Programmation orientée objet en C#

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Programmation orientée objet en Langage C#

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

01 Définition de la POO

Qu’est-ce que la Programmation Orientée Objet ?

- La **POO** est 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).
- 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 le factorisation et évite ainsi un bon nombre de lignes de code

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 matériaux 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

- **La définition de l’objet** (ou structure de l’objet)
  - 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…*

Les paradigmes de la Programmation Orientée Objet

La POO repose sur plusieurs concepts importants

- L'**Accessibilité** (ou **Visibilité**)
- L'**Encapsulation**
- L'**Héritage**
- Le **Polymorphisme**
- Les **Interfaces** et l'**Abstraction**
- La **Généricité**

Nous les aborderons tous par la suite.

Le concept de Accessibilité (ou Visibilité)

L'accessibilité repose sur le fait de **pouvoir donner l'accès** en **lecture** et/ou en **écriture** à **un élément d'un objet** à l'**extérieur** de celui-ci ou la garder accessible **seulement depuis la classe elle-même**.

Pour cela on pourra qualifier cet élément par exemple de :

- **privé** si il est accessible/visible **uniquement** via des **instructions à l'intérieur de la classe**
- **public** si il est accessible/visible via **toutes instructions**, interne ou non à la classe
- ...

Le concept de l'Encapsulation

- L'encapsulation **protège les attributs** (données) **de l'objet**
  - Par la mise en place de « **getter** » et « **setter** » public permettant d'accéder à ses **attributs privés**

- L'encapsulation **protège l'objet dans son ensemble**
  - On ne peut l'instancier que par son « **constructeur** »
  - On ne peut le manipuler qu'avec ses « **méthodes** » et « **propriétés** » (éléments exécutant des suites d'instructions)

Le concept d'Interface

Une **Interface** est un **contrat** que s'engage à **respecter** une classe. Il s'agit d'un ensemble de **méthodes publiques** que devra posséder toute classe qui remplit ce contrat (qui **implémente** l'interface)

- Une **interface** contient des **signatures de méthodes**, elle **ne contient pas forcément des instructions** (corps) pour ces méthodes
- Lorsque les méthodes d'une interface n'ont pas de **corps** il sera donc **obligatoire** de définir les **instructions des méthodes** de la classe qui **implémente** l'interface

Résumé de la Programmation Orientée Objet

L'approche orientée objet permet de **modéliser** une application sous la forme d'interactions entre **objets**

Les objets ont des **attributs** et peuvent faire des **actions** par le biais de **méthodes**

Elle masque la complexité d'une implémentation grâce à l'**encapsulation**

Les objets peuvent **hériter** de fonctionnalités d'autres **objets** et **implémenter** des **interfaces**

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

02 Définition des 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**

- 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**.

