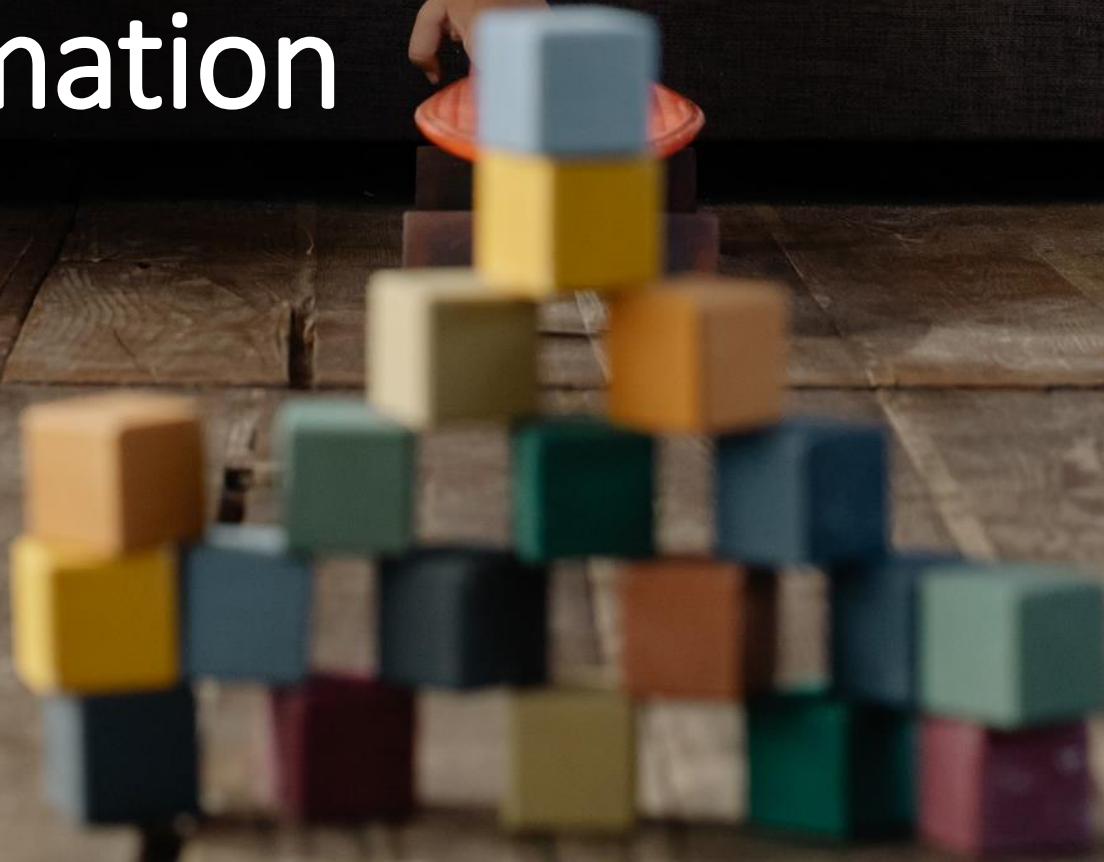


Programmation Orientée Objets



Origine

Programmation structurée ou procédurale limitée

Programme de + en + complexe

Systèmes d'exploitation graphiques



Approche « objet »

Décomposer un problème en un certain nombre d'entités indépendantes les unes des autres

Résoudre un problème :

- Identification des constituants du système
- Analyse des relations qui existent entre eux

Exemple : modélisons...

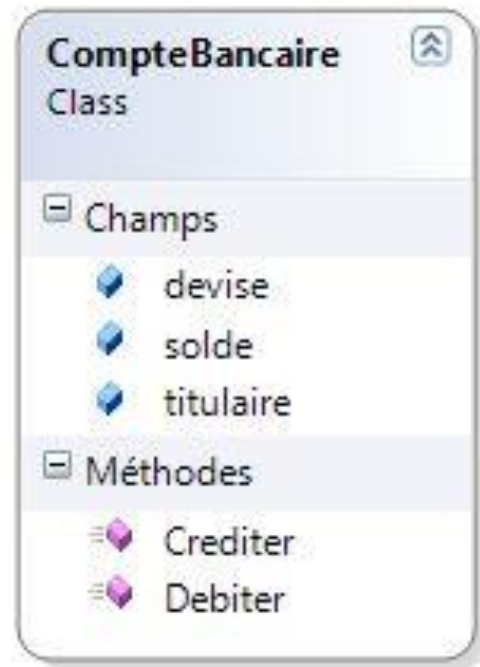


...une recette de cuisine !



