

# INTRODUCTION A LA P00

UTOPIOS



ANTHONY DI PERSIO



## DÉFINITION DE LA POO

Découverte des paradigmes de la POO

01

## DÉFINITION DES CLASSES

Les attributs, méthodes et constructeurs d'une classe C#

02

## LE POLYMORPHISME

Définition de la notion de polymorphisme

03

## DÉFINITION DE L'HÉRITAGE

Comprendre les principes de l'héritage

04

## TABLE DES MATIÈRES

05

## LES INTERFACES

Leur définition et leur rôles

06

## LES GÉNÉRIQUES

Piles...? List...? Kézako?

07

## LES DÉLÉGUÉS

Notions et rôles des délégués en C#

08

## LES EXCEPTIONS

La récupération des exceptions



01

# DÉFINITION DE LA POO

Découverte des paradigmes de la POO

# DÉFINITION DE LA POO

Qu'est-ce que la Programmation Orientée Objet ?

- La POO est une façon de développer une application qui consiste à représenter («modéliser») une application informatique sous la forme d'objets, ayant des propriétés et pouvant interagir entre eux
- 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 jours et l'ajout de nouvelle fonctionnalité en limitant les régressions.
- La POO permet également la réutilisabilité et évite ainsi un bon nombre de lignes de code



# DÉFINITION DE LA POO

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

- Commençons par définir les objets dans le mode réel, une table, une chaise, une voiture... etc.
  - **Ils possèdent des propriétés** : *Une chaise à 4 pieds, une couleur précise, un matériaux précis...etc.*
  - **Certains objets peuvent faire des actions** : *la voiture peut rouler, klaxonner...etc.*
  - **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...etc.*
- Le concept d'objet en programmation s'appuie sur ce fonctionnement. Un objet à des caractéristiques qui le définissent (propriétés, fonctionnement, interaction...)

# DÉFINITION DE LA POO

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...etc.
  - **L'instance d'un objet**
    - ✓ C'est la création réelle de l'objet : *Objet Chaise*
    - ✓ En fonction de sa définition : *4 pieds, bleu...etc.*
    - ✓ Il peut y avoir **plusieurs instances** : *Plusieurs chaises, de couleurs différentes, matériaux différents...etc.*

# DÉFINITION DE LA POO

Les paradigmes de la Programmation Orientée Objet

- La POO repose sur plusieurs concepts importants
  - L'Encapsulation
  - L'Héritage
  - Le Polymorphisme et la Substitution
  - Les Interfaces

# DÉFINITION DE LA POO

## Le concept de l'encapsulation

- Cela permet de protéger l'information contenue dans notre objet et de le rendre manipulable uniquement par ses actions ou propriétés
  - L'encapsulation protège les attributs de l'objet
    - ✓ Par la mise en place de « **getter** » et « **setter** » *public* permettant d'accéder à ses attributs *privées*
  - 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** »



# DÉFINITION DE LA POO

## Le concept d'Interface

- Une **Interface** est un **contrat** que s'engage à **respecter** un **objet**.
- Il s'agit donc d'un ensemble de **méthodes** accessibles depuis l'**extérieur** d'une **classe**, par lesquelles on peut **modifier** un **objet**, le « **spécialiser** » ou plus généralement **communiquer** avec lui
  - Une **interface** est une **signature** de **méthode**, elle ne contient pas d'**instructions**
  - Elle n'a pas de **corps** et il est **obligatoire** de **définir** les instructions de la **l'interface** (le corps de l'interface) dans la **classe** qui **utilise** cette **interface**

# DÉFINITION DE LA POO

Résumé de la Programmation Orientée Objet

- L'approche orientée objet permet de modéliser son 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 s'il y a une relation d'héritage entre eux



02

# DÉFINITION DES CLASSES

Les attributs, méthodes et constructeurs d'une classe C#

# DÉFINITION DES CLASSES

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

- Un **Classe** (Class) permet de regrouper tous les éléments représentant un Objet : ses propriétés, sa structure, ses méthodes
  - Nous avons déjà pu voir une **Classe** dans le code que nous avons utilisé précédemment et qui a été généré par Visual Studio, la classe Program.

```
class Program
{
    0 références
    static void Main(string[] args)
    {
    }
}
```



# DÉFINITION DES CLASSES

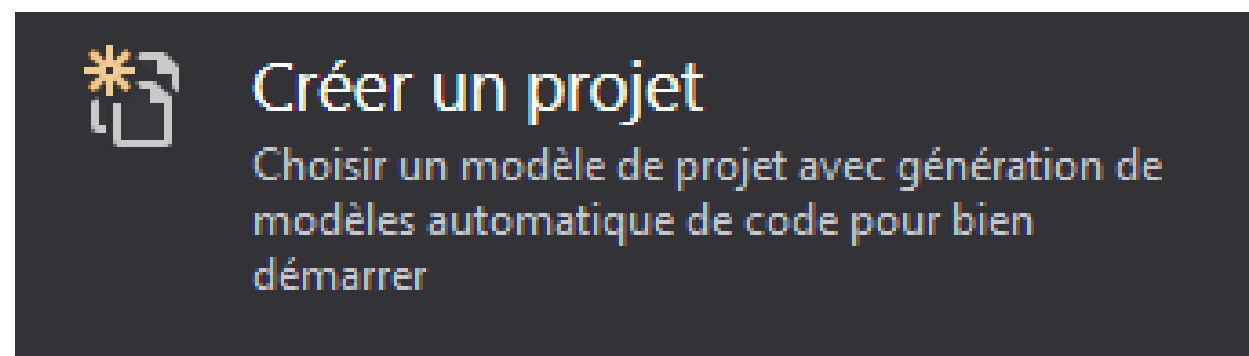
Qu 'est-ce qu'une **Classe** ?

- La `class Program` est une classe particulière car elle contient la méthode « `Main()` » qui est le point d'entrée de notre application
  - Mais elle fonctionne comme toutes les classes, elle définit l'objet `Program` qui peut faire des actions, ici c'est la méthode `Main()` les définies
  - Notez la présence des accolades qui délimitent la classe ( le bloc de code de celle-ci ) et par ce biais la portée des variables contenues dans celle-ci
  - Les noms des classes comme des méthodes s'écrivent en **PascalCase**. *Exemple: MaNouvelleClasse*