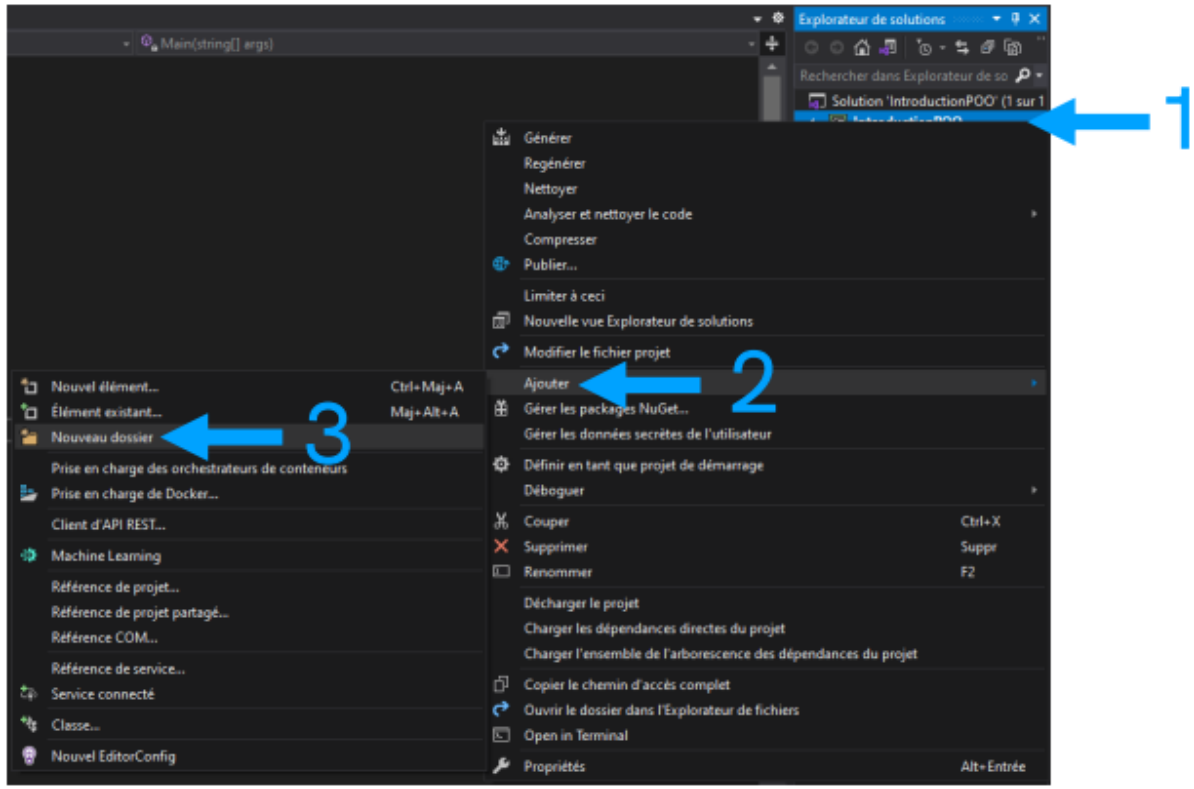
Qu ’est-ce qu’une Classe ?

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

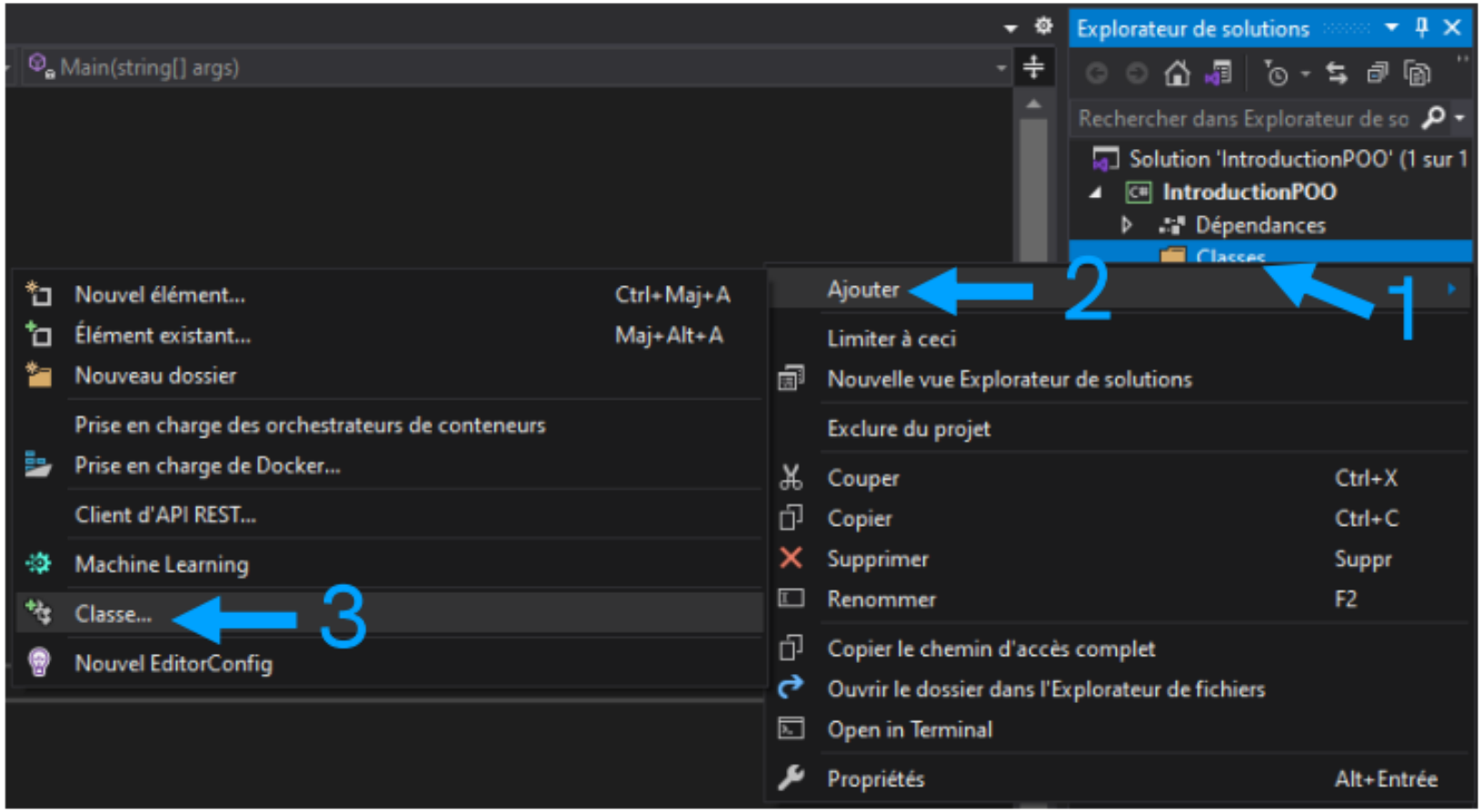
- Créer un nouveau projet console
- Par défaut, l'onglet «Program.cs» est ouvert
- Dans explorateur de solution créez un dossier nommé «Classes» en faisant un clic droit sur le nom de votre projet