A vous de « jouer » !

- Modéliser un compte bancaire



Avantages

- Grande structuration du projet
 - Réflexion préalable avant le codage
 - Décomposition naturelle du système
- Indépendance des composants
 - Développement, tests et maintenance indépendants pour chaque composant
 - Intégration plus facile grâce aux interfaces
 - Documentation



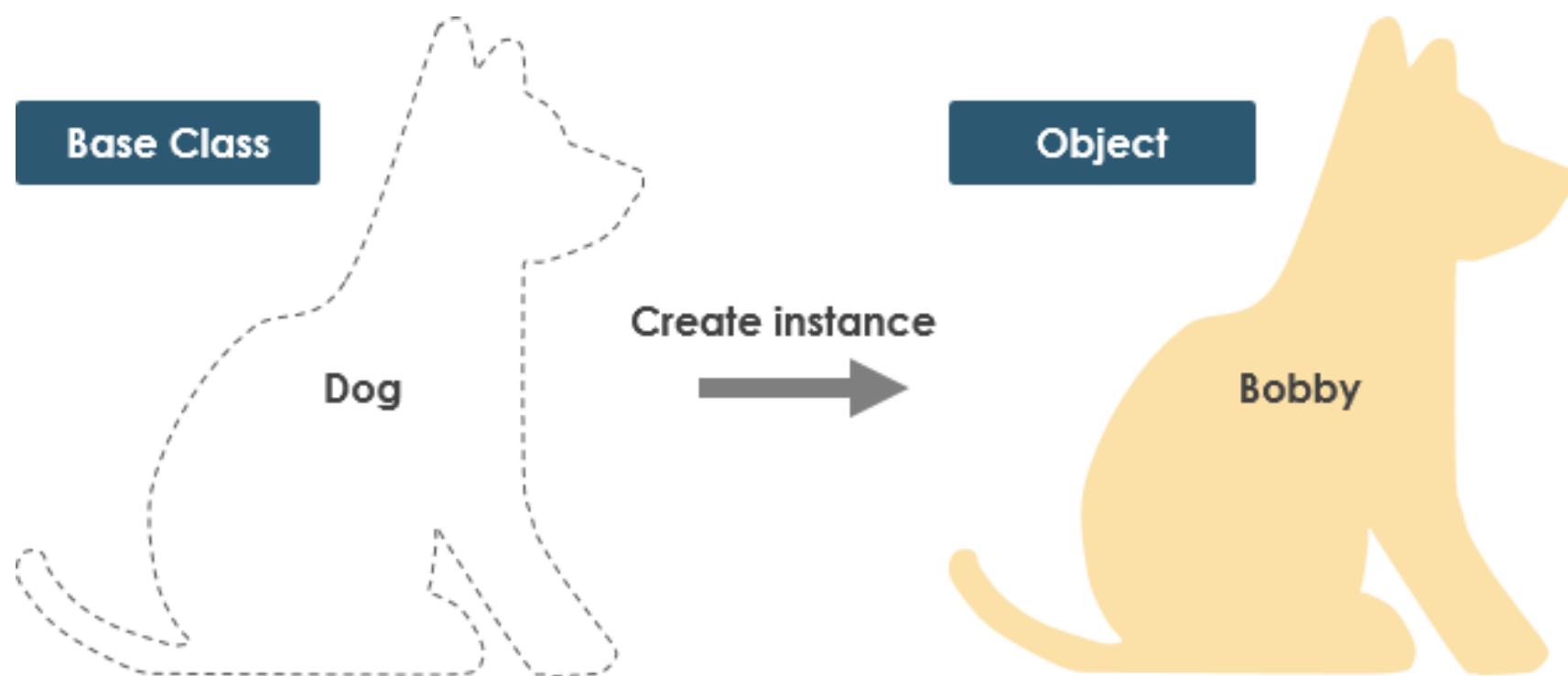
Inconvénients

- Conception plus longue
- Respecter les règles de l'art ou...
 - Système extrêmement complexe...
 - ... et difficile à faire évoluer
 - Composants inutiles
 - Mauvaises performances