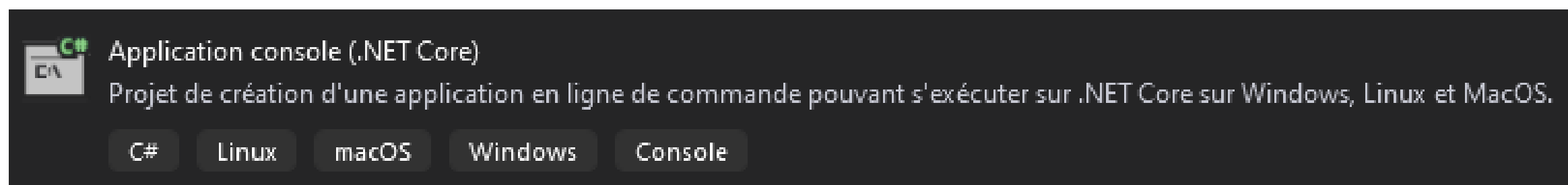
# DÉFINITION DES CLASSES

Création d'un nouveau projet Console

- Ouvrez Visual Studio et sur la page d'accueil cliquez sur Créer un projet



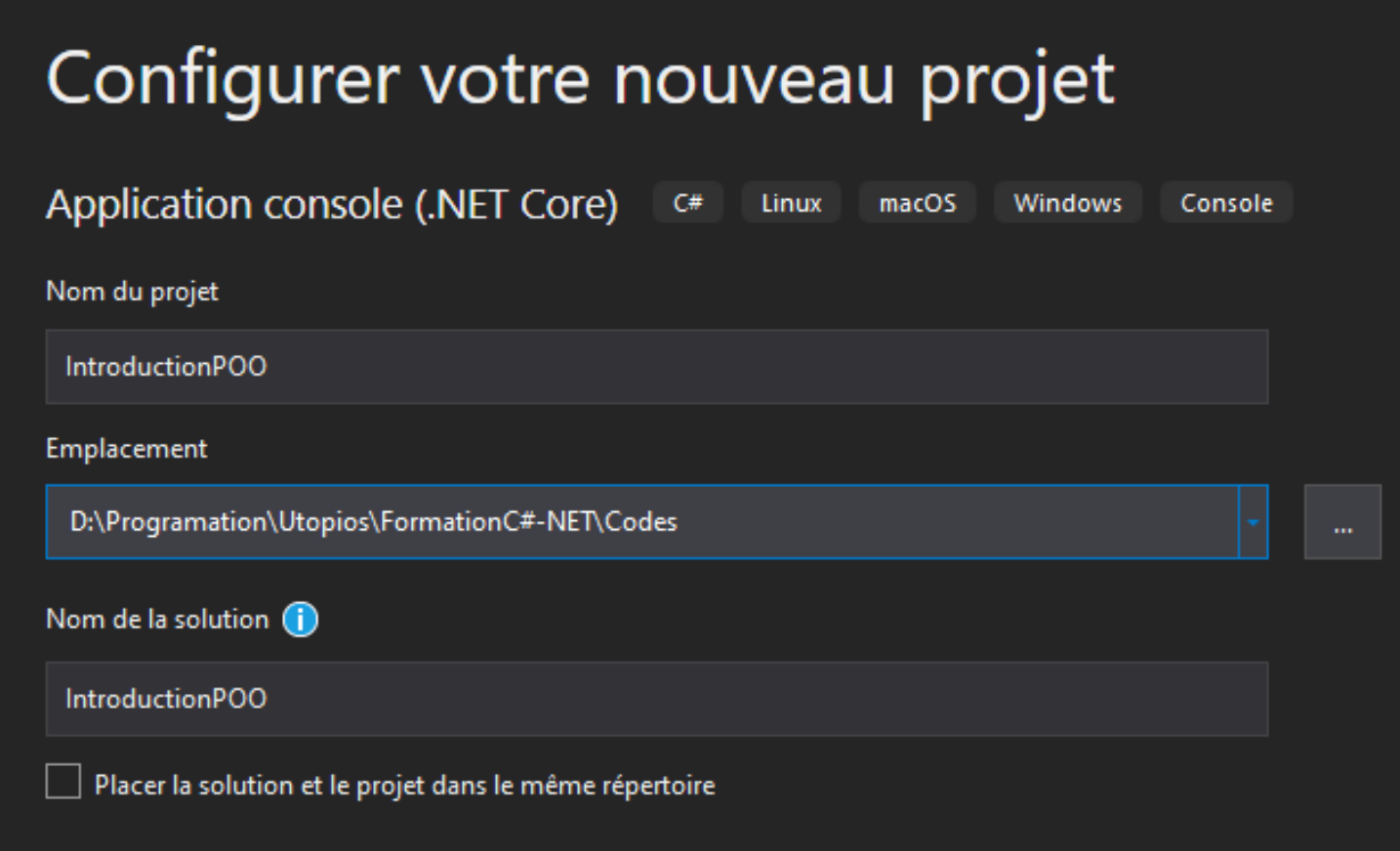
➤ Sélectionnez Application console (.NET Core)





# DÉFINITION DES CLASSES

Création d'un nouveau projet Console



The screenshot shows the 'Configurer votre nouveau projet' (Configure your new project) dialog in Visual Studio. The title is 'Configurer votre nouveau projet'. Below it, there are tabs for 'Application console (.NET Core)', 'C#', 'Linux', 'macOS', 'Windows', and 'Console'. The 'Application console (.NET Core)' tab is selected. Under 'Nom du projet' (Project name), the text 'IntroductionPOO' is entered. Under 'Emplacement' (Location), the path 'D:\Programation\Utopios\FormationC#-NET\Codes' is entered. Under 'Nom de la solution' (Solution name), the text 'IntroductionPOO' is entered. At the bottom, there is a checkbox labeled 'Placer la solution et le projet dans le même répertoire' (Place the solution and the project in the same directory), which is currently unchecked.

- Saisissez le nom de votre projet: *Ici « IntroductionPOO »*
- Choisissez le chemin de votre projet
- Saisissez le nom de la Solution: *Ici identique au projet*

# DÉFINITION DES CLASSES

Création d'un nouveau projet Console

```
1      using System;
2
3      namespace IntroductionP00
4      {
5          0 références
6          class Program
7          {
8              0 références
9              static void Main(string[] args)
10             {
11                 Console.WriteLine("Hello World!");
12             }
13         }
14     }
```

