interfaces comme contrat

- Une interface définit un **contrat** : une classe qui l'implémente s'engage à fournir des implémentations pour toutes ses méthodes.
- Exemple : une interface `Volant` peut définir des méthodes `Decoller` et `Atterrir`.

```
public interface IVolant {  
    void Decoller();  
    void Atterrir();  
}
```

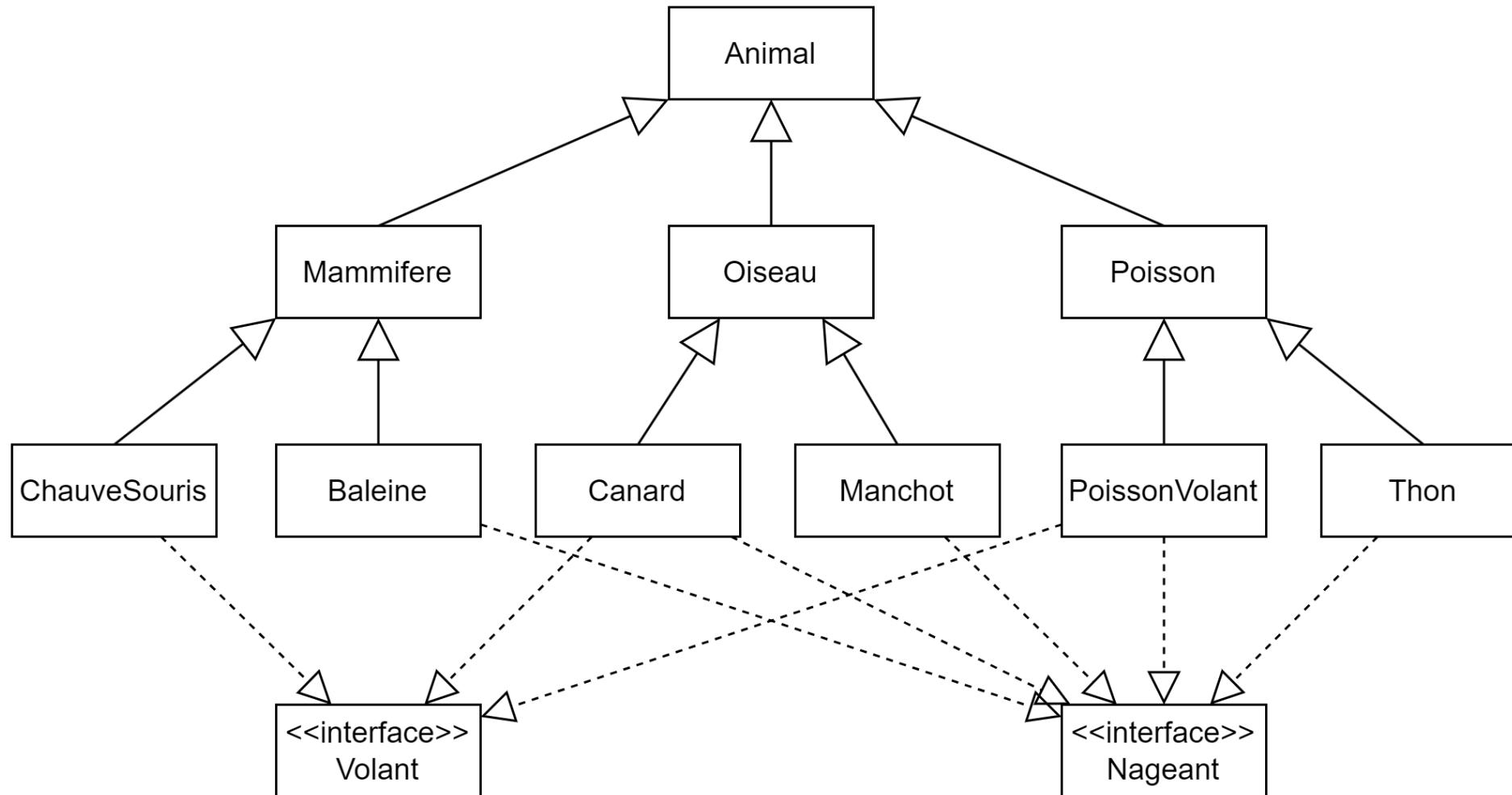
*Notez aussi la présence de la Syntaxe `IInterface`, norme en C#*

# Exemple de classes et interfaces

```
public abstract class Animal {  
    // Attributs et méthodes communs  
}  
  
public abstract class Mammifere : Animal {  
    // Attributs et méthodes spécifiques aux mammifères  
}  
  
public class Baleine : Mammifere, INageant {  
    public void Nager() {  
        Console.WriteLine("La baleine nage.");  
    }  
}
```

```
// Interfaces  
public interface IVolant {  
    void Decoller();  
    void Atterrir();  
}  
  
public interface INageant {  
    void Nager();  
}
```

# Démonstration avec diverses classes et interfaces



# Implémentation de 2 classes

```
public class Canard : Mammifere, IVolant, INageant {  
    public void Decoller() {  
        Console.WriteLine("Le canard décolle.");  
    }  
  
    public void Atterrir() {  
        Console.WriteLine("Le canard atterrit.");  
    }  
  
    public void Nager() {  
        Console.WriteLine("Le canard nage.");  
    }  
}
```

```
public class PoissonVolant : Poisson, IVolant, INageant {  
    public void Decoller() {  
        Console.WriteLine("Le poisson volant décolle.");  
    }  
  
    public void Atterrir() {  
        Console.WriteLine("Le poisson volant atterrit.");  
    }  
  
    public void Nager() {  
        Console.WriteLine("Le poisson volant nage.");  
    }  
}
```

# Exemple d'utilisation

```
Animal[] zooDeLille = new Animal[] {  
    new Baleine(),  
    new Canard(),  
    new Thon(),  
    new PoissonVolant(),  
    new ChauveSouris(),  
    new PoissonVolant(),  
    new Pigeon(),  
    new Pigeon(),  
};
```

```
foreach (Animal animal in zooDeLille) {  
    Console.WriteLine(animal);  
  
    if (animal is Poisson) {  
        Console.WriteLine("C'est un poisson!");  
    }  
  
    if (animal is IVolant volant) {  
        volant.Decoller();  
        volant.Atterrir();  
    }  
  
    if (animal is INageant nageant) {  
        nageant.Nager();  
    }  
}
```

## Résumé

- Les interfaces en C# offrent une manière de définir des **contrats de comportement** sans les contraintes de l'héritage multiple.
- Elles permettent d'implémenter le **polymorphisme** de manière **flexible et modulaire**.
- Utilisées correctement, elles améliorent la **maintenabilité** et la **réutilisabilité** du code.



# Génériques

# Introduction

La **généricité** en C# permet de définir des **classes**, des **interfaces** et des **méthodes** avec des **types paramétrés**.

Cela permet d'écrire du code plus flexible et réutilisable tout en garantissant la sécurité des types à la compilation.

On parle en général de **classes "Moule"** faites pour **accueillir** et **s'adapter** à d'**autres classes**.

L'exemple le plus connu est celui des **collections**, dans `List<T>` la **liste s'adapte au type interne** pour faire des opérations avec ce type.

## 1. Classes et Interfaces Génériques

Les **classes** et **interfaces** peuvent être définies avec des **paramètres de type**, ce qui permet de travailler avec des **types spécifiques** tout en maintenant une **structure générale**.

Ce **type ne pourra pas changer** dans le temps **lors de l'utilisation** de la classe, le **typage reste Fort**.

# Exemple 1

Classe générique :

```
public class Boite<T>
{
    private T valeur;

    public T Valeur
    {
        get { return valeur; }
        set { valeur = value; }
    }
}
```

Utilisation :

```
Boite<int> boiteEntier = new Boite<int>();
boiteEntier.Valeur = 123;
int valeurEntier = boiteEntier.Valeur;
Console.WriteLine(valeurEntier); // Affiche 123

Boite<string> boiteString = new Boite<string>();
boiteString.Valeur = "Bonjour";
string valeurString = boiteString.Valeur;
Console.WriteLine(valeurString); // Affiche Bonjour
```

## 2. Méthodes Génériques

Les **méthodes** peuvent également être **génériques**, ce qui permet de définir des méthodes avec des **paramètres de type**.