Concepts de base

- **Classe** : un nouveau type de données que nous créons, composé de **membres** de 2 sortes :
 - **Attributs** : données qui décrivent l'état de la classe
 - **Méthodes** : fonctions pour agir sur les données
- **Objet** = instance de classe (les "variables" qui vont utiliser notre nouveau type)

**Properties**

Color

Eye Color

Height

Length

Weight

Methods

Sit

Lay Down

Shake

Come

Property Values

Color: Yellow

Eye Color: Brown

Height: 17 in

Length: 35 in

Weight: 24 pounds

Methods

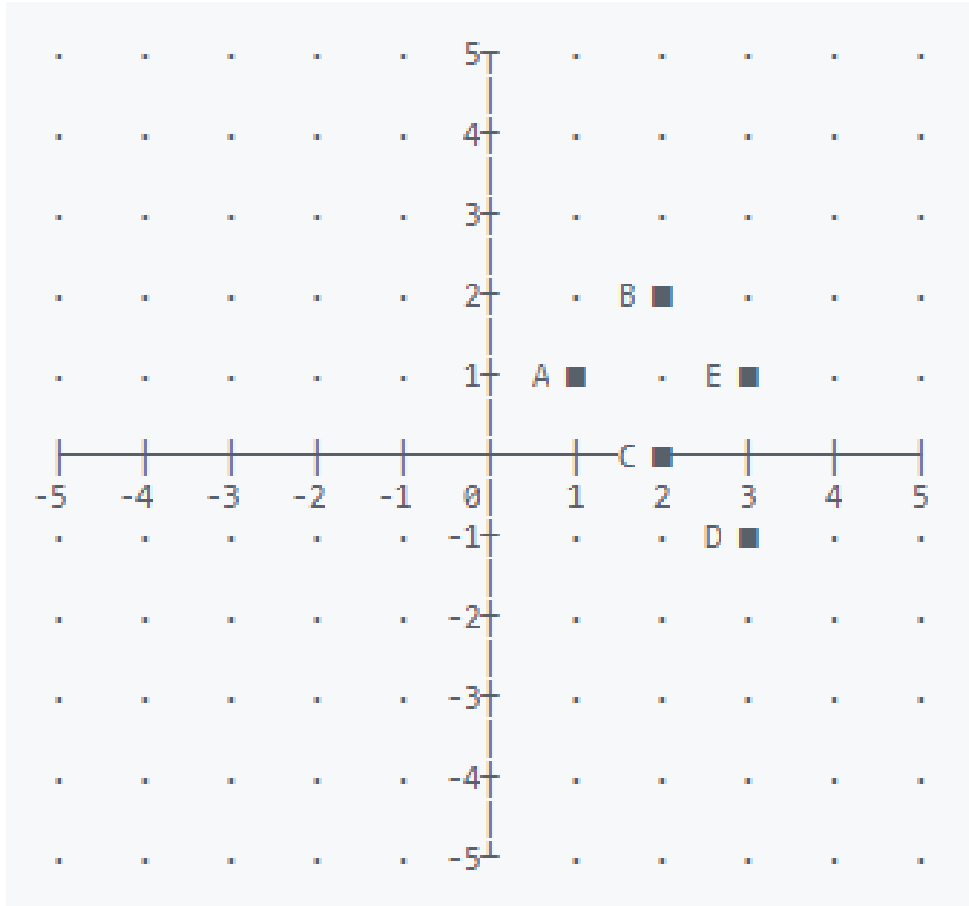
Sit

Lay Down

Shake

Come

Concepts de base - Exemple



- Pour réaliser un programme, permettant de manipuler des points dans un espace à deux dimensions :

- Classe :