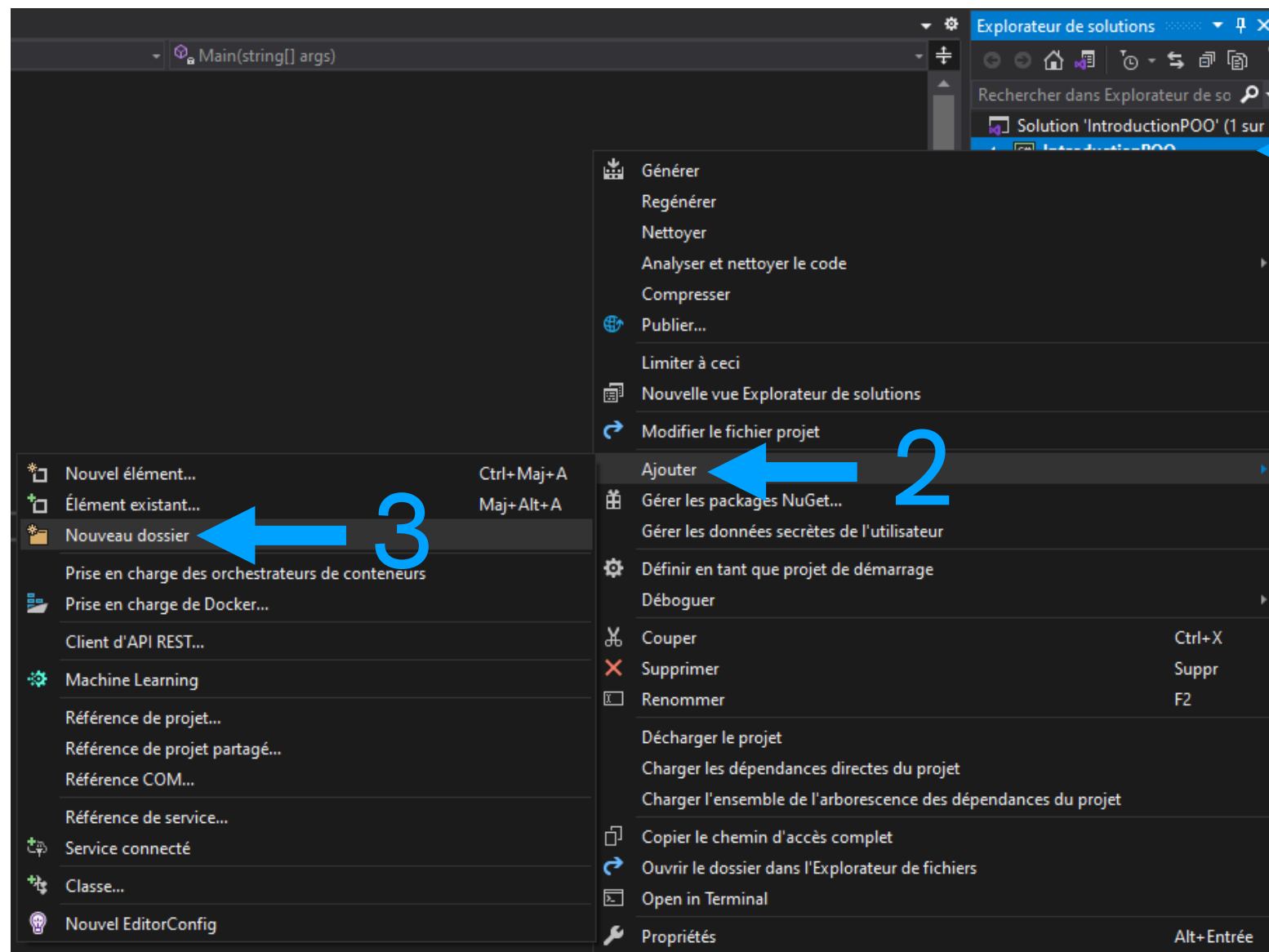
- Par défaut, l'onglet « **Program.cs** » est ouvert avec ce bout de code que vous pouvez exécuter



# DÉFINITION DES CLASSES

## Création d'une nouvelle Classe

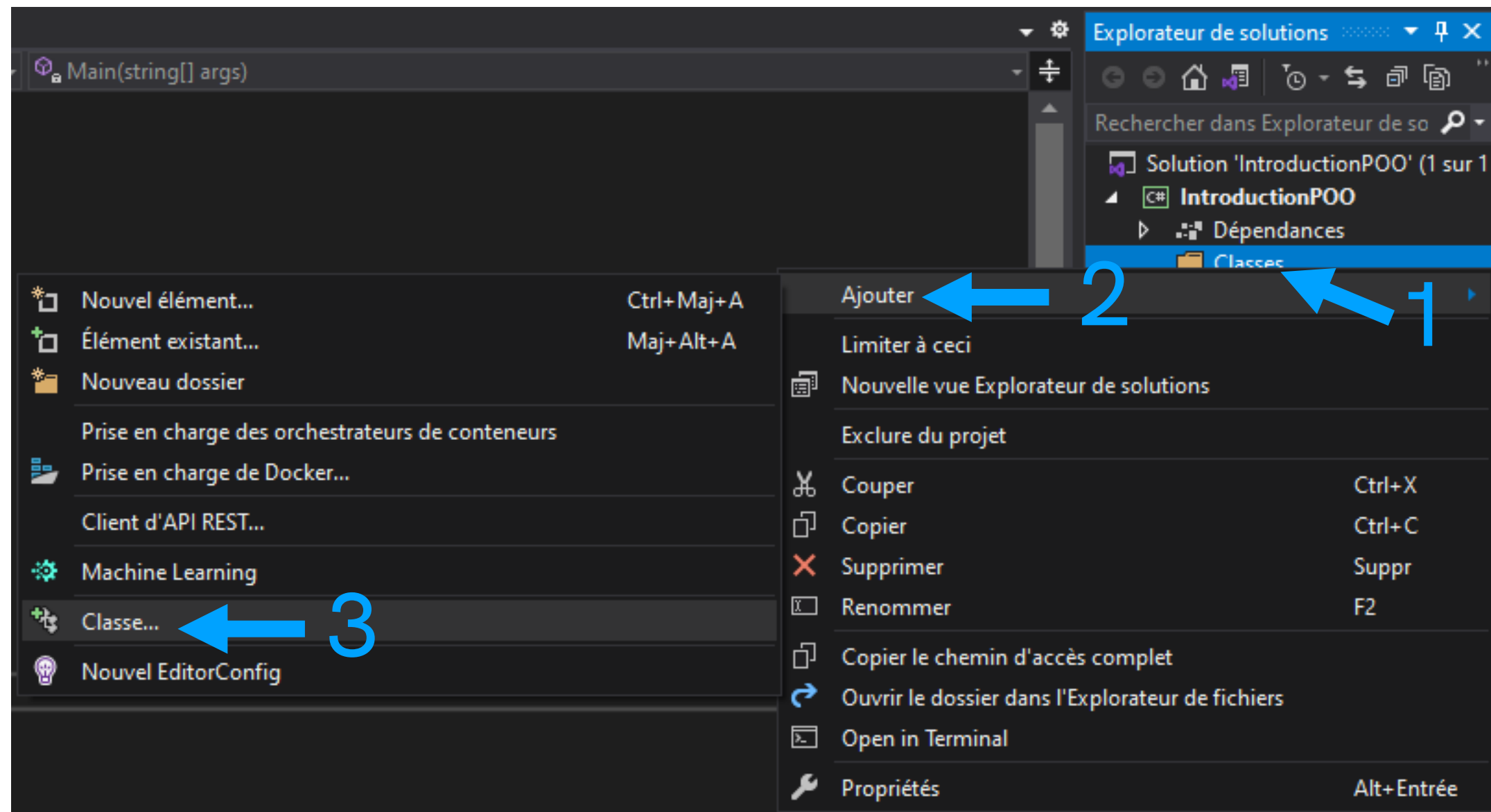
- Dans explorateur de solution créez un dossier nommé « Classes » en faisant un clic droit sur le nom de votre projet