## Méthode générique :

```
public class Util
{
    public static void ImprimerTableau<T>(T[] tableau)
    {
        foreach (T élément in tableau)
        {
            Console.WriteLine(élément + " ");
        }
        Console.WriteLine();
    }
}
```

## Utilisation :

```
int[] tableauEntiers = { 1, 2, 3, 4, 5 };
string[] tableauStrings = { "un", "deux", "trois" };

Util.ImprimerTableau(tableauEntiers); // Affiche 1 2 3 4 5
Util.ImprimerTableau(tableauStrings); // Affiche un deux trois
```

### 3. Bornes de Types

Vous pouvez **restreindre les types** qui peuvent être **utilisés** avec des **paramètres de type** en utilisant des **contraintes** :

- `where T : class` pour les **références de classe**
- `where T : struct` pour les **types par valeurs**
- `where T : new()` pour les types avec un **constructeur sans paramètre**
- `where T : BaseClass` pour les types **dérivés d'une classe spécifique**, le plus utile

## Exemple de bornes de types : Utilisation :

```
public class Util
{
    public static void ImprimerNombres<T>(T[] tableau) where T : struct, IComparable
    {
        foreach (T nombre in tableau)
        {
            Console.WriteLine(nombre + " ");
        }
        Console.WriteLine();
    }
}
```

```
int[] tableauEntiers = { 1, 2, 3, 4, 5 };
double[] tableauDoubles = { 1.1, 2.2, 3.3 };

Util.ImprimerNombres(tableauEntiers);
// Affiche 1 2 3 4 5

Util.ImprimerNombres(tableauDoubles);
// Affiche 1.1 2.2 3.3

// Util.ImprimerNombres(new string[ ]{"un", "deux"});
// Erreur de compilation
```

# Exemple Complet

L'exemple suivant illustre l'**utilisation complète des génériques** en C#.

Nous allons :

- Définir des classes Vis, VisCruciforme et VisPlate
- Utiliser un Tournevis générique
- Utiliser un TournevisPlat, héritant de Tournevis
- Utiliser un TournevisUniversel, pour démontrer l'utilisation des méthodes génériques.

# Définitions des Vis utilisées

```
public abstract class Vis
{
    protected string taille;

    public Vis(string taille)
    {
        this.taille = taille;
    }

    public abstract void Serrer();
    public abstract void Desserrer();
}

public class VisCruciforme : Vis
{
    public VisCruciforme(string taille) : base(taille) { }

    public override void Serrer()
    {
        Console.WriteLine("Serrer la vis cruciforme de taille " + taille);
    }

    public override void Desserrer()
    {
        Console.WriteLine("Desserrer la vis cruciforme de taille " + taille);
    }
}
```

```
public class VisPlate : Vis
{
    public VisPlate(string taille) : base(taille) { }

    public override void Serrer()
    {
        Console.WriteLine("Serrer la vis plate de taille " + taille);
    }

    public override void Desserrer()
    {
        Console.WriteLine("Desserrer la vis plate de taille " + taille);
    }
}
```

# Classe Générique

```
public class TournevisAEmbout<T> where T : Vis
{
    // s'adaptera à la vis passée à linstanciation et la définition
    public void Utiliser(T vis)
    {
        vis.Serrer();
        vis.Desserrer();
    }
}
```

## Utilisation :

```
VisCruciforme visCruciforme = new VisCruciforme("M4");
TournevisAEmbout<VisCruciforme> tournevisCruciforme = new TournevisAEmbout<VisCruciforme>();
tournevisCruciforme.Utiliser(visCruciforme);
```

# Héritage Générique

```
public class TournevisPlat : TournevisAEmbout<VisPlate>
{
    // En héritant on spécifie le type ici
    // Possibilité d'ajouter des méthodes spécifiques pour les vis plates
}
```

## Utilisation :

```
VisPlate visPlate = new VisPlate("M5");
TournevisPlat tournevisPlat = new TournevisPlat();
tournevisPlat.Utiliser(visPlate);
```

# Méthode Générique

```
public static class TournevisUniversel
{
    public static void Utiliser<T>(T vis) where T : Vis
    {
        // s'adaptera à la vis passée à l'appel
        vis.Serrer();
        vis.Desserrer();
    }
}
```

Utilisation :

```
TournevisUniversel.Utiliser(new VisCruciforme("M4"));
TournevisUniversel.Utiliser(new VisPlate("M5"));
```

# Quelle lettre utiliser pour le Paramètre de Type ?

En C#, les **lettres couramment utilisées** pour les paramètres de type générique sont :

- **T** : Pour représenter un **type générique (Type)**.
- **TKey** : Pour les **clés**.
- **TValue** : Pour les **valeurs**.
- **T1, T2, etc.** : Pour d'autres types génériques **supplémentaires**.

Ces lettres sont choisies pour leur **clarté** et par **convention** dans la communauté C#.

## Conclusion

La générnicité en C# permet d'écrire des **classes**, des **interfaces** et des **méthodes** plus **flexibles** et **réutilisables** tout en maintenant la sécurité des types.

Elle facilite le développement de **bibliothèques génériques** et améliore la lisibilité et la maintenabilité du code.



# Collections

Utopios® Tous droits réservés

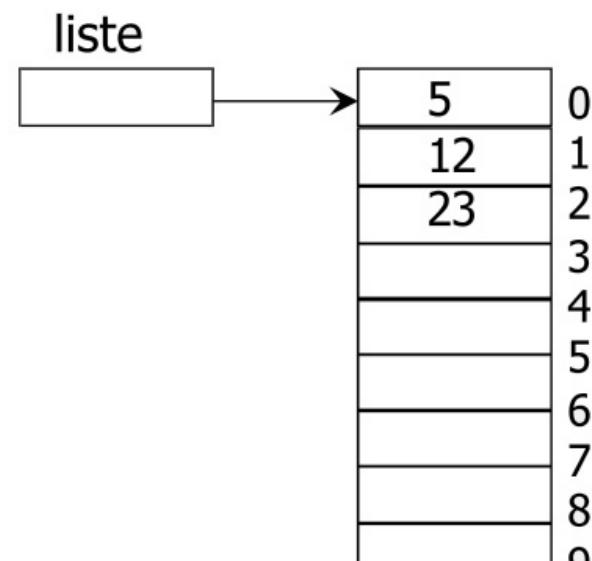
# Tableaux de primitives

- Déclaration :
  - Tableau de nombres entiers

```
int[] tab;
```
  - **tab** fera **référence** à un tableau d'entiers
- Instanciation du tableau

```
tab = new int[11];
```

```
tab[0] = 5;  
tab[1] = 12;  
tab[3] = 23;  
for (int i = 0; i < tab.Length; i++) {  
    Console.WriteLine(tab[i]);  
}
```



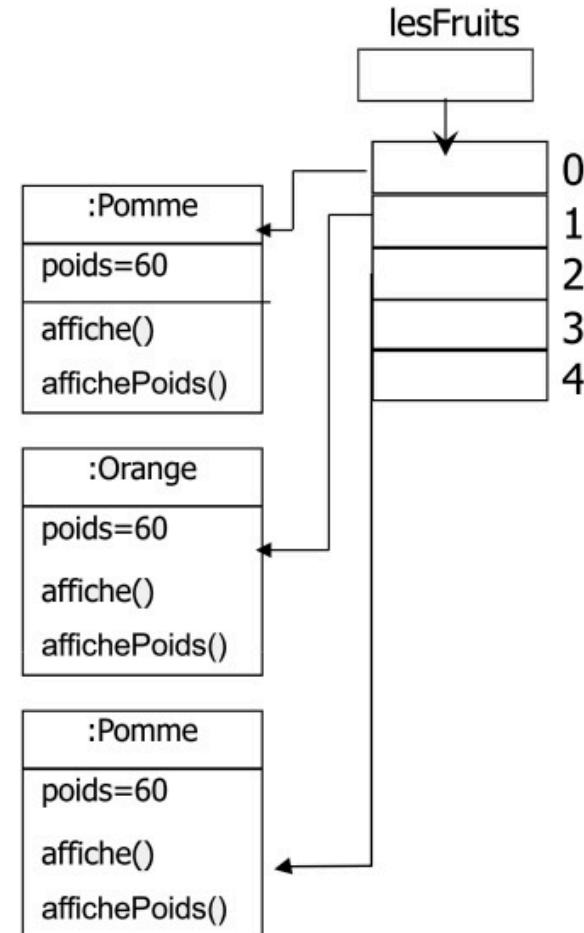
# Tableaux d'objets

- Déclaration d'un Tableau d'objets Fruit :

```
Fruit[] lesFruits;
```

- Instanciation du tableau
  - lesFruits = new Fruit[5];
- Création des objets :

```
lesFruits[0] = new Pomme(60);
lesFruits[1] = new Orange(100);
lesFruits[2] = new Pomme(55);
```