Point
- x : int - y : int
+ initialiser(int, int) + déplacer(int, int) + afficher()

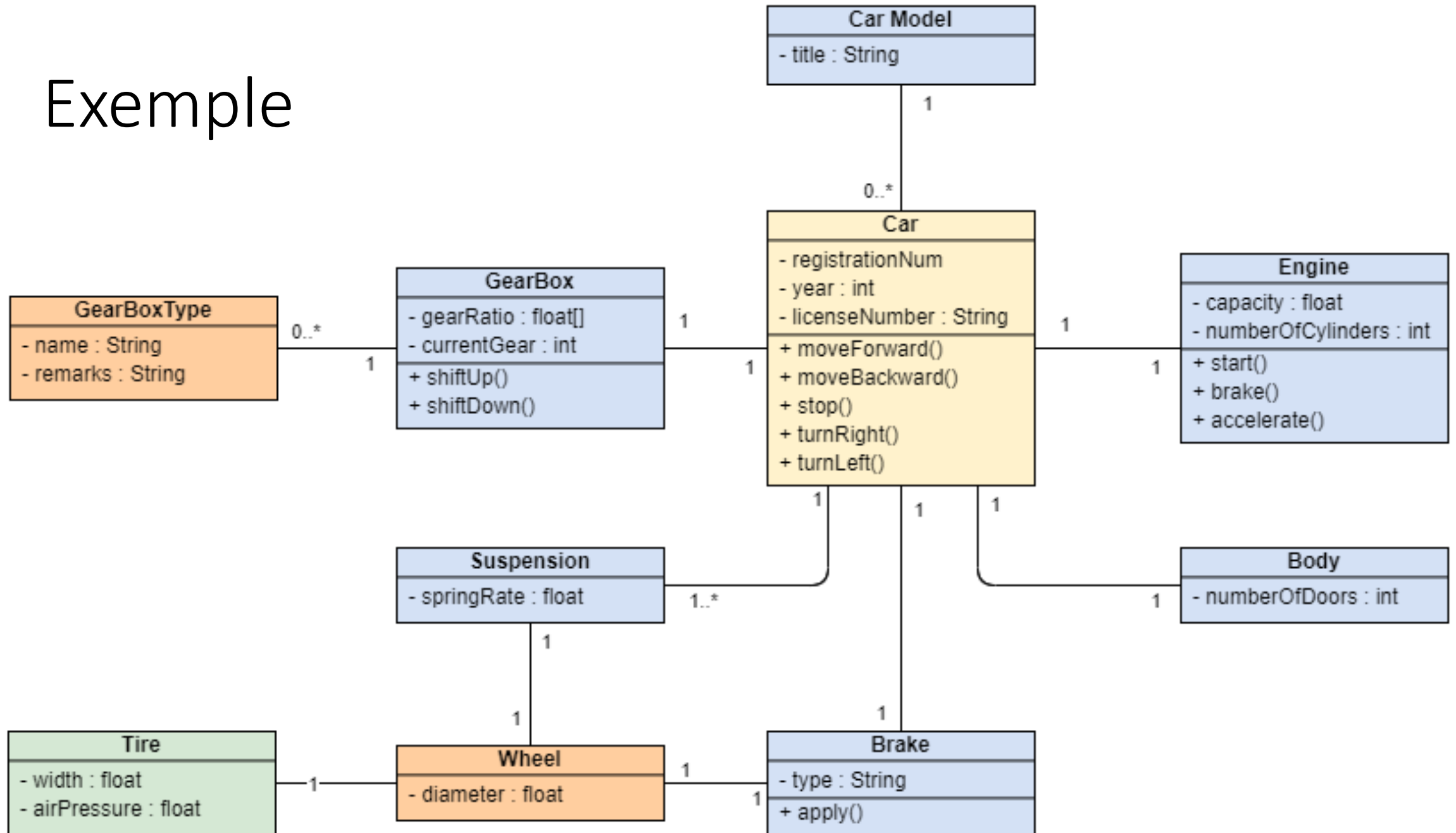
- Objets : A(1, 1), B(2, 2), C(2, 0)...

