

A large, dense pile of various colored LEGO bricks and pieces, including plates, beams, and connectors, creating a vibrant, textured background.

# Module 02 - Introduction à la programmation orientée objet

# Objectifs

- Programmation orientée objet
- Concrètement en C# ?
- Représentation UML des classes (sans relation)
- SRP

# Programmation orientée objet

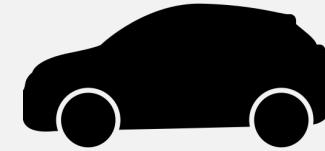
- La POO est un **paradigme** de programmation basé autour du concept **d'objets**
- Un objet contient :
  - Des **données** aussi appelées données membres, attributs, propriétés
  - Des **méthodes** aussi appelées **comportements**
- La plupart des langages de programmation orientée objet se base sur la notion de **classe** :
  - C'est le cas de C#, Java, C++
  - La classe est un type créé par l'utilisateur
  - Un objet est **l'instance** d'une classe
  - La classe est donc le **modèle** d'un objet

# Programmation orientée objet

- Classe = plan, description des attributs et des comportements

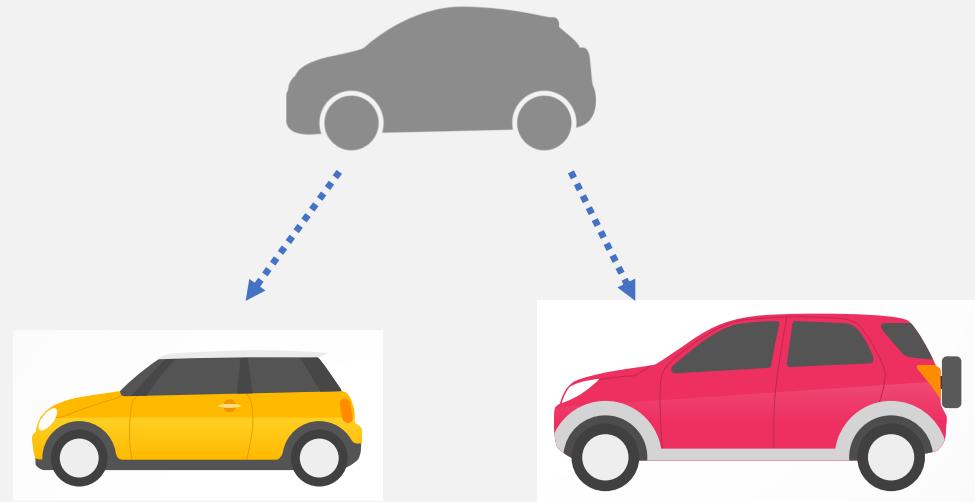
- Exemple Voiture :

- Couleur
- Nombre de portes
- Vitesse
- Accélérer
- Freiner
- SeDécrire



# Programmation orientée objet

- Objets = instances de la classe, des voitures construites (`new`) qui ont des valeurs pour chaque caractéristique
  - voiture1 :
    - Couleur = jaune
    - Nombre portes = 2
    - Vitesse = 0
  - voiture2 :
    - Couleur = rouge
    - Nombre portes = 5
    - Vitesse = 0



# Représentation en C#

- Nous allons utiliser les **classes** de C# comme vous l'avez déjà fait pour représenter les **structures**
- En fait, les classes sont des structures (données) mais avec des méthodes qui s'appliquent sur les données
- Une méthode a donc accès à ses attributs / données au travers du mot clef « **this** » qui représente l'instance courante (objet)
  - Ce mot clef est implicite et peut donc être omis en C#
  - Cependant, **dans ce cours**, dans un but pédagogique, **son utilisation est obligatoire**

# Représentation en C# - Données

- Jusqu'à maintenant, vous avez utilisé les auto-propriétés ou propriétés implicites :

```
public class Voiture
{
    public string Couleur { get; set; }
    public int NombrePortes { get; set; }
    public double Vitesse { get; set; }
}
```