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

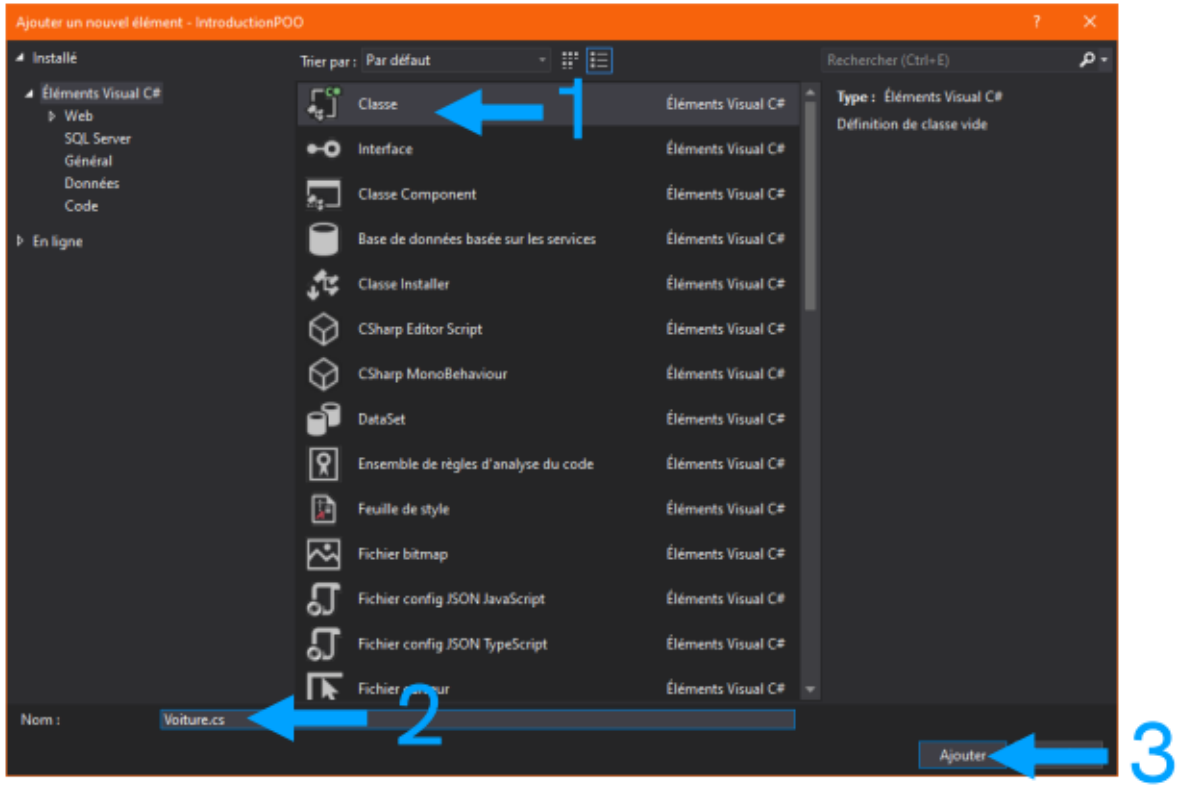
Création d'une nouvelle Classe

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**



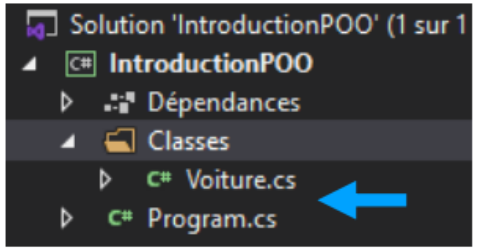
Ces templates peuvent être d'autres types de fichiers (cshtml, razor, ...)

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 (classes, attributs, méthodes, propriétés, ...).

Il en existe 5 types ne s'appliquant pas à tout les éléments:

Visibilité	Description
public	Accès non restreint
private	Accès uniquement depuis la même classe
protected	Accès depuis la même classe ou depuis une classe dérivée (cf héritage)
internal	Accès restreint à la même assembly (de base pour les classes)

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

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 à 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 **dépends 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 définit**

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 définit

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

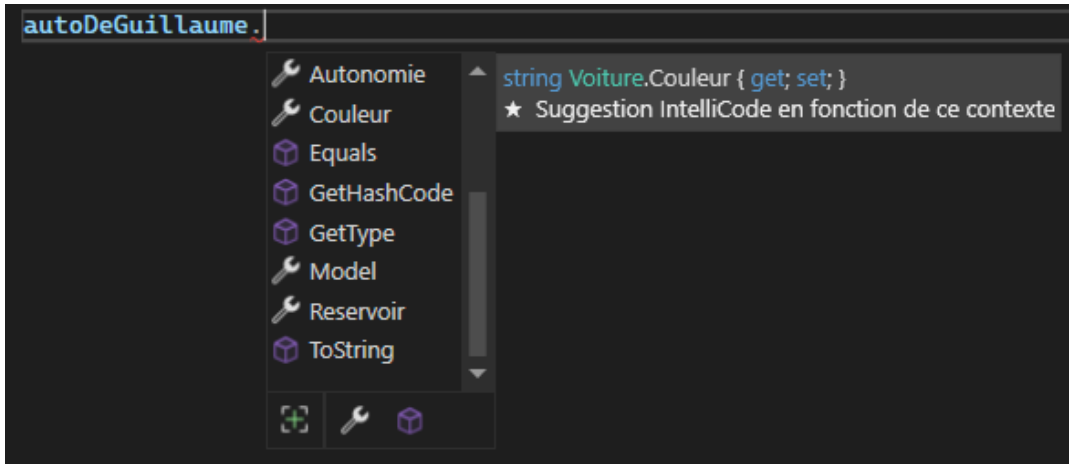
Ici, les attributs auront les valeurs définies à l'appel du constructeur.