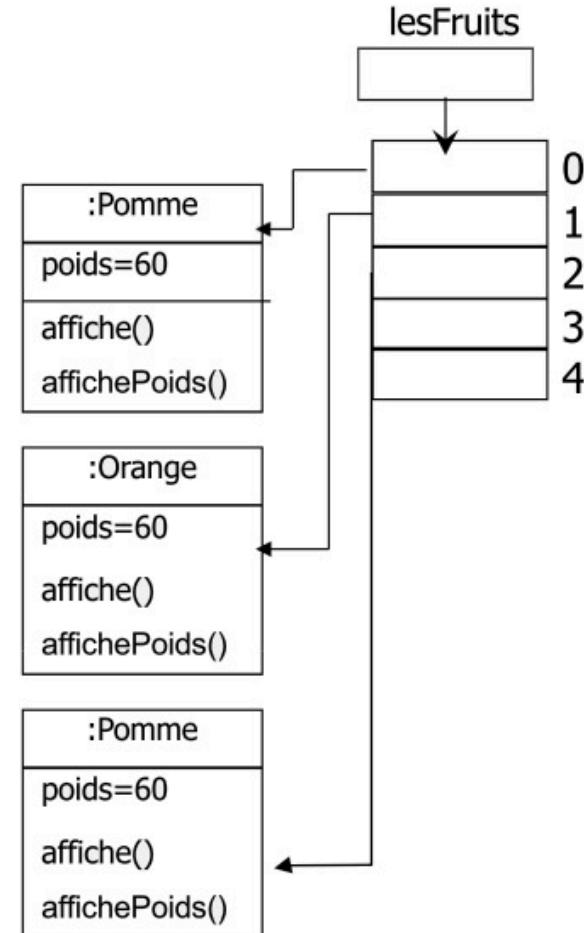
# Tableaux d'objets

- Un tableau d'objets est un tableau de **références**
- Manipulation des objets :

```

for (int i = 0; i < lesFruits.Length; i++) {
    lesFruits[i].Affiche();
    if (lesFruits[i] is Pomme)
        ((Pomme)lesFruits[i]).AffichePoids();
    else
        ((Orange)lesFruits[i]).AffichePoids();
}

```



## Namespace System.Collections

C# propose le **Namespace Collections** qui offre un socle riche et des implémentations d'**objets de type collection** enrichies au fur et à mesure des versions de .NET.

Le Namespace Collections possède **deux grandes familles** chacune définies par une **Interface Générique** :

- **System.Collections.Generic.ICollection<T>** : pour gérer un groupe d'objets
- **System.Collections.Generic.IDictionary< TKey, TValue >** : pour gérer des éléments de type **paires de clé/valeur**, ils sont assimilables aux **dictionnaires** d'autres langages

# Namespace Collections

Le **Namespace Collections** définit enfin :

- Deux interfaces pour le **parcours** de certaines **collections** :  
`IEnumerator` et `IEnumerator<T>`.
- Une interface pour permettre le **tri** de certaines collections :  
`IComparer<T>`

## Interfaces des Collections

- Les **interfaces génériques de collection** principales regroupent différents types de collections.
- Ce sont des **interfaces** donc elles **ne fournissent pas d'implémentation** !

# Collections

- Une collection est un **tableau dynamique d'objets** de type `object` (pas de types primitifs directement, mais les types valeur sont utilisables via `System.Object`).
- Une collection fournit un **ensemble de méthodes** qui permettent :
  - **Ajouter** un nouveau objet dans le tableau
  - **Supprimer** un objet du tableau
  - **Rechercher** des objets selon des critères
  - **Trier** le tableau d'objets
  - **Filtrer** les objets du tableau

# Quand utiliser une collection et laquelle utiliser ?

- Dans un problème, les **tableaux** peuvent être utilisés quand la **dimension** du tableau est obligatoirement **fixe**.
- **Dans le cas contraire, il vaut mieux utiliser les collections**, que nous allons lister ensuite.

# Quand utiliser une collection et laquelle utiliser ?

- **List**, le plus souvent, pour **regrouper des éléments simplement**
- **HashSet** pour **éviter les doublons**
- **Dictionary< TKey, TValue >** pour des **correspondances clé-valeur**
- **Queue** et **Stack** dans des cas particuliers (cf. **FIFO/LIFO**)
- Les version **Sorted...<>** lorsqu'un tri automatique est nécessaire
- Si l'on fait du **Multithreading**, on préférera les **collections Thread-safe** comme **ConcurrentDictionary< TKey, TValue >**

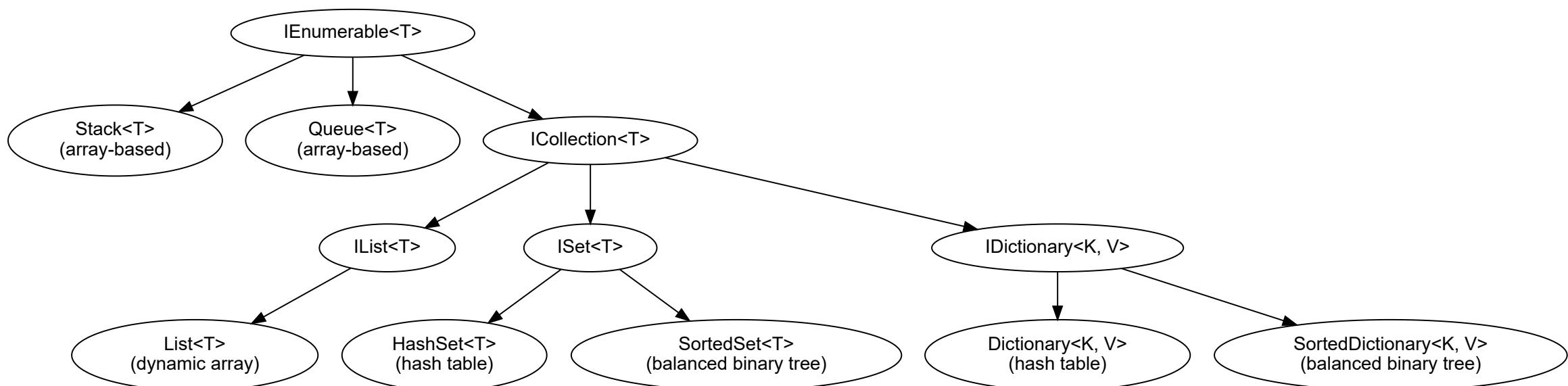
# Collections

C# fournit plusieurs types de collections :

- `List<T>`
- `HashSet<T>`
- `Dictionary< TKey, TValue >`
- `Queue<T>`
- `Stack<T>`
- Etc...
- Dans cette partie du cours, nous allons présenter uniquement comment utiliser les collections de type **List**, **Set** et **Dictionary**

# Architecture des Collections

Voici à quoi ressemble l'architecture (**implémentations d'interfaces**) pour les **collections génériques** principales en c#.



# A quoi ressemble l'interface Collection

```
public interface ICollection<T> : IEnumerable<T> {
    // Basic operation
    int Count { get; }
    bool IsReadOnly { get; }
    void Add(T item);
    void Clear();
    bool Contains(T item);
    void CopyTo(T[] array, int arrayIndex);
    bool Remove(T item);

    // IEnumerable<T> implementation
    IEnumerator<T> GetEnumerator();
}
```

## IEnumerable

L'interface Générique **IEnumerable** est plus globale, elle rassemble l'ensemble des choses sur lesquelles on peut itérer, notamment les collections mais aussi les **Générateurs** (mot clé `yield` dans des fonctions) et les **Queryable** de Linq (chapitre suivant).