Diagramme de classes

- Unified Modeling Language (UML)
- Schéma présentant les classes d'un programme et leurs relations

NomDeLaClasse
- attribut1 : type
...
+ attributN : type
+ methode1(typeParametre,...) : typeRetour
...
- methodeN(typeParametre,...) : typeRetour

Example



Spécificateurs d'accès

- Permet de régler la visibilité des attributs et des méthodes.
- Il existe 3 spécificateurs d'accès :
 - **public (+)** / public : le membre est visible par tous les objets
 - **privé (-)** / private : le membre n'est visible qu'à l'intérieur de la classe
 - **protégé (#)** / protected : notion abordée plus tard (cf. Héritage)

Exemple

- L'attribut **couleur** est public (+) :
 - On peut y accéder et le modifier depuis l'intérieur et l'extérieur (le main() par exemple) de la classe.
- L'attribut **x** est privé (-) :
 - On ne peut ni y accéder ni le modifier depuis l'extérieur de la classe (il est invisible).
 - On peut le modifier depuis les méthodes initialiser(), déplacer(), comparer() et estHorsLimite() car elles sont à l'intérieur de la classe.

Point
- x : int - y : int + couleur : string
+ initialiser(int, int) + déplacer(int, int) + comparer(Point) : bool - estHorsLimite() : bool

Exemple

- L'attribut **x** est privé (-) :
 - La méthode comparer() permet de vérifier si l'objet Point passé en paramètre est le même que l'objet courant (même x et y ou pas).

Les deux objets partageant la classe Point, dans cette méthode, **le x de l'objet passé en paramètre sera visible**, on pourra comparer les attributs x des deux objets Point.

Point
- x : int - y : int + couleur : string
+ initialiser(int, int) + deplacer(int, int) + comparer(Point) : bool - estHorsLimite() : bool

Exemple

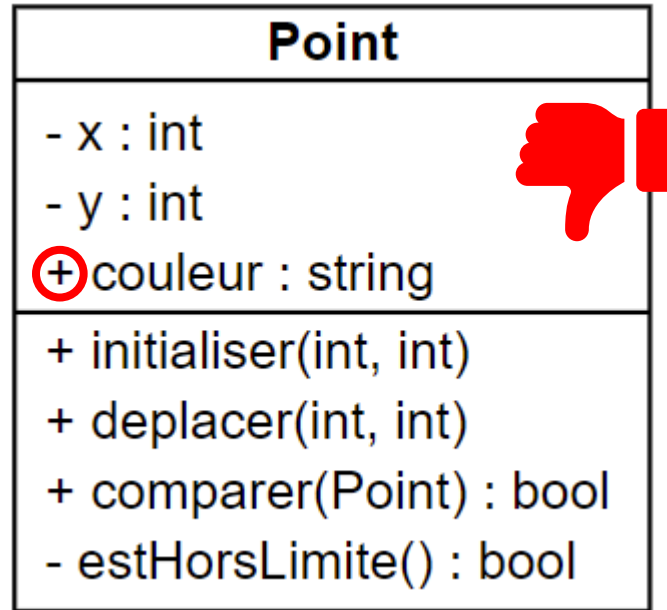
- La méthode **initialiser()** est publique (+) :
 - Elle peut être appelée depuis l'intérieur et l'extérieur (le main() par exemple) de la classe.
- La méthode **estHorsLimite()** est privée (-) :
 - On ne peut pas l'appeler depuis l'extérieur de la classe (elle est invisible)
 - On peut l'appeler depuis les méthodes initialiser(), déplacer(), comparer() et estHorsLimite() (récursivité) car elles sont à l'intérieur de la classe.