Cependant si vous avez définit 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.Cou1eur = "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 (!Demaree)
    {
        Demaree = true;
        Console.WriteLine( "La voiture est démarée... le moteur tourne !");
    }
    else
        Console.WriteLine("La voiture est déja 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)
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

03 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** portants le même nom mais avec d**es 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**) 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 polymorphisme **avec signatures différentes**
  - par **Surcharge / Overload** (aussi nommé « **ad hoc** »)
  - **Paramétrique**
- Les polymorphisme 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 polymorphisme 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**

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

04 Définition de l’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 `Mammifere`
- La classe `Mammifere` est une sorte de la classe `Animal`
- La classe `Animal` est une sorte de la classe `ÊtreVivant`

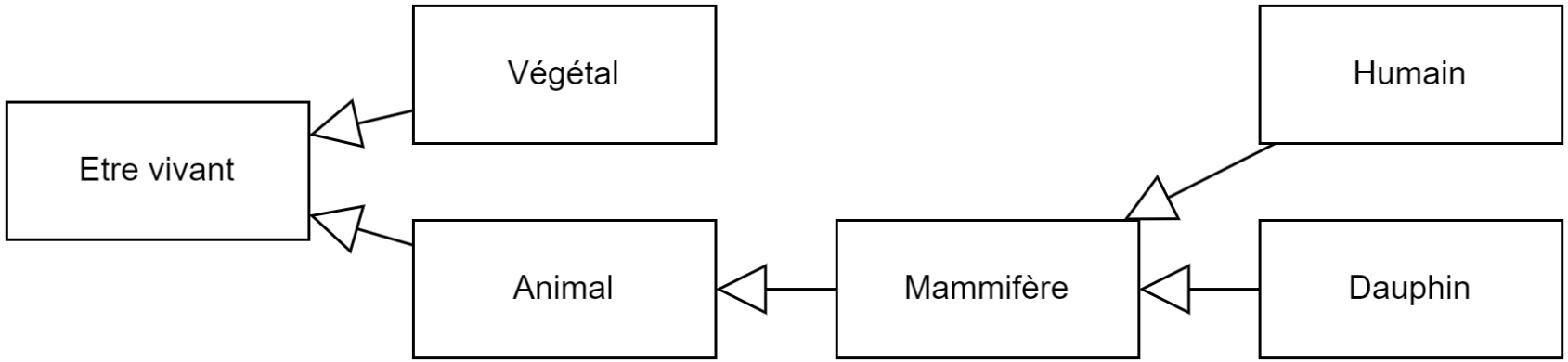
Chaque **parent** est un plus **général** que son **enfant**
Et inversement, chaque **enfant** est un plus **spécialisé** que son **parent**
L'**enfant** aura donc **les caractéristiques du parent** auxquelles s'ajoute ses **spécificités**

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 **acceder 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 polymorphisme 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 mammifere 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`.

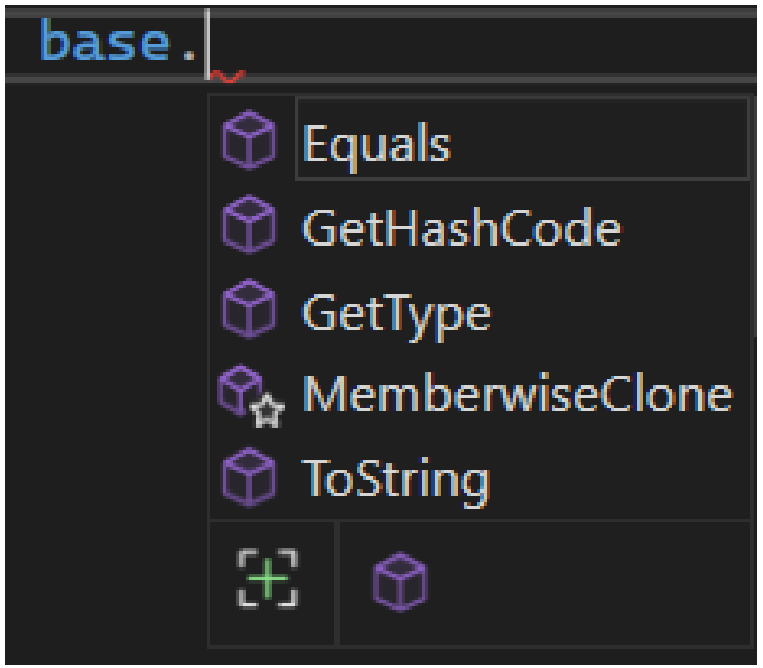
```
internal class Animal
{
    public string Nom { get; set; }
    public bool EstVivant { get; set; }
    public Animal(string nom, bool estVivant)
    {
        Nom = nom;
        EstVivant = estVivant;
    }
    public virtual void Respirer()
        => Console.WriteLine("L'animal respire");
}
```

```
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 mammifere 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}";
}
```