# Représentation en C# - Données

- Vous pouvez aussi déclarer explicitement les propriétés :

```
public string Couleur { get; set; }
```



```
private string m_couleur;  
public string Couleur  
{  
    get  
    {  
        return this.m_couleur;  
    }  
    set  
    {  
        this.m_couleur = value;  
    }  
}
```

« *value* » est un paramètre implicite

# Représentation en C# - Données

- Vous pouvez aussi déclarer explicitement les propriétés :

```
public class Voiture
{
    private string m_couleur;
    public string Couleur
    {
        get
        {
            return this.m_couleur;
        }
        set
        {
            this.m_couleur = value;
        }
    }
}
```

```
private int m_nombrePortes;
public int NombrePortes
{
    get
    {
        return this.m_nombrePortes;
    }
    set
    {
        this.m_nombrePortes = value;
    }
}
// Vitesse ...
```

# Représentation en C# - Données

- Concrètement une auto-propriété est un triplet :
  - Une méthode « get »
  - Une méthode « set »
  - Une donnée membre privée du même type que la variable
- C'est un sucre syntaxique offert par C#
- **Sauf indications contraires ou contraintes d'utilisation, utilisez les auto-propriétés**

# Représentation en C# - Données

- Cas où ne pas utiliser les auto propriétés :
  - Valeur du get calculée à partir d'un autre membre
    - Exemple : une propriété nommée Age qui renvoie l'age d'une personne en se basant une donnée membre qui représente une date de naissance
  - Validation à effectuer dans la méthode set
    - Exemple : la méthode set d'une propriété qui définit le poids d'une personne, le poids ne peut pas être négatif.

# Représentation en C# - Données

- Les données membres ne sont pas obligatoirement exposées par des propriétés, nous y reviendrons dans la partie sur l'encapsulation

# Représentation en C# - Méthodes

```
public void Accelerer(double p_acceleration, double p_dureeSecondes) {
    if (p_acceleration < 0) {
        throw new ArgumentException("L'accélération ne doit pas être négative !", "p_acceleration");
    }
    if (p_dureeSecondes < 0) {
        throw new ArgumentException("La durée ne doit pas être négative !", "p_dureeSecondes");
    }

    double futureVitesse = this.Vitesse + p_acceleration * p_dureeSecondes;

    if (futureVitesse > 300_000 * 3600) {
        throw new InvalidOperationException("Einstein n'est pas d'accord !");
    }

    this.Vitesse += p_acceleration * p_dureeSecondes;
}
```