Pour **itérer** les objets de `IEnumerable`, utiliser la boucle `foreach`.

## IEnumerator

- **IEnumerator** est une interface qui itère les éléments. Elle permet de parcourir la liste et de modifier les éléments, ou d'utiliser un générateur.
- **L'interface Ienumerator a trois méthodes qui sont mentionnées ci-dessous :**
  1. **bool MoveNext()** – Cette méthode renvoie true si l'itérateur a plus d'éléments.
  2. **T Current { get; }** – Il renvoie l'élément actuel.
  3. **void Reset()** – Cette méthode réinitialise l'énumérateur à sa position initiale.

# IEnumerator

- La collection de type `IEnumerator` du namespace `System.Collections` est souvent utilisée pour afficher les objets d'une autre collection.
- En effet il est possible d'obtenir un `IEnumerator` à partir de chaque collection.

# Exemple

```
// Cr ation d'une liste de Fruit.  
List<Fruit> fruits = new List<Fruit>();  
// Ajouter des fruits   la liste  
fruits.Add(new Pomme(30));  
fruits.Add(new Orange(25));  
fruits.Add(new Pomme(60));  
// Cr ation d'un IEnumarator   partir de cette liste  
IEnumerator<Fruit> it = fruits.GetEnumerator();  
// Parcourir l'IEnumerator :  
while (it.MoveNext()) {  
    Fruit f = it.Current;  
    f.Affiche();  
}
```



# List

# List

- `List<T>` est une classe de `System.Collections.Generic` qui implémente l'interface `IList<T>`.
- Elle permet de **stocker des éléments** de manière **ordonnée** avec des **index**, comme un tableau mais **sa taille est variable** en fonction des éléments qu'elle contient.
- Les éléments peuvent être **ajoutés**, **accédés**, **modifiés**, et **supprimés** par leur index.

# Utilisation de List

- Déclaration d'une `List` pour stocker des objets `Fruit` :

```
List<Fruit> fruits = new List<Fruit>();
```

- Ajout de deux objets `Fruit` à la liste :

```
fruits.Add(new Pomme(30));  
fruits.Add(new Orange(25));
```

- Affichage des objets de la liste :

```
for (int i = 0; i < fruits.Count; i++) {  
    Console.WriteLine(fruits[i]);  
}
```

# Utilisation de List

- Utilisation de la boucle `foreach` pour afficher les objets :

```
foreach (Fruit f in fruits) {  
    Console.WriteLine(f);  
}
```

- Suppression du deuxième objet de la liste :

```
fruits.RemoveAt(1);
```

# Exemple d'utilisation de List

```
// Déclaration d'une liste de type Fruit
List<Fruit> fruits;
// Création de la liste
fruits = new List<Fruit>();
// Ajout de 3 objets Pomme,Orange et Pomme à la liste
fruits.Add(new Pomme(30));
fruits.Add(new Orange(25));
fruits.Add(new Pomme(60));
// Parcourir tous les objets
for (int i = 0; i < fruits.Count; i++) {
    // Faire appel à la méthode affiche()
    // de chaque Fruit de la liste
    fruits[i].Affiche();
}
// Une autre manière plus simple pour parcourir une liste
foreach(Fruit f in fruits) { // Pour chaque Fruit de la liste
    // Faire appel à la méthode affiche() du Fruit f
    f.Affiche();
}
```



# Set

## HashSet<T>

- **HashSet** est une collection qui ne peut pas contenir d'éléments en double.
- Modélise les **ensembles mathématiques**.
- Exemple :
  - Trouver chaque mot/lettre utilisé dans un Livre
  - Une main de poker (pas possible d'avoir plusieurs cartes identiques).

## HashSet

- HashSet utilise une **table de hachage** pour le stockage.
- Contient uniquement des éléments uniques.
- Les éléments sont triés selon leur **hash**

Un **hash** est une **valeur fixe générée par une fonction de hachage** à partir d'**une entrée de taille variable**, servant à **identifier rapidement et de manière unique** les données d'origine.

## SortedSet

- **SortedSet** maintient ses éléments dans l'**ordre croissant** selon **leur contenu**, s'applique surtout aux primitifs.
- Les objets seront triés différemment s'ils implémentent **IComparable<T>**.
- Fournit des opérations supplémentaires pour exploiter cet ordre.
- Utilisé pour les ensembles ordonnés naturellement comme les **listes de mots ou de chiffres**.

# Exemple

```
HashSet<string> set = new HashSet<string>();
set.Add("Java");
set.Add("Python");
set.Add("Python3");
set.Add("C++");
set.Add("C++");
set.Add("C++");
Console.WriteLine("HashSet : " + string.Join(", ", set));

// Démo pour SortedSet
SortedSet<string> sortedSet = new SortedSet<string>();
sortedSet.Add("Java");
sortedSet.Add("Python");
sortedSet.Add("Python3");
sortedSet.Add("C++");
sortedSet.Add("C++");
sortedSet.Add("C++");
Console.WriteLine("SortedSet : " + string.Join(", ", sortedSet));

// Méthodes pour SortedSet
Console.WriteLine("1. Premier élément : " + sortedSet.First());
Console.WriteLine("2. Dernier élément : " + sortedSet.Last());
SortedSet<string> headset = new SortedSet<string>(sortedSet.GetViewBetween(sortedSet.First(), "Python"));
Console.WriteLine("3. Sous-ensemble avant 'Python' : " + string.Join(", ", headset));
```



# Dictionary

## Interface **IDictionary<K,V>**

- **IDictionary** est une interface pour les collections qui **associent** des **clés** aux **valeurs** correspondantes.
- **Ne peut pas contenir de clés en double.**
- Chaque **clé** correspond à au plus **une valeur**.
- **Pas d'index**

# Exemple de Dictionary

Un panier dans un site e-commerce :

Produit: string ou Produit	Quantité : int
"Pomme"	3
"Banane"	4
"Bière"	4
"Pâtes Tortellini 5kg"	40
"Pâtes Spaghetti 1kg"	5
"Boisson énergisante 50cl"	12
"Concombre"	1
"Cookie nougatine"	128

## Dictionary<TKey,TValue>

- **Dictionary<TKey,TValue>** est une classe qui **implémente l'interface IDictionary**.
- Ajout d'**objets de même type** et récupération par **clé**.
- **Itération** à travers **chaque couple clé-valeur** (**KeyValuePair<TKey, TValue>**) de la **collection** via **foreach**.
- Triés par **hash** comme pour **HashSet** mais associe des valeurs aux clés.

## SortedDictionary< TKey, TValue >

- **SortedDictionary< TKey, TValue >** maintient ses mappages dans l'**ordre croissant des clés**.
- Même fonctionnement que **SortedSet< T >** mais associe des valeurs aux clés.
- Utilisé pour les collections **ordonnées** de paires clé/valeur.

# Exemple

```
// Démo pour le Dictionary
Dictionary<string, int> dictionary = new Dictionary<string, int>();
dictionary["Java"] = 20;
if (!dictionary.ContainsKey("Java"))
    // si ne contient pas "Java" je l'ajoute
    dictionary["Java"] = 22;
dictionary["Python"] = 10;
dictionary["C++"] = 30;
Console.WriteLine("\nDictionary : " + string.Join(", ", dictionary));

// Méthodes pour Dictionary
Console.WriteLine("1. Nombre d'entrées du Dictionary : "
    + dictionary.Count);

Console.WriteLine("2. Valeur associée à 'Java' : "
    + dictionary["Java"]);

Console.WriteLine("3. Est-ce que 'Test' est présent ? : "
    + dictionary.ContainsKey("Test"));

Console.WriteLine("4. Suppression de l'entrée avec la clé 'Python' : ");
dictionary.Remove("Python");

Console.WriteLine("Nouveau Dictionary : "
    + string.Join(", ", dictionary));
```

```
foreach (KeyValuePair<string, int> entry in dictionary)
{
    Console.WriteLine(entry);
    Console.WriteLine("Clé :" + entry.Key);
    Console.WriteLine("Valeur :" + entry.Value);
}

foreach (KeyValuePair<string, int> entry in dictionary)
{
    Console.WriteLine(entry);
    Console.WriteLine("Clé :" + entry.Key);
    Console.WriteLine("Valeur :" + entry.Value);
}
```



# Exceptions

## Cas exceptionnels

- Les **programmes** doivent souvent gérer des **situations exceptionnelles**, rendant le code **complexe** et difficile à lire.
- Exemples : saisie utilisateur en int, division par zéro
- En algorithmie, on appelle ces **situations exceptionnelles** des **Exceptions**
- C# introduit un **mécanisme de gestion des exceptions** pour **séparer le code utile du traitement de ces cas exceptionnels**.