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

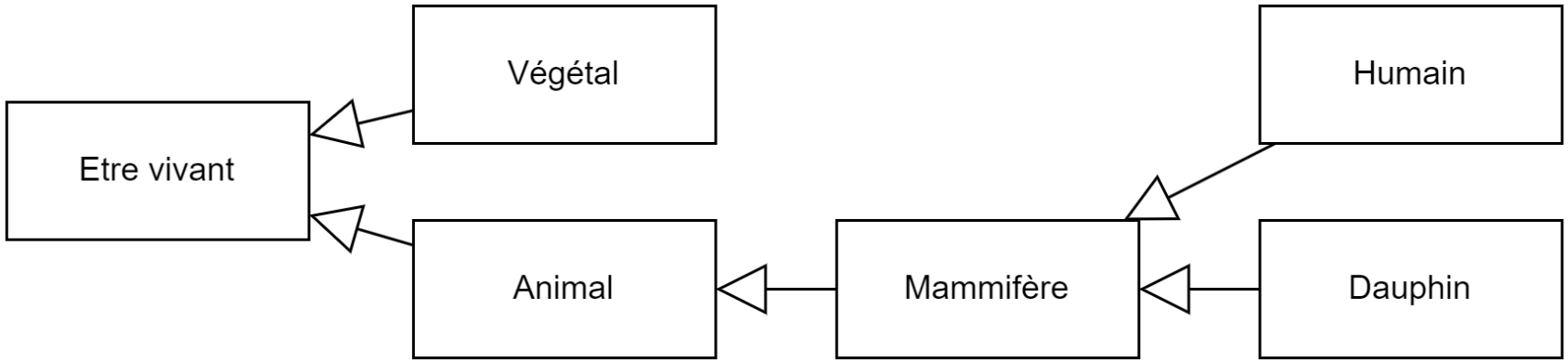
Les classes abstraites (abstract)

- 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 `Espece` 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**

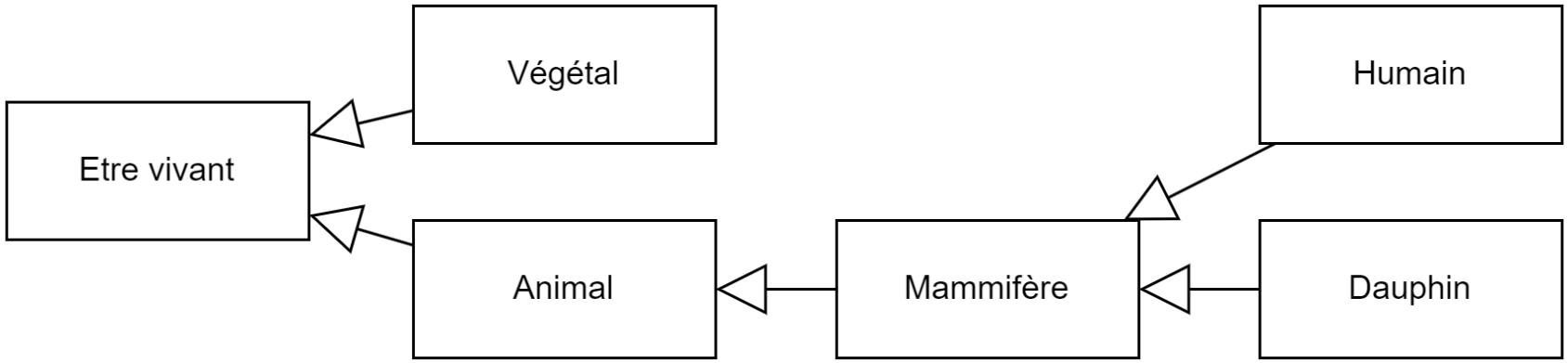
- Elle n'a **pas de corps** (pas de block de code)
- Une méthode `abstract` sera toujours dans une class `abstract`
- Pour être utilisables, les méthodes `abstract` doivent être redéfinies avec un `override`

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.