Point
- x : int - y : int + couleur : string
+ initialiser(int, int) + déplacer(int, int) + comparer(Point) : bool - estHorsLimite() : bool

Encapsulation

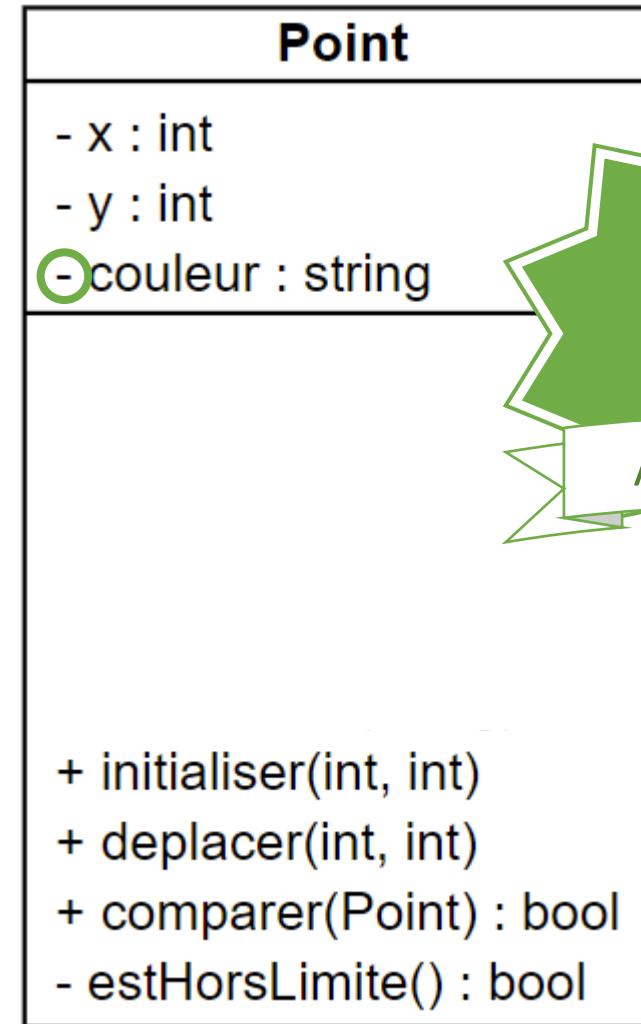
- L'encapsulation consiste à **cacher l'état interne d'un objet** et d'imposer de passer par des méthodes permettant un **accès sécurisé à l'état de l'objet** (contrôle sur les données).
- Comment la mettre en œuvre :
 1. Privatiser l'accès aux attributs
 2. Créer les méthodes d'accès (nécessaires) aux attributs
 - En lecture (**get** / accesseur)
 - En écriture (**set** / mutateur)