# DÉFINITION DES CLASSES

## Création d'une nouvelle Classe

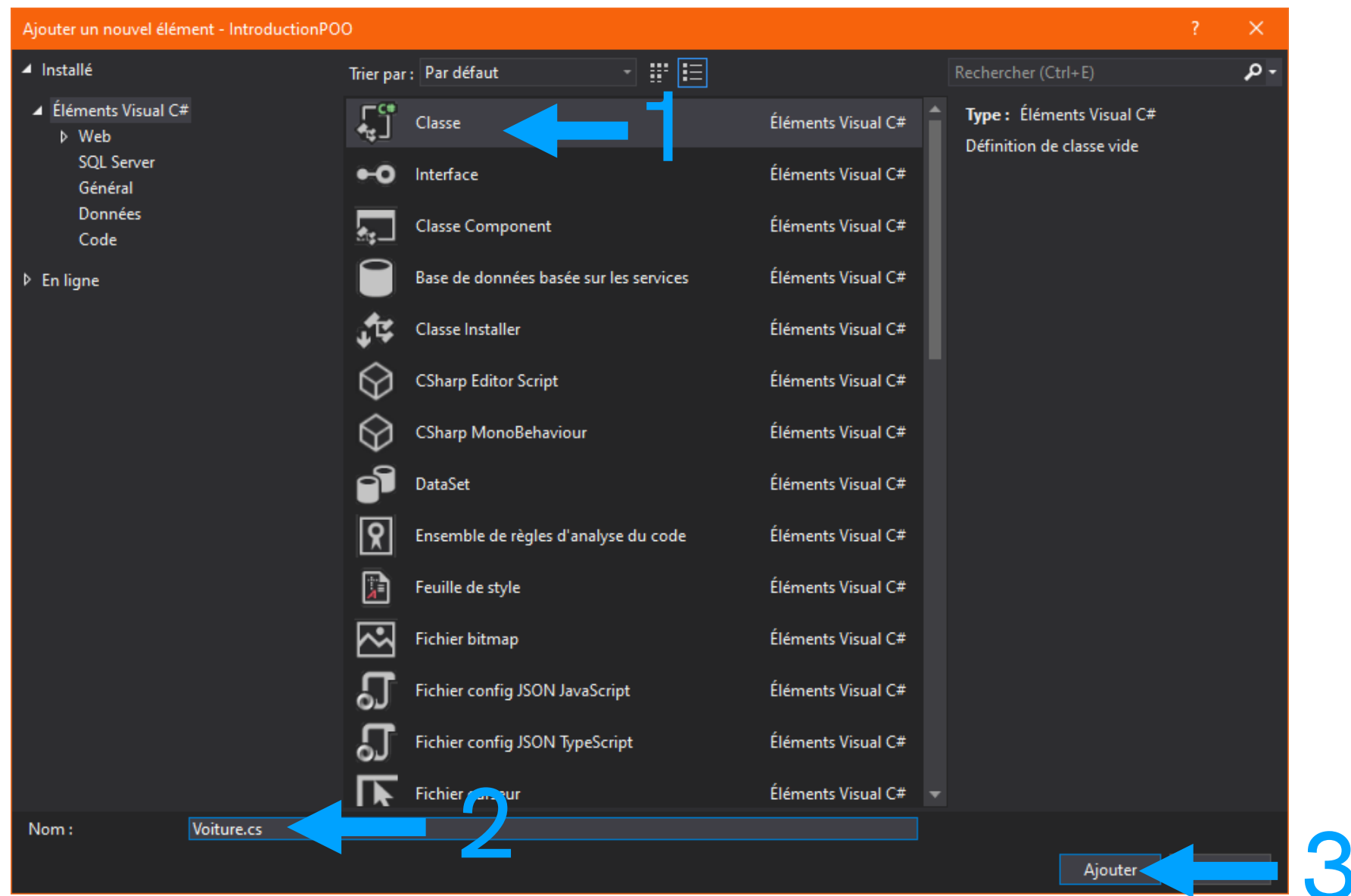
- Faire Clic droit sur votre dossier « Classes » puis « Ajouter » puis « Classe »



# DÉFINITION DES CLASSES

## Création d'une nouvelle Classe

- Enfin nommez cette nouvelle classe. Pour notre exemple nous l'appelons « Voiture.cs »





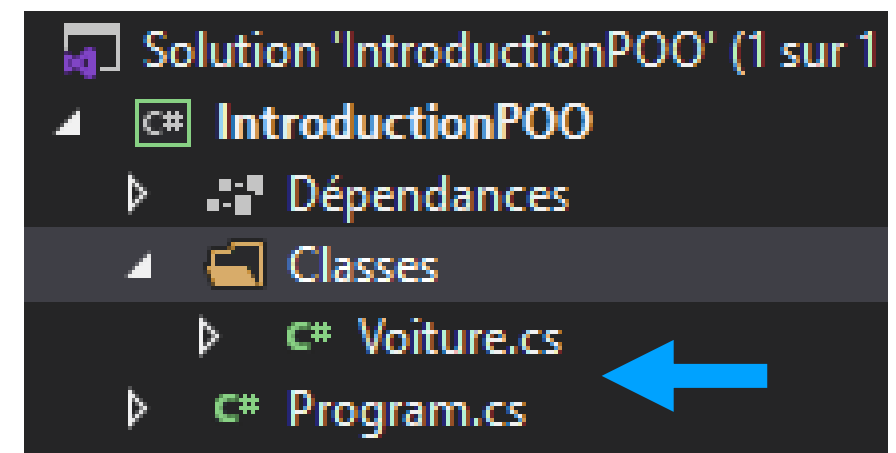
# DÉFINITION DES CLASSES

## Création d'une nouvelle Classe

- Visual Studio Ouvre cette nouvelle classe, elle apparaît dans l'arborescence de votre application

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

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



- Maintenant, nous allons pouvoir commencer à développer notre première classe, la `class Voiture` ...