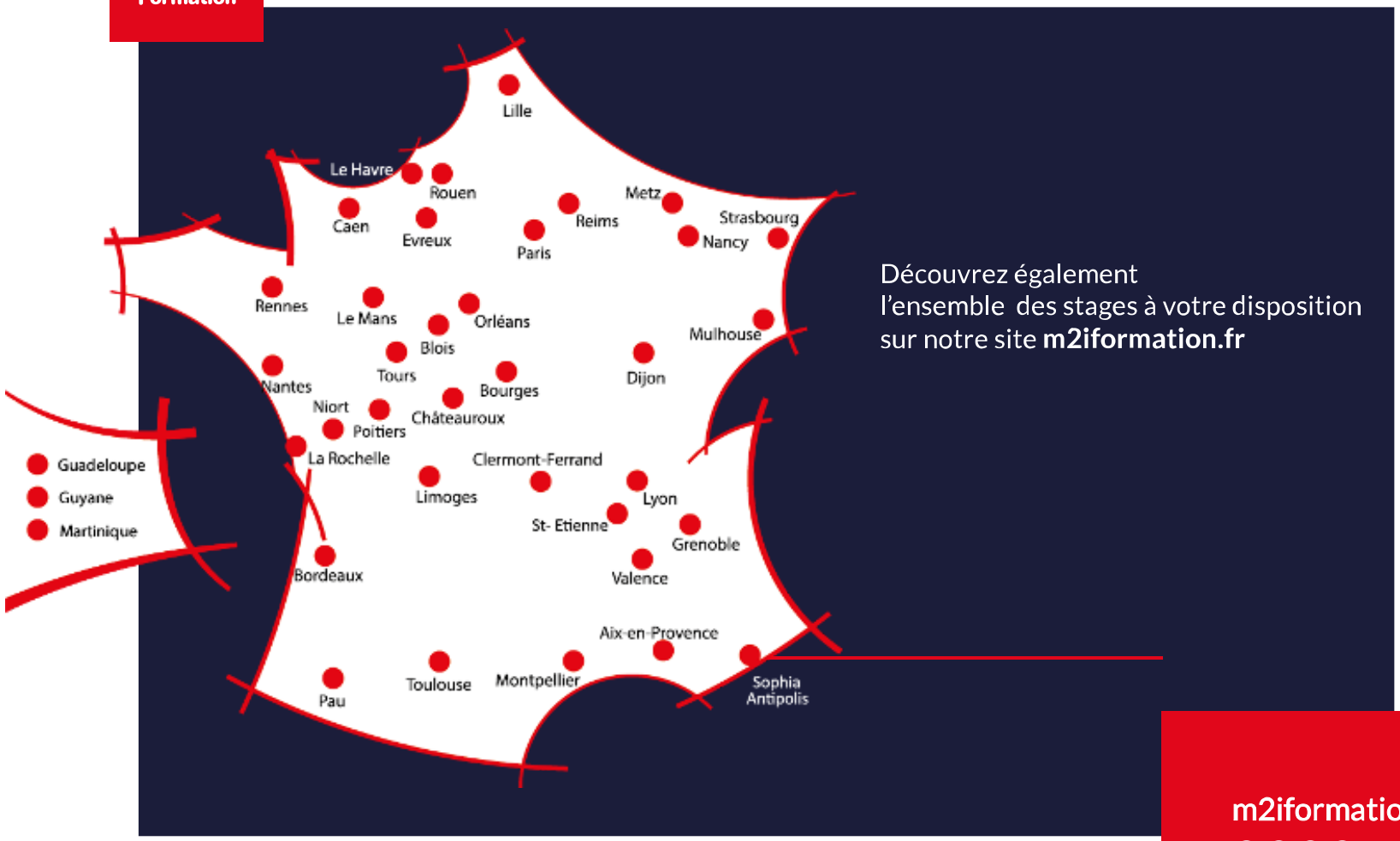
Si nous prenons l'exemple précédent avec la liste, nous pourrons ainsi itérer sur la liste puis **convertir chaque élément si besoin**.

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

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr