Chapitre 1 – Classes et objets

Tu sais déjà programmer. Tu sais déclarer des variables, écrire des fonctions, utiliser des listes. Maintenant, on change de perspective. En programmation orientée objet, on ne pense plus en termes d'instructions, mais en termes d'objets — des entités autonomes qui savent ce qu'elles sont et ce qu'elles peuvent faire. C# est un langage conçu pour ça, et ce premier chapitre pose la brique fondamentale : la classe.

Objectifs du chapitre

À la fin de ce chapitre, tu seras capable de :

- Expliquer la différence entre une **classe** et un **objet**
- Déclarer une classe en C# avec des **attributs** et des **méthodes**
- **Instancier** un objet à partir d'une classe
- Appeler les méthodes d'un objet instancié
- Stocker plusieurs objets dans une **liste**

5 notions-clés

#	Notion	En une phrase
1	**Classe**	Le plan de construction — le *moule* qui décrit à quoi ressemble un objet
2	**Objet**	Une instance concrète créée à partir d'une classe
3	**Attribut**	Une donnée propre à chaque objet (ex : la couleur d'une voiture)
4	**Méthode**	Une action que l'objet sait effectuer (ex : démarrer)
5	**Instanciation**	L'acte de créer un objet à partir d'une classe avec le mot-clé `new`

Classe = le moule, objet = la pièce

Imagine une usine automobile. L'ingénieur dessine un **plan de fabrication** pour la Peugeot 208 : couleur, nombre de portes, motorisation... Ce plan, c'est la **classe**.

Quand l'usine fabrique une vraie voiture à partir de ce plan, elle crée un **objet** (une instance). On peut fabriquer des dizaines de voitures à partir du même plan : chacune est un objet différent, mais elles partagent la même structure.

Classe = le plan (existe une seule fois dans le code)
Objet = la voiture réelle (on peut en créer autant qu'on veut)

Déclarer une classe en C#

```cs
class Voiture
{
    // Attributs : les données de chaque voiture
    public string Marque;
    public string Couleur;
    public int NombreDePortes;

    // Méthode : une action que la voiture peut effectuer
    public void Demarrer()
    {
        Console.WriteLine($"{Marque} démarre. Vroum !");
    }

    public void AfficherInfos()
    {
        Console.WriteLine($"Voiture : {Marque} | Couleur : {Couleur} | Portes : {NombreDePorte
    }
}
```

Ce qu'il faut retenir :

- Le mot-clé `class` déclare une nouvelle classe
- Les **attributs** (`Marque`, `Couleur`, `NombreDePortes`) stockent les données de l'objet
- Les **méthodes** (`Demarrer()`, `AfficherInfos()`) définissent ce que l'objet peut faire
- Le mot-clé `public` signifie que ces membres sont accessibles depuis l'extérieur de la classe (on reviendra sur ce point au chapitre 2)

Instancier un objet

Pour créer un objet à partir de la classe, on utilise le mot-clé **`new`** :

```cs
// Création (instanciation) d'un objet de type Voiture
Voiture maVoiture = new Voiture();

// On donne des valeurs à ses attributs
maVoiture.Marque = "Peugeot";
maVoiture.Couleur = "Rouge";
maVoiture.NombreDePortes = 5;

// On appelle ses méthodes
maVoiture.AfficherInfos();   // Voiture : Peugeot | Couleur : Rouge | Portes : 5
maVoiture.Demarrer();        // Peugeot démarre. Vroum !
```

Le point `.` (opérateur d'accès)

Pour accéder aux attributs ou aux méthodes d'un objet, on utilise le point `.` :

```
nomObjet.Attribut
nomObjet.Methode()
```

Plusieurs objets, une seule classe

C'est là que la POO devient puissante : on peut créer **autant d'objets qu'on veut** à partir du même plan.

```
// Trois voitures instanciées à partir de la même classe
Voiture voiture1 = new Voiture();
voiture1.Marque = "Peugeot";
voiture1.Couleur = "Rouge";
voiture1.NombreDePortes = 5;

Voiture voiture2 = new Voiture();
voiture2.Marque = "Renault";
voiture2.Couleur = "Bleu";
voiture2.NombreDePortes = 3;

Voiture voiture3 = new Voiture();
voiture3.Marque = "BMW";
voiture3.Couleur = "Noir";
voiture3.NombreDePortes = 5;

// Chaque objet est indépendant
voiture1.AfficherInfos();
voiture2.AfficherInfos();
voiture3.AfficherInfos();
```

Chaque objet possède ses **propres valeurs** pour chaque attribut. Modifier `voiture1.Couleur` n'affecte pas `voiture2` ni `voiture3`.

Stocker des objets dans une liste

En pratique, on travaille souvent avec de nombreux objets. On les stocke dans une `List<T>` :

```
List<Voiture> parking = new List<Voiture>();

// On crée et on ajoute chaque voiture
Voiture v1 = new Voiture();
v1.Marque = "Peugeot";
v1.Couleur = "Rouge";
v1.NombreDePortes = 5;
parking.Add(v1);

Voiture v2 = new Voiture();
v2.Marque = "Renault";
v2.Couleur = "Bleu";
v2.NombreDePortes = 3;
parking.Add(v2);

// On parcourt la liste avec foreach
foreach (Voiture v in parking)
{
```

```
19      v.AfficherInfos();
20  }
```

Exercice guidé — Le garage

Objectif

Créer une application console C# qui gère un garage de voitures.

Étape 1 — Crée la classe

Crée un nouveau projet console C# dans Rider. Ajoute un fichier `Voiture.cs` et déclare la classe `Voiture` avec les attributs et méthodes vus dans le cours.

Étape 2 — Instancie 3 voitures

Dans `Program.cs`, crée 3 voitures avec des caractéristiques différentes et affiche leurs informations.

Étape 3 — Utilise une liste

Stocke tes 3 voitures dans une `List<Voiture>` et affiche-les toutes avec un `foreach`.

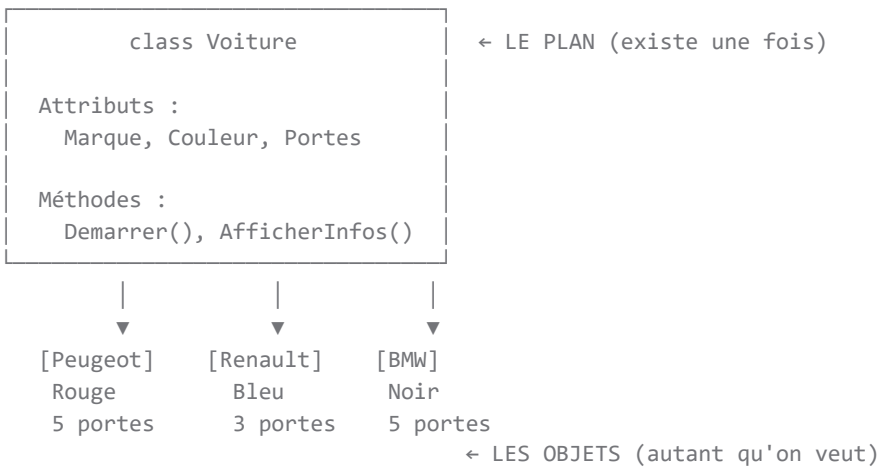
Étape 4 — Ajoute une méthode

Ajoute une méthode `Klaxonner()` à la classe. Elle affiche : `{Marque} fait : Tuuut !`. Appelle-la sur chaque voiture de ta liste.

Étape 5 — Défi

Ajoute un attribut `int Kilometrage` à ta classe. Crée une méthode `Rouler(int km)` qui **augmente** le kilométrage de la voiture et affiche un message.

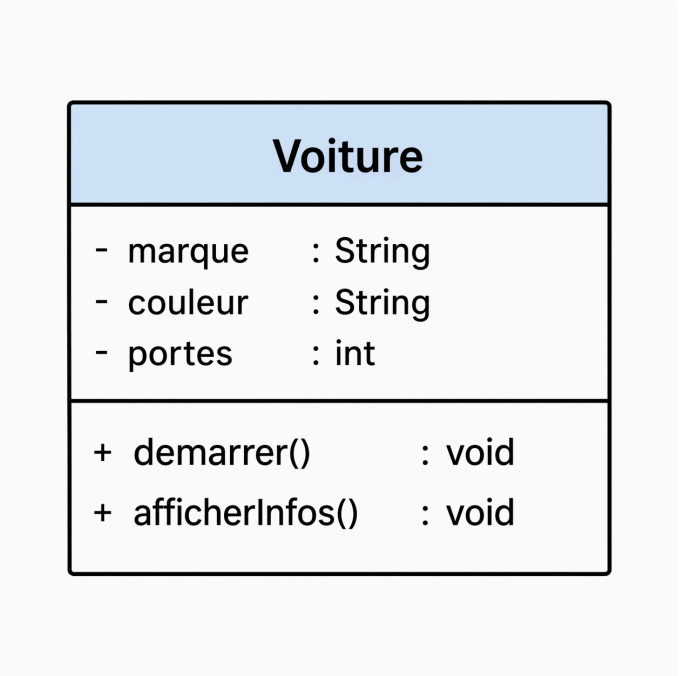
Schéma récapitulatif



Lire le schéma d'une classe

Quand on conçoit une application orientée objet, on dessine souvent les classes sous forme d'un schéma avant d'écrire le moindre code. C'est plus rapide qu'un programme, et ça permet de réfléchir à la structure sans se perdre dans la syntaxe.

Voici le schéma correspondant à notre classe `Voiture` :



Le schéma est découpé en **trois compartiments** :

En-tête — le nom de la classe, mis en évidence. C'est le nom qu'on utilisera dans le code avec le mot-clé `class`.

Attributs — chaque ligne indique un attribut avec son type.

Le signe `-` signifie que l'attribut est *privé* : on ne peut pas y accéder directement depuis l'extérieur de la classe. (On verra pourquoi c'est important au chapitre 2.)

Méthodes — chaque ligne indique une méthode avec son type de retour.

Le signe `+` signifie que la méthode est *publique* : n'importe qui peut l'appeler depuis l'extérieur.

`-` = privé · `+` = public · `: type`= ce que ça contient ou retourne

Ce schéma se lit donc comme ça :

Ce qu'on verra au chapitre suivant

Pour l'instant, nos attributs sont `public` : n'importe qui peut les modifier directement. C'est un problème — rien n'empêche d'écrire `voiture1.NombreDePortes = -12`. Au chapitre 2, on découvrira l'**encapsulation** : comment protéger les données d'un objet pour garantir qu'elles restent cohérentes.