# Représentation en C# - Méthodes

```
public void Freiner(double p_deceleration, double p_dureeSecondes) {
    if (p_deceleration > 0) {
        throw new ArgumentException("La décélération doit être négative !", "p_deceleration");
    }
    if (p_dureeSecondes < 0) {
        throw new ArgumentException("La durée ne doit pas être négative !", "p_dureeSecondes");
    }

    double futureVitesse = Math.Max(this.Vitesse + p_deceleration * p_dureeSecondes, 0);

    this.Vitesse = futureVitesse;
}

public string SeDecrirer()
{
    return $"Je suis une voiture {this.Couleur}, j'ai {this.NombrePortes} roues et je roule à {this.Vitesse} km / h.";
}
```

```
public class Voiture {
    public string Couleur { get; set; }
    public int NombrePortes { get; set; }
    public double Vitesse { get; set; }

    public void Accelerer(double p_acceleration, double p_dureeSecondes) {
        if (p_acceleration < 0) {
            throw new ArgumentException("L'accélération ne doit pas être négative !", "p_acceleration");
        }
        if (p_dureeSecondes < 0) {
            throw new ArgumentException("La durée ne doit pas être négative !", "p_dureeSecondes");
        }

        double futureVitesse = this.Vitesse + p_acceleration * p_dureeSecondes;

        if (futureVitesse > 300_000 * 3600) {
            throw new InvalidOperationException("Einstein n'est pas d'accord !");
        }

        this.Vitesse += p_acceleration * p_dureeSecondes;
    }

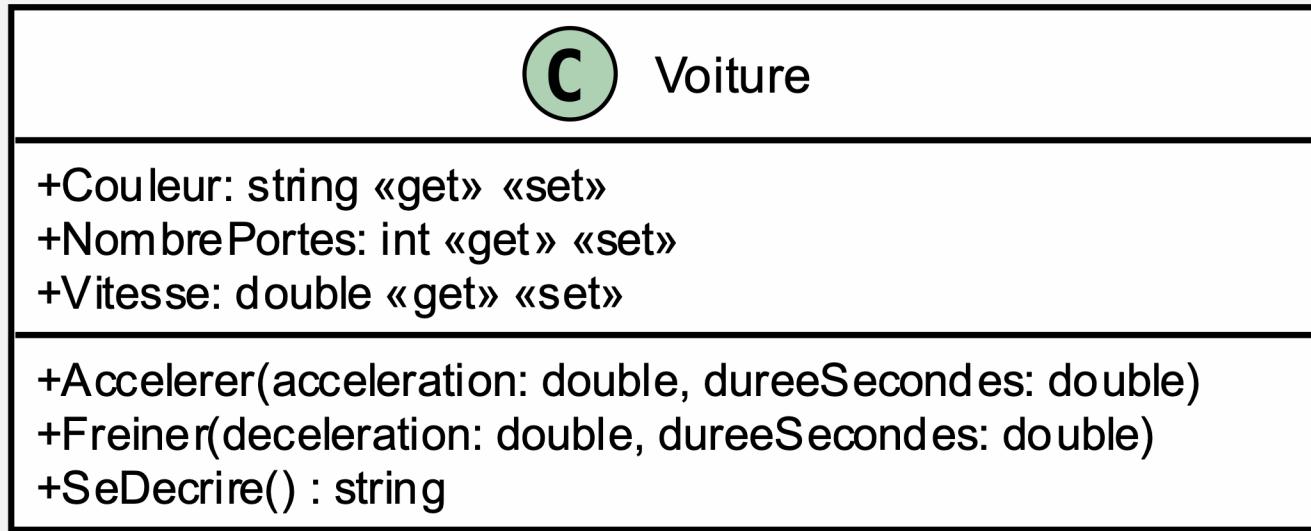
    public void Freiner(double p_deceleration, double p_dureeSecondes) {
        if (p_deceleration > 0) {
            throw new ArgumentException("La décélération doit être négative !", "p_deceleration");
        }
        if (p_dureeSecondes < 0) {
            throw new ArgumentException("La durée ne doit pas être négative !", "p_dureeSecondes");
        }

        double futureVitesse = Math.Max(this.Vitesse + p_deceleration * p_dureeSecondes, 0);

        this.Vitesse = futureVitesse;
    }

    public string SeDecrirer()
    {
        return $"Je suis une voiture {this.Couleur}, j'ai {this.NombrePortes} roues et je roule à {this.Vitesse} km / h.";
    }
}
```