# DÉFINITION DES CLASSES

Les attributs d'une classe

- Les attributs d'une classe correspondent à ses caractéristiques
  - Les attributs sont l'ensemble des variables permettant de définir les caractéristiques de notre objet
  - Ils doivent être déclarés au début de notre classe
  - Tous les types de variables sont utilisable pour la déclaration des attributs ( *int, float, string...etc.*) y compris des objets
  - Ils se déclarent comme suit et peuvent être initialisé ou non en fonction des besoins de votre application

```
string model;
```

# DÉFINITION DES CLASSES

Les attributs d'une classe

- Les **attributs** et leurs **propriétés**
  - Le principe de l'**encapsulation** de la **POO** a pour bonne pratique de laisser les **attributs** d'un objet (d'une classe) en **privé** (**private**), c'est-à-dire uniquement accessible depuis l'intérieur de cette classe, c'est sa **visibilité**
  - Nous allons générer des **propriétés publique** (**public**) qui elles seront **accessible** pour le **reste** de notre **application** avec l'emploi de « **getter** » et « **setter** »
  - Voici la **syntaxe** pour la **propriété** d'un **attribut** en **C#**

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



# DÉFINITION DES CLASSES

Les attributs d'une classe

- Voici notre class Voiture après la déclaration des **attributs** et de leurs **propriétés**

```
class Voiture
{
    private string model;
    private string couleur;
    private int reservoir;
    private int autonomie;

    0 références
    public string Model { get => model; set => model = value; }
    0 références
    public string Couleur { get => couleur; set => couleur = value; }
    0 références
    public int Reservoir { get => reservoir; set => reservoir = value; }
    0 références
    public int Autonomie { get => autonomie; set => autonomie = value; }
}
```

# DÉFINITION DES CLASSES

Le constructeur d'un objet

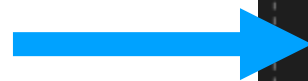
- Maintenant que notre objet **Voiture** a des attributs et des propriétés, il nous faut un outil pour pouvoir « **construire** » notre objet.
  - Cet outils s'appel le naturellement le **constructeur**, il définit les caractéristiques nécessaires et la manière de construire notre objet
  - Voici la **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)
{
    this.model = Model;
    this.couleur = Couleur;
    this.reservoir = Reservoir;
    this.autonomie = Autonomie;
}
```

# DÉFINITION DES CLASSES

Vue d'ensemble de notre `class Voiture` à présent

- Notez la présence d'un constructeur vide, nous y reviendrons



```
class Voiture
{
    private string model;
    private string couleur;
    private int reservoir;
    private int autonomie;

    0 références
    public Voiture()
    {
    }

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

    0 références
    public string Model { get => model; set => model = value; }
    0 références
    public string Couleur { get => couleur; set => couleur = value; }
    0 références
    public int Reservoir { get => reservoir; set => reservoir = value; }
    0 références
    public int Autonomie { get => autonomie; set => autonomie = value; }
}
```



# DÉFINITION DES CLASSES

L'instanciation d'un objet

- Maintenant que notre objet **Voiture** a des attributs, des propriétés et un constructeur, nous allons pouvoir le créer réellement depuis notre class **Program**

➤ Voici la **syntaxe** pour l'instanciation d'un objet en C#

```
Voiture VoitureDeNicolas = new Voiture();
```

- Comme vous le constatons, la **class Voiture** n'est pas reconnu par VS. En effet il nous manque une choses:
  - ✓ L'import de notre dossier Classes à notre application

# DÉFINITION DES CLASSES

L'instanciation d'un objet

- Corrigeons cette erreurs
  - Faisons l'import de notre dossier Classes dans le fichier de la `class Program` afin que notre programme puis trouver la `class Voiture`

```
using IntroductionP00.Classes;
```

- Maintenant nous pouvons instancier notre premier objet

```
Voiture VoitureDeNicolas = new Voiture();
```

# DÉFINITION DES CLASSES

La notion de visibilité

- L'indicateur de visibilité sert à indiquer qui peut accéder à l'élément qui le suit
  - Il existe 5 indicateurs de visibilité qui définissent l'accès aux attributs, méthodes, classes...etc.

Visibilité	Description
public	Accès non restreint
protected	Accès depuis la même classe ou depuis une classe dérivée
private	Accès uniquement depuis la même classe
internal	Accès restreint à la même assembly
protected internal	Accès restreint à la même assembly ou depuis une classe dérivée

- La visibilité par défaut est « internal » ce qui nous a posé problème au moment d'instancier notre objet **Voiture**



# DÉFINITION DES CLASSES

La modification d'un objet instancié

- Maintenant que nous avons instancié notre objet **Voiture** grâce à l'emploi du mot-clé **new** pour pouvons accéder à ses propriétés via l'auto complétion de l'IDE

```
Voiture VoitureDeNicolas = new Voiture();  
VoitureDeNicolas.  
Autonomie int Voiture.Autonomie { get; set; }  
Couleur  
Model  
Reservoir
```

- Nous pouvons désormais saisir les propriétés de notre objet VoitureDeNicolas

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

# DÉFINITION DES CLASSES

Affichage de notre objet **Voiture** dans la console

- Maintenant que nous avons instancié notre objet **Voiture** complété, nous pouvons l'utiliser à notre gré. Procédons à son affichage dans la console

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

➤ Ce qui nous donne le résultat suivant dans la console

```
Notre première voiture est une Clio de couleur Noir  
Elle à un réservoir de 45 litres pour une autonomie de 900 km.
```

➤ Nous avons instancié et inséré les propriétés en plusieurs étapes afin de bien comprendre le processus. Cette instantiation à utilisé le constructeur vide, sans paramètres

# DÉFINITION DES CLASSES

L'instanciation d'un objet avec paramètres

- Maintenant voyons comment instancier l'objet Voiture avec ses paramètres (utilisation du deuxième constructeur)

```
Voiture VoitureDeJulie = new Voiture("208", "Blanche", 40, 800);
```

- Nous ferons appel à la signature du deuxième constructeur

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

- Ce qui nous donne le résultat suivant dans la console

```
La voiture est une 208 de couleur Blanche
Elle à un réservoir de 40 litres pour une autonomie de 800 km.
```



# DÉFINITION DES CLASSES

Les méthodes d'une classe

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

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

- Nous appellerons cette Méthode **Afficher()** et prendra un objet **Voiture** en paramètre de méthode