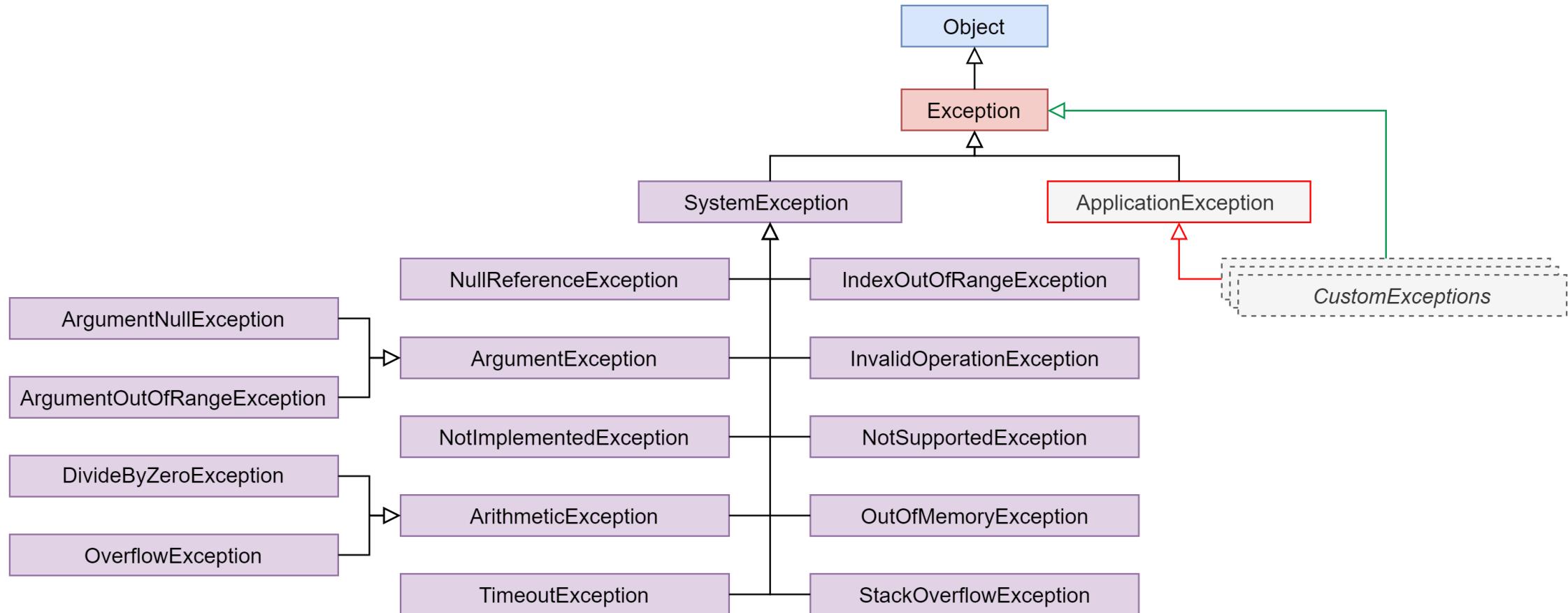
## Concept d'Exception et de throw

- **Exception** est la classe de base pour tous les objets pouvant être **lancés**.
- "Lancé" signifie qu'une **Exception** est **générée/instanciée et propagée dans le programme**.
- Lorsqu'une **condition anormale** survient, on utilise le mot-clé **throw** pour **créer et "lancer"** une instance **interrompant le flux normal d'exécution** du programme.
- Permet de **déclencher explicitement** une exception en réponse à une condition spécifique dans le code.

# Exemple

```
int a = 1;
int b = 0;
if (b == 0) {
    throw new ArithmeticException("Division par zéro");
}
Console.WriteLine(a/b);
```

# Architecture des exceptions les plus connues



`ApplicationException` n'est plus utilisée, on hérite d'`Exception` directement

# Architecture des exceptions les plus connues

- **Exception : base/classe mère**, hérite d' object
- **SystemException** : erreurs prévues par le système.
- **ApplicationException** : erreurs spécifiques à une application particulière, **plus utilisé actuellement**, on préférera directement Exception.

# Exemple

- Saisie de deux entiers, division et affichage du résultat :

```
int Calcul(int a, int b) {  
    return a / b;  
}  
  
Console.WriteLine("Donnez a: ");  
int a = int.Parse(Console.ReadLine());  
Console.WriteLine("Donnez b: ");  
int b = int.Parse(Console.ReadLine());  
int resultat = Calcul(a, b);  
Console.WriteLine("Résultat = " + resultat);
```

## Exécution

- Cas normal :

```
Donnez a: 12  
Donnez b: 6  
Résultat = 2
```

- Cas où b = 0 :

```
Donnez a: 12  
Donnez b: 0  
Unhandled exception. System.DivideByZeroException: Attempted to divide by zero.  
    at Program.Calcul(Int32 a, Int32 b) in .../Program.cs:line 6  
    at Program.Main(String[] args) in .../Program.cs:line 11
```

## Un bug dans l'application

- Le cas du **scénario 2** indique qu'une **erreur fatale s'est produite** dans l'application **au moment de l'exécution**.
- Cette exception est de type `DivideByZeroException`.
- Elle concerne une **division par zero**  
`Attempted to divide by zero`

## Un bug dans l'application

- L'**origine** de cette exception étant la **méthode Calcul** dans la ligne numéro 6.

```
at Program.Calcul(Int32 a, Int32 b) in Program.cs:line 6
```

- Cette exception **n'a pas été traitée** dans Calcul.
- Elle **remonte ensuite vers Main** à la ligne numéro 11 dont elle n'a pas été traitée.

```
at Program.Main(String[] args) in Program.cs:line 11
```

## Un bug dans l'application

- Après l'exception est **signalée** au **CLR** (Common Language Runtime).
- Quand une **exception arrive au CLR**, cette dernière **arrête l'exécution** de l'application, ce qui constitue **un bug fatal**.
- Le fait que le message « Résultat = » **n'a pas été affiché**, montre que l'application **ne continue pas son exécution normale** après la division par zero.

# Un bug dans l'application

- Tout ce chemin est affiché dans la console, c'est ce qu'on appelle la **StackTrace**
- Il est précédé par une ligne indiquant l'**Exception** et le **message lié** (explications)

```
Unhandled exception. System.DivideByZeroException: Attempted to divide by zero.  
  at Program.Calcul(Int32 a, Int32 b) in Program.cs:line 6  
  at Program.Main(String[] args) in Program.cs:line 11
```

# Traiter l'exception

Dans C#, pour traiter les exceptions, on doit utiliser le bloc **try catch** de la manière suivante:

```
int Calcul(int a, int b) {
    int c = a / b;
    return c;
}

Console.WriteLine("Donnez a: ");
int a = int.Parse(Console.ReadLine());
Console.WriteLine("Donnez b: ");
int b = int.Parse(Console.ReadLine());
int resultat = 0;
try {
    // on essaye le bloc suivant
    resultat = Calcul(a, b);
}
catch (DivideByZeroException e) {
    // si l'exception DivideByZeroException est levée
    Console.WriteLine("Division par zero");
}
Console.WriteLine("Résultat = " + resultat);
```

## Scénario 1

```
Donnez a: 12
Donnez b: 6
Résultat = 2
```

## Scénario 2

```
Donnez a: 12
Donnez b: 0
Division par zero
Résultat = 0
```

# Principaux Membres d'une Exception

Tous les types d'exceptions possèdent les membres suivants :