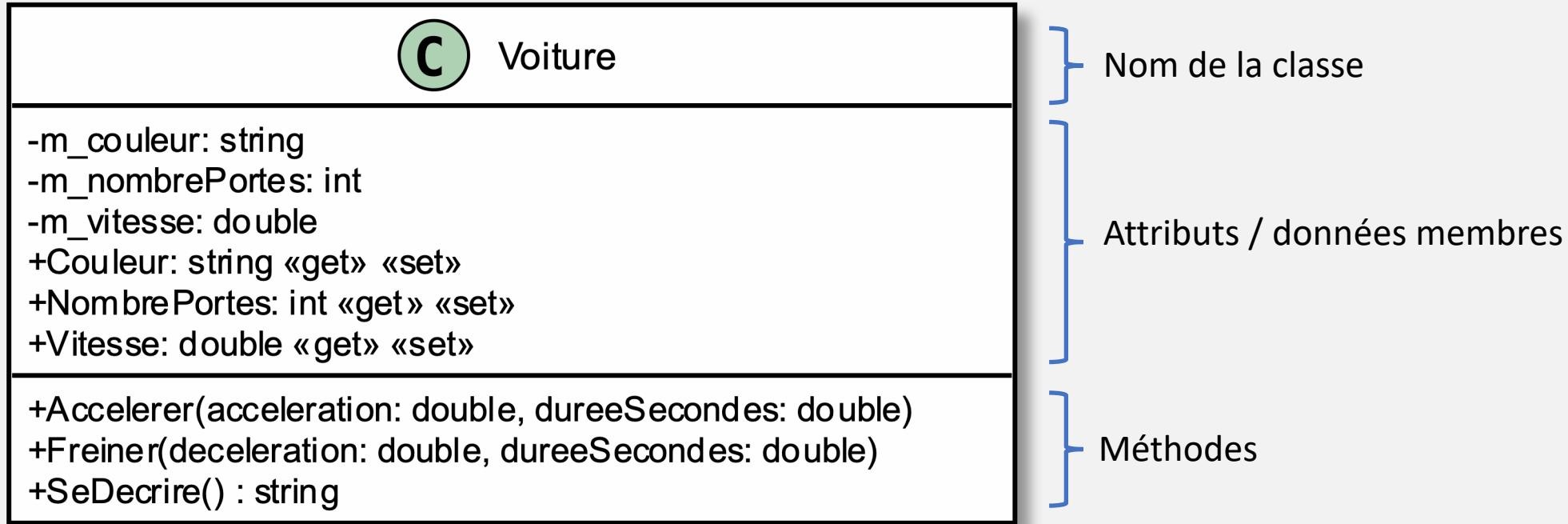
# UML – Diagramme de classes



]} Nom de la classe  
]} Attributs / données membres  
]} Méthodes

- : private  
+ : public

# UML – Diagramme de classes



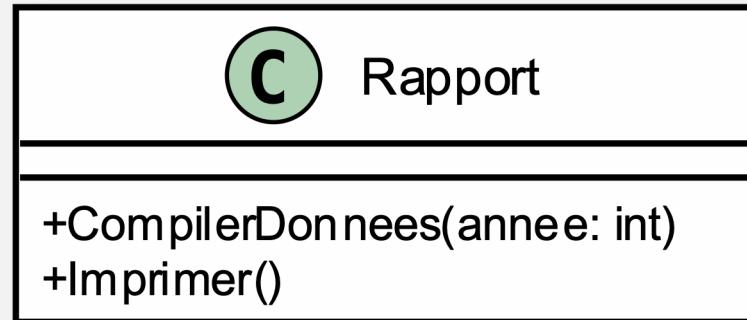
- : private  
+ : public

# Principe de responsabilité unique

- Le principe de responsabilité unique ou SRP (*Single Responsability Principle*) est le premier des 5 principes SOLID (Robert C. Martin)
- Le SRP stipule qu'une classe ne doit avoir qu'une seule responsabilité
- Pour expliciter la notion de responsabilité, oncle Bob le traduit par le fait qu'une classe ne doit **changer que pour une seule raison**
- La notion de responsabilité est donc contextuelle
- Vous avez appliqué ce principe avec les fonctions
- Si une classe a plusieurs responsabilités, vous devez la scinder en autant de classes que de responsabilités

# Principe de responsabilité unique

- Contre exemple : une classe a les responsabilités de compiler et d'imprimer un rapport.



- Ici, la classe peut changer pour deux raisons :
  - Le contenu peut changer (calcul des taxes, promotions, etc.)
  - Le format du rapport peut aussi changer (graphismes, supports : papier/web/courriel, etc.)