```
public static void Afficher(Voiture v)  
{  
    Console.WriteLine("La voiture est une " + v.Model + " de couleur "  
        + v.Couleur);  
    Console.WriteLine("Elle à un réservoir de " + v.Reservoir +  
        " litres pour une autonomie de " + v.Autonomie + " km.");  
}
```

# DÉFINITION DES CLASSES

Les méthodes d'une classe

- Penchons nous maintenant sur les méthodes de notre **class Voiture** car pour le moment elle existe mais ne bouge pas!
  - Faisons ensemble la Méthode **Demarrer()**.
    - ✓ Nous ajoutons un attribut **booléen demaree** et nous utiliserons sa propriété pour indiquer si le moteur tourne.
    - ✓ Nous ferons une structure conditionnelle afin de vérifier si le moteur tourne avant de le démarrer
    - ✓ Si il est éteint nous afficherons un message dans la console pour informer l'utilisateur que le moteur est démarré
    - ✓ Sinon nous indiquerons que le moteur tourne déjà

# DÉFINITION DES CLASSES

Les méthodes d'une classe

- Voici la Méthode **Demarrer()**.

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

- Maintenant c'est à vous
  - ✓ Réalisez le TP **class Voiture** avec les méthodes Arrêter( ), Rouler( ), Stopper( ) Klaxonner(),



03

# LE POLYMORPHISME

Définition de la notion de polymorphisme



# LE POLYMORPHISME EN C#

Les concepts de Polymorphisme et de Substitution

- Le mot **polymorphisme** suggère qu'un objet peut prendre **plusieurs formes**
- Le **polymorphisme** est aussi la capacité pour un objet de faire une même action avec différents types d'intervenants
- Il y a plusieurs types possibles de **polymorphisme** en **POO**
  - Le **polymorphisme** « **ad hoc** » par l'emploi de « **surcharges** »
  - Le **polymorphisme** « **paramétrique** » (signatures différentes)
  - Le **polymorphisme** « **d'Héritage** » c'est la « **Substitution** »

# LE POLYMORPHISME EN C#

Le polymorphisme « ad hoc »

- C'est la possibilité d'utiliser le même nom de méthode par l'emploi d'une surcharge de méthode

2 références

```
public void AjouterProduit(string NomProduit, string DescriptionProduit, int Quantite)
```

- Ici notre méthode prend 3 variables en arguments

2 références

```
public void AjouterProduit(Produit p)
```

- Ici notre méthode prend un objet en argument

# LE POLYMORPHISME EN C#

## Le polymorphisme « paramétrique »

- C'est la possibilité d'utiliser le même nom de méthode avec le même nombre d'arguments mais avec une signature différente

```
0 références
public int Additionner(int a, int b)
{
    return a + b;
}
```

➤ Ici notre méthode est signée int

```
0 références
public string Additionner(string a, string b)
{
    double resultat = Convert.ToDouble(a) + Convert.ToDouble(b);
    return "Le résultat est "+resultat ;
}
```

➤ Ici notre méthode est signée string

# LE POLYMORPHISME EN C#

## Les concepts de Polymorphisme et de Substitution

- La **substitution** est une manifestation du **polymorphisme**
- Il s'agit de la capacité d'un objet « **enfant** » à redéfinir des caractéristiques ou des actions d'un objet « **parent** »
- Prenons par exemple un objet « **mammifère** » qui sait faire l'action (méthode) « **se déplacer** »
  - L'objet **Homme** va « **se déplacer** » sur ses deux jambes
  - L'objet **Dauphin** va « **se déplacer** » grâce à ses nageoires
  - ✓ La **substitution** va donc permettre de redéfinir la méthode « **se déplacer** » pour chaque objet « **enfant** », il se « **spécialise** »



04

# DÉFINITION DE L'HÉRITAGE

Comprendre les principes de l'héritage

# DÉFINITION DE L'HÉRITAGE

## Le concept de l'héritage

- C'est un autre élément important dans la Programmation Orientée Objet
  - Un objet dit « **parent** » peut transmettre certaines de ses caractéristiques à un autre objet dit « **enfant** »
  - Pour cela, on pourra définir une relation d'héritage entre eux
- S'il y a une relation d'héritage entre un objet « **parent** » et un objet « **enfant** » ( Que l' « **enfant** » hérite du « **parent** » )
  - On dit que l'objet « **enfant** » est une « **spécialisation** » de l'objet « **parent** » ou qu'il *dérive* de l'objet « **parent** »



# DÉFINITION DE L'HÉRITAGE

## Le concept de l'héritage

- Afin de comprendre cette notion d'héritage, rien de tel que quelques exemple basés sur le réel
  - L'objet « **chien** » est une sorte d'objet « **mammifère** »
  - L'objet « **mammifère** » est une sorte d'objet « **animal** »
  - L'objet « **animal** » est une sorte d'objet « **être vivant** »
- Chaque « **parent** » est un peu plus « **général** » que son « **enfant** »
  - Et inversement, chaque « **enfant** » est un peu plus « **spécialisé** » que son « **parent** »

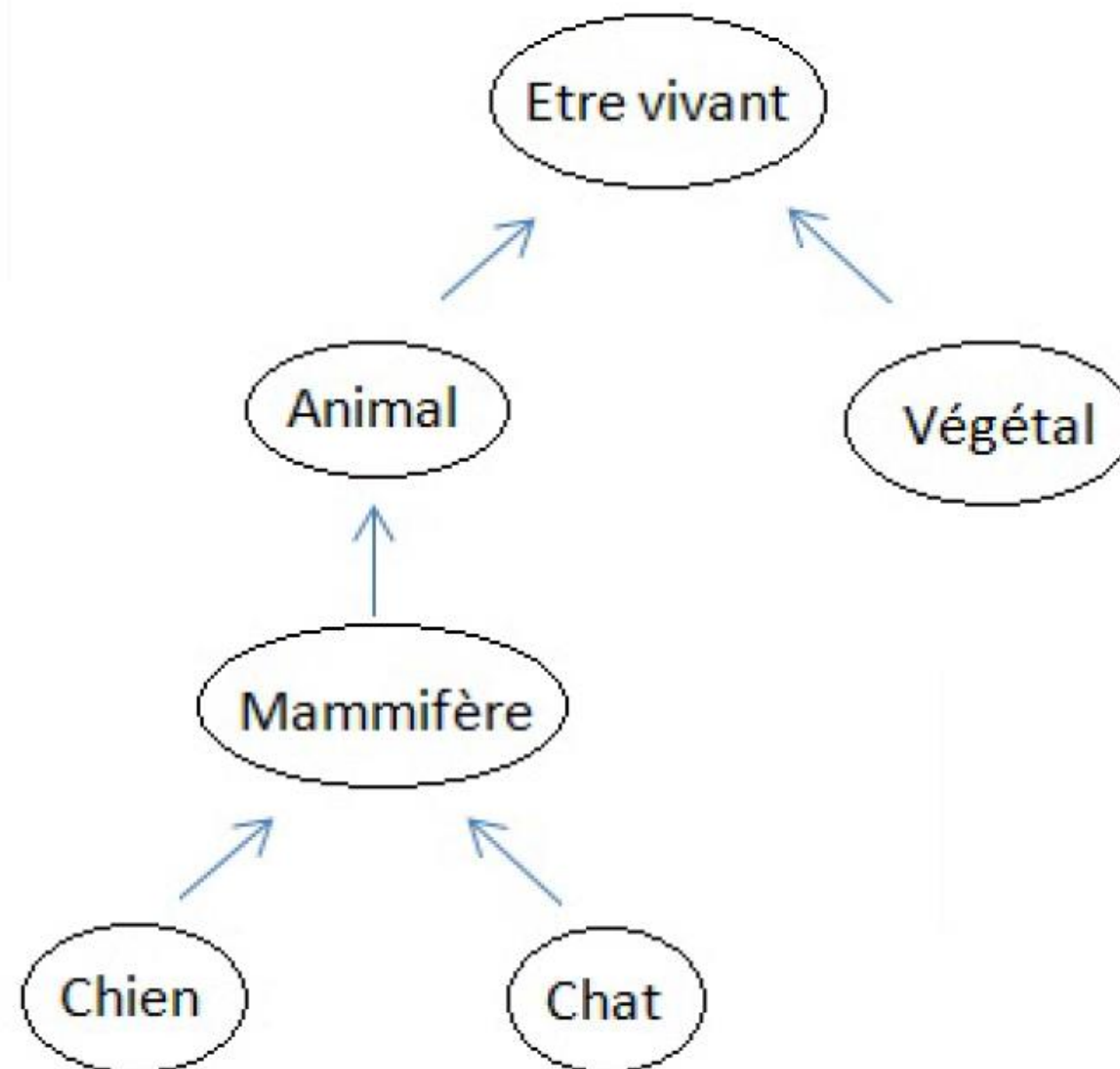
# DÉFINITION DE L'HÉRITAGE

## Le concept 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
  - L'héritage induit une sorte de relation entre les objets « **parent** » et « **enfant** » qui le « **spécialise** »
  - L' « **enfant** » aura donc les caractéristiques du « **parent** » auxquelles s'ajoute ses « **spécificités** »

# DÉFINITION DE L'HÉRITAGE

Le concept de l'héritage





# DÉFINITION DE L'HÉRITAGE

L'héritage d'une classe

- Afin d'indiquer qu'une classe hérite des attributs, propriétés et méthodes d'une autre classe on utilise la syntaxe suivante