- **Message** : retourne le message de l'exception

```
Console.WriteLine(e.Message);  
// Attempted to divide by zero.
```

- **ToString()** : retourne une chaîne qui contient le type de l'exception et le message de l'exception.

```
Console.WriteLine(e.ToString());  
// System.DivideByZeroException: Attempted to divide by zero.
```

# Principaux Membres d'une Exception

- `StackTrace` : retourne la stacktrace de l'exception

```
Console.WriteLine(e.StackTrace);
```

```
/*
```

Résultat affiché :

```
    at Program.Calcul(Int32 a, Int32 b) in Program.cs:line 6
```

```
    at Program.Main(String[] args) in Program.cs:line 11
```

```
*/
```

# Générer, Relancer ou Jeter une Exception

- Exemple avec une classe **Compte** :

```
public class Compte {  
    private int code;  
    private float solde;  
  
    public void Verser(float montant) {  
        solde += montant;  
    }  
  
    public void Retirer(float montant) {  
        if (montant > solde)  
            throw new Exception("Solde insuffisant");  
        solde -= montant;  
    }  
  
    public float GetSolde() {  
        return solde;  
    }  
}
```

```
public class Application {  
    public static void Main(string[] args) {  
        Compte compte = new Compte();  
        Console.WriteLine("Montant à verser:");  
        float mt1 = float.Parse(Console.ReadLine());  
        compte.Verser(mt1);  
        Console.WriteLine("Solde actuel: " + compte.GetSolde());  
        Console.WriteLine("Montant à retirer:");  
        float mt2 = float.Parse(Console.ReadLine());  
        compte.Retirer(mt2);  
        // Le compilateur ou l'IDE signalent l'Exception  
        // Il nous oblige à nous en occuper  
    }  
}
```

# Deux solutions pour Traiter l'Exception

- Utilisation de **try-catch**

```
try {
    compte.Retirer(mt2);
} catch (Exception e) {
    Console.WriteLine(e.Message);
}
```

- Ou **propagation** de l'exception, elle sera **remontée au niveau supérieur** dans la **Pile d'appel (Call Stack)**, ici, le **CLR**

```
public static void Main(string[] args) {
    // code ...
    compte.Retirer(mt2);
}
```

# Personnaliser les Exceptions

- L'exception générée dans la méthode `Retirer` est une exception **métier** (relative à nos besoins spécifiques).
- Il est plus professionnel de **créer une nouvelle Exception** nommée `SoldeInsuffisantException` de la manière suivante :

```
public class SoldeInsuffisantException : Exception
{
    // Notez la norme de nommage ...Exception
    public SoldeInsuffisantException(string message) : base(message)
    {
    }
}
```

# Utiliser l'Exception personnalisée

```
public void Retirer(float montant)
{
    if (montant > solde)
        throw new SoldeInsuffisantException("Solde Insuffisant");
    solde -= montant;
}
```

# Exemple suivant

Testez avec ces scénarios :

## Scénario 1

```
Montant à verser: 5000
Solde Actuel: 5000.0
Montant à retirer: 2000
Solde Final: 3000.0
```

## Scénario 2

```
Montant à verser: 5000
Solde Actuel: 5000.0
Montant à retirer: 7000
Solde Insuffisant
Solde Final: 5000.0
```

## Scénario 3

```
Montant à verser: azerty
Exception non gérée: System.FormatException: Input string was not in a correct format.
at System.Number.ThrowOverflowOrFormatException(ExceptionResource resource)
at System.Number.ParseSingle(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)
at System.Single.Parse(String s)
at Program.Main(String[] args) in Program.cs:line 13
```

## Améliorer l'application

- Dans le **scénario 3**, nous découvrons qu'une **autre exception est générée** lorsque nous **saisissons une chaîne de caractères** au lieu d'un nombre.
- Cette exception est de type `FormatException`, générée par la méthode `Parse` de la classe `Single(float)`.
- Nous devrions inclure **plusieurs blocs catch** dans la méthode `Main`.
- Similairement à un `else if`, on passera par chaque `catch` jusqu'à **trouver une Exception qui correspond**.

**!\\ Commencer par `Exception` attraperait toutes les Exceptions, rendant les catch suivants inaccessibles (héritage)**

# Multiples catch

```
public static void Main(string[] args)
{
    Compte cp = new Compte();
    try
    {
        Console.WriteLine("Montant à verser: ");
        float montantVerser = float.Parse(Console.ReadLine());
        cp.Verser(montantVerser);
        Console.WriteLine("Solde Actuel: " + cp.GetSolde());

        Console.WriteLine("Montant à retirer: ");
        float montantRetirer = float.Parse(Console.ReadLine());
        cp.Retirer(montantRetirer);
    }
    catch (SoldeInsuffisantException e)
    {
        Console.WriteLine(e.Message);
    }
    catch (FormatException e)
    {
        Console.WriteLine("Problème de saisie");
    }
    Console.WriteLine("Solde Final: " + cp.GetSolde());
}
```

## Le cas MontantNegatifException

- L'exception métier MontantNegatifException

```
public class MontantNegatifException : Exception
{
    public MontantNegatifException(string message) : base(message)
    {
    }
}
```

- La méthode Retirer de la classe Compte

```
public void Retirer(float montant)
{
    if (montant < 0) throw new MontantNegatifException("Montant " + montant + " négatif");
    if (montant > solde) throw new SoldeInsuffisantException("Solde Insuffisant");
    solde -= montant;
}
```

# Application : Contenu de la méthode main

```

Compte cp = new Compte();
try
{
    Console.WriteLine("Montant à verser: ");
    float montantVerser = float.Parse(
        Console.ReadLine());
    cp.Verser(montantVerser);
    Console.WriteLine("Solde Actuel: "
        + cp.GetSolde());

    Console.WriteLine("Montant à retirer: ");
    float montantRetirer = float.Parse(
        Console.ReadLine());
    cp.Retirer(montantRetirer);
}
catch (SoldeInsuffisantException e)
{
    Console.WriteLine(e.Message);
}
catch (FormatException e)
{
    Console.WriteLine("Problème de saisie");
}
catch (MontantNegatifException e)
{
    Console.WriteLine(e.Message);
}
Console.WriteLine("Solde Final: "
    + cp.GetSolde());

```

## Scénario 1

Montant à verser: 5000  
 Solde Actuel: 5000.0  
 Montant à retirer: 2000  
 Solde Final: 3000.0

## Scénario 2

Montant à verser: 5000  
 Solde Actuel: 5000.0  
 Montant à retirer: 7000  
 Solde Insuffisant  
 Solde Final: 5000.0

## Scénario 3

Montant à verser: 5000  
 Solde Actuel: 5000.0  
 Montant à retirer: -2000  
 Montant -2000.0 négatif  
 Solde Final: 5000.0

## Scénario 4

Montant à verser: azerty  
 Problème de saisie  
 Solde Final: 0.0

# Syntaxe multi-catch (filtre)

```
Compte cp = new Compte();
try
{
    Console.Write("Montant à verser: ");
    float montantVerser = float.Parse(Console.ReadLine());
    cp.Verser(montantVerser);
    Console.WriteLine("Solde Actuel: " + cp.GetSolde());

    Console.Write("Montant à retirer: ");
    float montantRetirer = float.Parse(Console.ReadLine());
    cp.Retirer(montantRetirer);
}
catch (Exception e) when (e is SoldeInsuffisantException
                         || e is FormatException
                         || e is MontantNegatifException
                         /*&& booléen/condition*/)
{
    // multi-catch avec filtre d'exception
    Console.WriteLine(e.Message);
}
Console.WriteLine("Solde Final: " + cp.GetSolde());
```

# Le bloc finally

- La syntaxe complète du bloc try est la suivante :

```
try
{
    Console.WriteLine("Traitement Normal");
}
catch (SoldeInsuffisantException e)
{
    Console.WriteLine("Premier cas Exceptionnel");
}
catch (NegativeArraySizeException e)
{
    Console.WriteLine("Deuxième cas Exceptionnel");
}
finally
{
    Console.WriteLine("Traitement par défaut!");
}
Console.WriteLine("Suite du programme!");
```



# Les Lambdas et Délégués

# Introduction

Les **lambdas** et les **délégués** sont des **concepts essentiels** en C# pour la **programmation fonctionnelle** et la **manipulation des méthodes**.