```
1 référence  
public class Fleur : Vegetal
```

- Ici la classe Fleur hérite de végétal
- Elle hérite donc de ses attributs, propriétés et méthodes

# DÉFINITION DE L'HÉRITAGE

Les classes et les méthodes **abstraites** (Abstract)

- Une classe **Abstract** est une classe particulière qui ne peut pas être instanciée

```
12 références  
public abstract class EtreVivant
```

- Nous ne pourrions pas utiliser l'opérateur **new**
- Pour être utilisables, les classes abstraites doivent être héritées et les méthodes redéfinies
  - En général, les classes abstraites sont utilisées comme classe de base pour d'autres classes

# DÉFINITION DE L'HÉRITAGE

Les classes et les méthodes **abstraites** (Abstract)

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

```
7 références  
public abstract void Alimentation();
```

- Elle n'a pas de corps (pas de block de code)
- Pour être utilisables, les méthodes abstraites doivent être redéfinies
  - Une méthode « **abstract** » se trouvera obligatoirement dans une classe « **abstract** »

# DÉFINITION DE L'HÉRITAGE

Les méthodes « **Virtual** » et notions d'« **override** » et « **new** »

- Une méthode « **virtual** » est une méthode qui tend à être redéfinie.

```
7 références  
public virtual void Alimentation()  
{  
    Console.WriteLine("Tous les êtres vivant doivent manger...");  
}
```

➤ Elle sera redéfinie dans les classes qui en héritent

- Dans la Classe qui hérite de cette méthode nous pourrons « **override** » cette méthode ou encore utiliser « **new** » pour la spécialiser .

# DÉFINITION DE L'HÉRITAGE

Les méthodes « **Virtual** » et notions d'« **override** » et « **new** »

- Quand nous faisons un « **override** » alors la méthode spécifiée prendra le dessus sur la méthode de la classe mère.

```
7 références  
public override void Alimentation()  
{  
    Console.WriteLine("Je rumine... je mange de l'herbe!");  
}
```

➤ Elle sera spécifique à la classe fille, ici la classe Vache

- Mais elle peut également être redéfinie par « **new** »

```
1 référence  
public new void Alimentation()  
{  
    Console.WriteLine("Je mange des croquettes pour chien!");  
}
```

➤ Dans ce cas la méthode issue de la classe mère est prioritaire



# DÉFINITION DE L'HÉRITAGE

## Les Classes « sealed »

- Nous utilisons la notion de « sealed » quand une classe ne peut plus être héritée.
  - Elle sera la dernière classe de la lignée

```
2 références  
public sealed class Chien : Mammifere
```

- Ici notre classe Chien hérite de mammifère mais aucune class ne pourra hériter de la classe Chien

05

# LES INTERFACES

Leur définition et leur rôles



# LES INTERFACES EN C#

## Définition d'une interface en POO

- Une interface est un ensemble de signatures de méthodes publiques d'un objet
- C'est un contrat que s'engage à respecter un objet
  - Il s'agit donc d'un ensemble de méthodes accessibles depuis l'extérieur d'une classe, par lesquelles on peut modifier un objet, ou plus généralement communiquer avec lui
- Si une classe souhaite communiquer avec une interface, on dit qu'elle doit « implémenter » cette interface.
  - Elle contiendra donc l'ensemble des méthodes « abstract » de l'interface qu'elle définira.

# LES INTERFACES EN C#

Définition d'une interface en POO

- Prenons l'exemple simplifié d'une interface « IVolant »

```
10 références
interface IVolant
{
    3 références
    void Decoller();

    5 références
    void Voler();

    4 références
    void Atterrir();
}
```

- Elle contient 3 méthodes à implémenter: Decoller(), Voler(), Atterrir().

# LES INTERFACES EN C#

Définition d'une interface en POO

- Toute les classes souhaitant l'implémenter (répondre à son contrat) devront avoir ses méthodes complétées.

```
0 références
class Avion : IVolant
{
    4 références
    public void Atterrir()
    {
        Console.WriteLine("J'attéris sur une piste");
    }

    3 références
    public void Decoller()
    {
        Console.WriteLine("Je décolle depuis une piste");
    }

    5 références
    public void Voler()
    {
        Console.WriteLine("Je peux voler grâce à des réacteurs");
    }
}
```



# LES INTERFACES EN C#

Définition d'une interface en POO

- Ainsi, l'interface nous permet de dialoguer avec des types très diversifiés

```
0 références
class Oiseau : IVolant
{
    4 références
    public void Atterrir()
    {
        Console.WriteLine("J'atterris... comme je peux!");
    }

    3 références
    public void Decoller()
    {
        Console.WriteLine("Je décolle depuis la branche d'un arbre");
    }

    5 références
    public void Voler()
    {
        Console.WriteLine("Je peux voler grâce à des ailes");
    }
}
```

# LES INTERFACES EN C#

Définition d'une interface en POO

- Elle nous permet maintenant d'utiliser ces objets de type pourtant différent en attribut d'une classe qui les emploiera

```
2 références
class TransportColis
{
    private IVolant volant;

    1 référence
    public TransportColis(IVolant v)
    {
        volant = v;
    }

    1 référence
    public void Transporter()
    {
        volant.Voler();
        Console.WriteLine("Et le livre des colis !");
    }

    1 référence
    public void livrer()
    {
        volant.Atterrir();
    }
}
```



06

# LES GÉNÉRIQUES

Piles...? List...? Kézako?

# LES GÉNÉRIQUES EN C#

## Définition des génériques en C#

- Les génériques permettent de créer des méthodes ou des classes qui sont indépendantes d'un type.
  - On les appellera des méthodes génériques et des types génériques
- Ils permettent de regrouper des types différents avec la même Classe
- Il existe bon nombre de classe générique
  - Queue<> => FIFO (First in First Out)
  - Dictionary<> => Collection indexé par « clé = Valeur »
  - List<> => de loin celle que vous utiliserez le plus



# LES GÉNÉRIQUES EN C#

## Définition des génériques en C#

- Les classes et les méthodes génériques combinent la réutilisabilité, la cohérence des types et l'efficacité d'une façon que leurs équivalents non génériques ne peuvent pas
- Les génériques sont plus fréquemment utilisés dans des collections et des méthodes qui agissent sur eux
- L' espace de noms [System.Collections.Generic](#) contient plusieurs classes de collection génériques.
- vous pouvez également créer des types et des méthodes génériques personnalisés pour fournir des solutions et des modèles de conception généralisés



07

# LES DÉLÉGUÉS

Notions et rôles des délégués en C#

# LES DÉLÉGUÉS EN C#

## Définition des délégués (delegate)

- Un délégué est un type qui représente des références aux méthodes avec une liste de paramètres et un type de retour particuliers
- Lorsque vous instanciez un délégué, vous pouvez associer son instance à toute méthode ayant une signature et un type de retour compatibles
  - Vous pouvez appeler la méthode par le biais l'instance de délégué
  - Les délégués sont utilisés pour passer des méthodes comme arguments à d'autres méthodes



# LES DÉLÉGUÉS EN C#

## Les gestionnaires d'évènements

- Les gestionnaires d'événements sont tout simplement des méthodes appelées par le biais de délégués
- Vous créez une méthode personnalisée, et une classe telle qu'un contrôle Windows peut appeler votre méthode lorsqu'un certain événement se produit
  - Toute méthode de n'importe quelle classe ou structure accessible qui correspond au type de délégué, peut être assignée au délégué
  - La méthode peut être une méthode d'instance ou statique

# LES DÉLÉGUÉS EN C#

## Les gestionnaires d'évènements

- Cette capacité à faire référence à une méthode en tant que paramètre fait des délégués les instruments idéaux pour définir des méthodes de rappel
- Vous pouvez écrire une méthode qui compare deux objets dans votre application
  - Cette méthode peut être utilisée dans un délégué pour un algorithme de tri
  - Étant donné que le code de comparaison est séparé de la bibliothèque, la méthode de tri peut être plus générale

# LES DÉLÉGUÉS EN C#

Les délégués ont les propriétés suivantes

- Ils sont comparables aux pointeurs de fonction C++, sauf que les délégués sont totalement orientés objet, et contrairement aux pointeurs C++ vers les fonctions membres, les délégués encapsulent une instance d'objet et une méthode
- Les délégués permettent aux méthodes d'être transmises comme des paramètres
- Les délégués peuvent être utilisés pour définir des méthodes de rappel



# LES DÉLÉGUÉS EN C#

Les délégués ont les propriétés suivantes

- Les délégués peuvent être chaînés ; par exemple, plusieurs méthodes peuvent être appelées sur un seul événement
- Les méthodes ne doivent pas nécessairement correspondre exactement au type délégué
- Les **expressions lambda** sont un moyen plus concis d'écrire des blocs de code inline. Les expressions lambda (dans certains contextes) sont compilées en types délégués

08

# LES EXCEPTIONS

La récupération des exceptions



# LES EXCEPTIONS EN C#

Qu'est-ce que les exceptions en C#

- Les exceptions nous aident à gérer les situations inattendues ou exceptionnelles qui se produisent lorsqu'un programme est en cours d'exécution
  - La gestion des exceptions utilise les mots clé « **try** » « **catch** » et « **finally** » pour essayer des actions qui peuvent échouer
  - Les exceptions peuvent être générées par le Common Language Runtime (CLR), par les bibliothèques .NET ou tierces, ou par le code d'application
- Les exceptions sont créées avec le mot clé « **throw** »

# LES EXCEPTIONS EN C#

Qu'est-ce que les exceptions en C#

- Une exception peut être levée non pas par une méthode appelée directement par votre code, mais par une autre méthode plus loin dans la pile des appels
  - Quand une exception est levée, le CLR déroulera la pile, en recherchant une méthode avec un block **catch** pour le type d'exception spécifique et l'exécutera
  - S'il ne trouve pas de bloc « catch » dans la pile des appels, il termine le processus et affiche un message à l'utilisateur
- Le programme se retrouvera donc bloqué et se fermera

# LES EXCEPTIONS EN C#

Les exceptions ont les propriétés suivantes

- Les exceptions sont des types qui dérivent tous en définitive de « System.Exception »
- Utiliser un block « **try** » autour des instruction qui peuvent lever des exceptions
- Dès qu'une exception se produit dans le bloc « **try** » le flux de contrôle passe immédiatement au premier gestionnaire d'exceptions associé présent dans la pile des appels
- En C#, le mot clé « **catch** » est utilisé pour définir un gestionnaire



# LES EXCEPTIONS EN C#

Les exceptions ont les propriétés suivantes

- Si aucun gestionnaire d'exceptions n'est présent pour une exception donnée, le programme s'arrête avec un message d'erreur
- N'interceptez pas d'exception, sauf si vous pouvez la gérer et conserver l'application dans un état connu.
- Si vous interceptez une erreur « **System.Exception** », levez là de nouveau avec le mot clé « **throw** » à la fin du bloc « **catch** »
- Si un bloc « **catch** » définit une variable d'exception, vous pouvez l'utiliser pour obtenir plus d'informations sur le type d'exception qui s'est produit

# LES EXCEPTIONS EN C#

Les exceptions ont les propriétés suivantes

- Les exceptions peuvent être générées explicitement par un programme avec le mot clé « **throw** »
- Les objets Exception contiennent des informations détaillées sur l'erreur, telles que l'état de la pile des appels et une description du texte de l'erreur
- Le code qui se trouve dans un bloc « **finally** » est exécuté même si une exception est levée.
- Utilisez un bloc « **finally** » pour libérer des ressources, par exemple pour fermer tous les flux ou fichiers qui ont été ouverts dans le bloc « **try** »