- **Lambdas : Fonctions anonymes** permettant d'écrire des méthodes de manière concise.
- **Délégués : Types** représentant des **références** à des méthodes avec une signature particulière.

# Les Expressions Lambda

Les lambdas sont des expressions concises pour définir des **fonctions anonymes**, souvent utilisées avec des **délégués** et dans **Linq** (cf partie suivante).

```
var op = (a, b) => a + b; // assignation à une variable
int result = op(4, 2); // appel => result = 6
```

# Syntaxe Lambda

Syntaxe **Courte** (Une instruction, avec un `return` implicite)

```
(parameters) => expression
```

Syntaxe **Longue** (Un Bloc d'instruction)

```
delegate (parameters)
{
    instruction;
    instruction;
    return expression;
};
```

# Attribution à une Variable

- **Type différent** selon les cas (**délégués**)
- Possible d'utiliser `var`

```
TypeLambda<TParam, TRetour> fctLambda = (parameters) -> expression;
```

# Les Délégués (Delegates)

- Un délégué est un **type** qui **encapsule une méthode**.  
**Action** et **Func** sont les délégués les plus communs
- Il est possible de les **définir** dans une **Classe** ou directement un **Namespace**.

```
// variables de type délégué
Func<int, int, int> Operation2;
Operation Operation3;
```

```
public class Program
{
    // Déclaration d'un délégué dans une classe
    public delegate int Operation(int a, int b);
    // soit Func<int,int,int>

    public static int Add(int a, int b) => a + b;

    public static void Main()
    {
        // Utilisation du délégué
        // Operation op = new Operation(Add);
        // ou :
        Operation op = Add;
        int result = op(4, 2); // result = 6
        Console.WriteLine(result); // Affiche 6
    }
}
```

## Différences entre Lambda et Délégué

- **Délégué** : **Type** représentant une **référence** à une **méthode** avec une **signature spécifique**.
- **Lambda** : **Syntaxe** pour définir des **méthodes anonymes** de manière concise.

## Types Func<> et Action<>

- **Func<T1,T2,...,TResult>** : Déclare des délégués pour les fonctions **retournant une valeur**.

```
Func<int, int, int> add = (int a, int b) => a + b;  
Func<int, int, int> add = (a, b) => a + b; // implicite
```

- **Action<T1,T2,...>** : Déclare des délégués pour les procédures **retournant void**.

```
Action<string> print = (string msg) => Console.WriteLine(msg);  
Action<string> print = msg => Console.WriteLine(msg); // implicite
```

## Récupération de la référence d'une méthode existante

La **récupération de la référence d'une méthode** existante en C# se fait en mettant **directement** le nom de la méthode **sans les parenthèses** (on ne l'appelle pas).

Cet opérateur permet de **créer une référence à une méthode existante**, en évitant d'écrire une lambda.

Les références de méthodes rendent le code plus lisible et concis en réutilisant des méthodes existantes directement dans les expressions lambda.

# Exemples de récupérations de Méthodes

```
// Méthode statique
Action<string> print = Console.WriteLine;
print("Hello, World!");

// Méthode d'instance d'un objet particulier
List<int> numbers = new List<int> { 3, 1, 2 };
Action sortNumbers = numbers.Sort;
sortNumbers();
numbers.ForEach(Console.WriteLine); // Affiche : 1 2 3

// Transformation de méthode à "fonction"
Func<string, string> toUpperCase = str => str.ToUpper();
string result = toUpperCase("hello");
Console.WriteLine(result); // Affiche : HELLO

// Exemple avec un Constructeur
Func<List<string>> strLstCtor = () => new List<string>();
List<string> stringList = strLstCtor();
stringList.Add("Hello");
```

# Passer des Fonctions en Paramètre

De part l'existence de ces 2 types, les lambdas permettent de **passer des fonctions comme arguments d'autres fonctions.**

```
void AfficheResultat(int a, int b, Func<int, int, int> calcul) {  
    int resultat = calcul(a, b);  
    Console.WriteLine($"Résultat : {resultat}");  
}  
  
AfficheResultat(4, 2, (a, b) => a * b); // Affiche: Résultat : 8
```

# Démo complète

```
// fonctions locales
int Add(int a, int b) { return a + b; }
int Subtract(int a, int b) { return a - b; }
// lambdas stockés
Func<int, int, int> AddBis = delegate (int a, int b) { return a + b; };
Func<int, int, int> AddTer = (int a, int b) => a + b;
// fonction passée en paramètre (callback)
void CalculeEtAffiche(int a, int b, Func<int, int, int> leCalcul)
{
    Console.WriteLine($"Le résultat du calcul est : {leCalcul(a, b)}");
}
// Exemples d'appels
CalculeEtAffiche(4, 2, Add);
CalculeEtAffiche(4, 2, Subtract);
CalculeEtAffiche(4, 2, AddBis);
CalculeEtAffiche(4, 2, delegate (int a, int b) { return a * b; });
CalculeEtAffiche(4, 2, (int a, int b) => a + b);
CalculeEtAffiche(4, 2, (a, b) => a + b);
CalculeEtAffiche(4, 2, Math.Max);
```

# Le Multicast des Délégués

- Les délégués en C# peuvent invoquer plusieurs méthodes à la suite.
- Pour se faire on utilise l'opérateur +

```
Action<int> actions = (x) => Console.WriteLine(x);
actions += (x) => Console.WriteLine(x * 2);
actions += (x) => Console.WriteLine(x * 10);
actions(5); // Affiche 5 puis 10 puis 50
```

- Avec le type Func, c'est le résultat du **dernier appel** qui est **retourné**, les autres retours sont perdus

```
Func<int, int> actions = (x) => x;
actions += (x) => x * 2;
actions += (x) => x * 10;
Console.WriteLine(actions(5)); // Affiche 50
```

## Comparaison avec d'autres langages

- **Java** : Utilise des interfaces fonctionnelles et des lambdas.
- **C++** : Utilise des pointeurs de fonction et des lambdas.
- **Python** : Utilise des fonctions anonymes avec `lambda`.
- **Javascript** : Utilise des fonctions anonymes, souvent appelées "callback".

En C#, les termes "lambda", "délégué" et "callback" peuvent varier selon le contexte d'utilisation.



# LINQ

Utopios® Tous droits réservés

## Introduction à LINQ 1/2

- LINQ (**L**anguage **I**ntegrated **Q**uery) permet d'**interroger** des **collections de données** de manière **déclarative**: comme en SQL, on demande **ce que l'on veut**.
- LINQ peut être utilisé pour interroger des **collections en mémoire**, des **bases de données**, des **documents XML**, et d'autres **sources de données**.
- LINQ offre deux syntaxes : **syntaxe par méthodes** et **syntaxe de requête**.

## Introduction à LINQ 2/2

- **LINQ to Objects** : Pour les **collections** en mémoire, ce que nous allons voir dans cette partie.
- **LINQ to SQL** : Pour interroger les **bases de données SQL**.
- **LINQ to XML** : Pour interroger les **documents XML**.

## LINQ to Objects

- Utilisé pour **interroger et manipuler des collections d'objets en mémoire.**
- Opère principalement sur des types `IEnumerable<T>`.

# IEnumerable et Enumerable

- **IEnumerable** :
  - **Interface** principale pour LINQ to Objects
  - **Type** retourné par la majorité des méthodes Linq
  - **Implémentée** par les **Collections**
- **Enumerable** : **Classe** rassemblant des **Méthodes** pratiques et des **Méthodes d'extension statiques** s'appliquant aux **IEnumerable**.

## Optimisation des requêtes

- LINQ to Objects utilise la technique du **lazy loading/chargement paresseux**.
- Les **opérations** ne sont **exécutées** que **lorsque les résultats sont réellement nécessaires/demandés**, on parle d'**Execution Différée**. Exemple : iteration sur les résultats, affichage, conversion en `List`.
- En pratique, lorsque l'on quitte le type `IEnumerable` avec une **méthode Linq**, les **calculs** seront le plus souvent **effectués**.

# Enumerable.Range

- Avec `Enumerable.Range`, il est possible de créer des intervalles rapidement.

```
Enumerable.Range(<Début>, <nb elements>)
```

- Nous pourrons ensuite effectuer d'autres transformations Linq sur l'`IEnumerable` résultat.

```
List<int> mesNombres = new int[] { 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 };
mesNombresAuCarre.ForEach(Console.WriteLine);
// équivalent Linq
List<int> mesNombres = Enumerable.Range(2, 10).ToList();
mesNombresAuCarre.ForEach(Console.WriteLine);
```

## Select(Func<TSource,TResult>)

- `Select` permet de **transformer** un `Enumerable` en un **autre** en **appliquant une fonction/un délégué sur chaque élément**

```
List<int> mesNombres = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
List<int> mesNombresAuCarre = mesNombres.Select(n => n * n)
                                         .ToList();
mesNombresAuCarre.ForEach(Console.WriteLine);
```

- Avec des méthodes telle que `.ToList()`, `.ToArray()`, `.ToHashSet()` et `.ToDictionary(Func<TSource, TKey>)`, on retrouve des collections classiques

## Where(Func<TSource,bool>)

- `Where` permet de **transformer** un `Enumerable` en un **autre** en **filtrant chaque élément** avec une **fonction/un délégué qui retourne un booléen**, on parle de **prédictat**

```
List<int> nombresPairs2 = mesNombres.Where(x => x % 2 == 0).ToList();
nombresPairs2.ForEach(Console.WriteLine);
// équivalent C# classique
List<int> nombresPairs = new();
foreach(int n in mesNombres)
{
    if (n % 2 == 0) nombresPairs.Add(n);
}
nombresPairs.ForEach(Console.WriteLine);
```

# Liste et Classe à utiliser ensuite

```
public class Personne
{
    public string Prenom { get; init; }
    public string Nom { get; init; }
    public string Email { get; init; }
    public string Phone { get; init; }
    public int Age { get; init; }
}
```

```
// creation d'une liste de personnes
List<Personne> mesPersonnes = new()
{
    new() { Nom = "DUPONT", Prenom = "John", Age = 47, Email = "j.dupont@gmail.com", Phone = "+33 147 741 256" },
    new() { Nom = "SCHMIDT", Prenom = "Martha", Age = 29, Email = "m.schmit@hotmail.com", Phone = "+33 159 236 478" },
    new() { Nom = "DUPONT", Prenom = "Chloé", Age = 16, Email = "c.dupont@gmail.com", Phone = "+33 125 896 478" },
    new() { Nom = "MALTEZ", Prenom = "Clark", Age = 47, Email = "c.martez@aol.com", Phone = "+32 147 852 369" }
};
```

## Find(Func<TSource,bool>)

- Pour **trouver un élément** dans une collection, on peut utiliser la fonction **Find**

```
Personne? chloe = mesPersonnes.Find(x => x.Prenom == "Chloé");
```

```
Personne chloeParDefaut = mesPersonnes.Find(x => x.Prenom == "Chloé")
?? new() { Nom = "DÉFAUT", Prenom = "Chloé" };
```

- Mais avec Linq, on peut **optimiser** la chose via **trois méthodes** possédant chacune une variante retournant la valeur par défaut en cas de condition non remplies ou de non trouvaille

# Équivalents à Find

```
// On peut chercher dans le listing à partir du début
Personne chloeAvecLinq = mesPersonnes.First(x => x.Prenom == "Chloé");
Personne? chloeAvecLinqNullable = mesPersonnes.FirstOrDefault(x => x.Prenom == "Chloé");

// A partir de la fin
Personne chloeAPartirDeLaFin = mesPersonnes.Last(x => x.Prenom == "Chloé");
Personne? chloeAPartirDeLaFinNullable = mesPersonnes.LastOrDefault(x => x.Prenom == "Chloé");

// A partir du début et s'assurer des non doublons de notre critère
Personne laSeuleEtUniqueChloe = mesPersonnes.Single(x => x.Prenom == "Chloé");
Personne? laSeuleEtUniqueChloeNullable = mesPersonnes.SingleOrDefault(x => x.Prenom == "Chloé");
```

- Les versions `...OrDefault` renvoient `null` si elles ne **trouvent rien** alors que les autres lèvent une exception

## Premier et Dernier

Via les méthodes `.First()` et `.Last()` ainsi que leurs versions `...OrDefault`, on peut, sans critère de sélection, avoir la **première** et **dernière** valeur de l'Enumerable

```
var premierePersonneDeLaListe = mesPersonnes.FirstOrDefault();  
var dernierePersonneDeLaListe = mesPersonnes.LastOrDefault();
```

# Trier

- Pour obtenir la série de données **trié selon un critère**, on peut utiliser  
`.OrderBy(Func<TSource, TKey>)` et  
`.OrderByDescending(Func<TSource, TKey>)`

```
var personnesTriesParAgeCroissant = mesPersonnes.OrderBy(x => x.Age)
                                                .ToHashSet();
var personnesTriesParAgeDecroissant= mesPersonnes.OrderByDescending(x => x.Age)
                                                .ToHashSet();
var personnesTrieesParNomPuisPrenom = mesPersonnes.OrderBy(x => x.Nom)
                                                .ThenBy(x => x.Prenom)
                                                .ToHashSet();
```

## Skip et Take

- Pour ne prendre que `n` valeurs, on peut utiliser `.Take(n)`
- Retirer les `n` premières valeurs, on peut utiliser `.Skip(n)`

```
var personneLaPlusJeune = mesPersonnes.OrderBy(x => x.Age).Take(1);  
int noPage = 0;  
var personnesALaPageX = mesPersonnes.Skip(50 * (noPage - 1)).Take(50).ToHashSet();
```

- Dans le cadre d'un **paging**, on a souvent recourt à l'utilisation de `.Take()` précédé de `.Skip()` en se basant sur une **taille** de page et un **numéro** de page.
- Ceci dans le but d'**envoyer à l'utilisateur toutes les valeurs d'un coup**

# Fonctions statistiques

- Bien entendu, les **fonction statistiques** classiques sont aussi présente dans Linq

```
var ageTotalDesPersonnes = mesPersonnes.Sum(x => x.Age);  
var personneLaPlusVieille = mesPersonnes.Max(x => x.Age);  
var personneLaPlusJeuneBis = mesPersonnes.Min(x => x.Age);  
var moyenneDesAges = mesPersonnes.Average(x => x.Age);
```

# Agrégation

- Il est aussi possible de faire de l'agrégation avec

```
.Aggregate(TAccumulate seed, Func<TAccumulate, TSource, TAccumulate> func)
```

Exemple:

```
String chaineDesInitiales = mesPersonnes.Aggregate("Liste des initiales : ", (concat, element) =>
{
    if (mesPersonnes.Last() != element) return concat + element.Nom.Substring(0, 1).ToUpper() + ", ";
    return concat + element.Nom.Substring(0, 1).ToUpper();
});

Console.WriteLine(chaineDesInitiales);
```

- Le **GroupBy** existe lui aussi pour des utilisations plus avancées.

## Pourquoi utiliser **IEnumerable** plutôt que **List** ?

- Lorsque l'on fait des **méthodes qui utilisent Linq et renvoient des collections**, on pourrait être tenté de renvoyer directement des **List**, **Dictionary**, ...
- Grace au principe de **lazy loading**, il est **préférable** de renvoyer plutôt des **IEnumerable**
- C'est une **interface plus flexible** et qui permet de créer des **requêtes différées**, ce qui peut **améliorer les performances**.
- Utiliser **IEnumerable** permet de garder le code **plus général** et **plus réutilisable**.

# IQueryable<>

- `IQueryable` est une **interface** dans .NET pour construire des **requêtes de données dynamiques**, elle implémente `IEnumerable`.
- Elle est utilisée principalement avec **LINQ** pour interagir avec des **sources de données** comme des **bases de données**.
- Permet la composition de **requêtes** avec des **conditions** et des opérations de **tri**, de **filtrage** et de **pagination**.
- Les requêtes `IQueryable` sont exécutées de manière **différée**, optimisées par les **accesseurs de données** comme l'**ORM Entity Framework Core**.



**Merci pour votre attention**

**Des questions ?**