Exemple

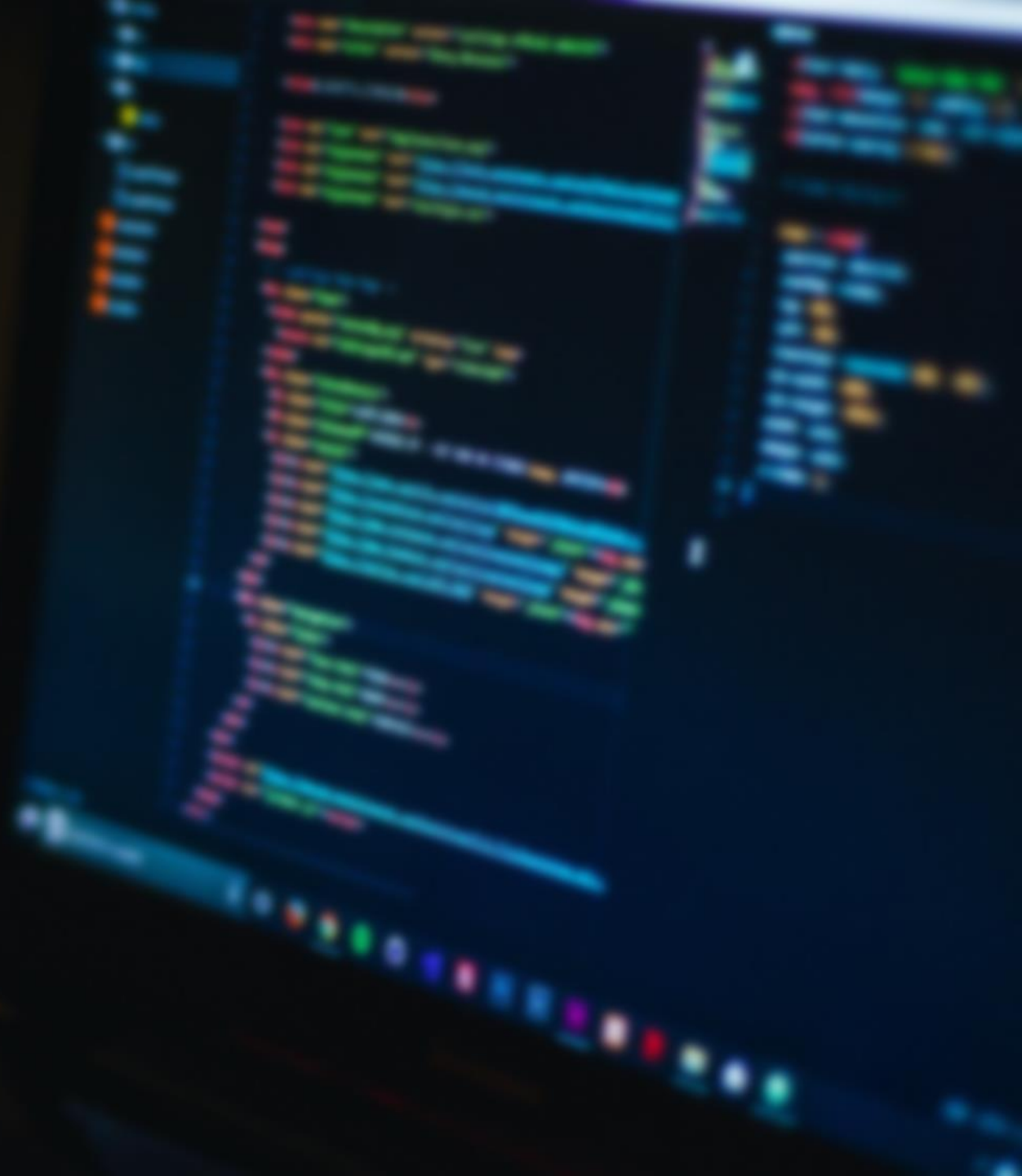


Accesseur ?

Mutateur ?



Et en C++ ?



Déclaration

```
// déclaration de la classe Point
class Point {           // nom de la classe
    private :           // déclaration des membres privés
        int x;
        int y;
    public :             // déclaration des membres publics
        void initialiser (int, int);
        void déplacer (int, int);
        void afficher();
};
```



```
// déclaration de la classe Point
class Point {
    private :
        int x;
        int y;
    public :
        void initialiser (int, int);
        void deplacer (int, int);

        // Déclaration inline
        void afficher() {
            cout << "Je suis en (" << x << ", " << y << ")" << endl;
        }
};
```

/* définition des fonctions membres en dehors de la classe

```
void Point::initialiser (int abs, int ord) {
```

```
    x = abs;
```

```
    y = ord;
```

```
}
```

:: = Opérateur de résolution de portée

```
void Point::deplacer (int dx, int dy) {
```

```
    x = x + dx;
```

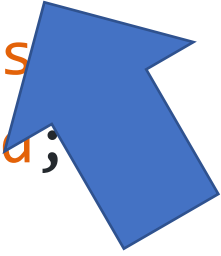
```
    y = y + dy;
```

```
}
```

```
void Point::afficher () {
```

```
    cout << "Je suis en (" << x << ", " << y << ")" << endl;
```

```
}
```



```
// Utilisation
int main() {
    Point a, b;
    a.initialiser(5, 2);
    a.afficher();           // Je suis en (5, 2)
    a.deplacer(-2, 4);
    a.afficher();           // Je suis en (3, 6)
    b.initialiser(1, -1);
    b.afficher();           // Je suis en (1, -1)
    b.x = 1;
    // ?
    return 0;
}
```



Constructeur

Méthode permettant d'initialiser les attributs d'un objet **lors de sa déclaration**.

```
class Point {  
    private :  
        int x;  
        int y;  
    public :  
        Point (int, int);    // Constructeur = pas de type retour  
        void deplacer (int, int);  
        void afficher();  
};  
Point::Point (int abs, int ord) {  
    x = abs;  
    y = ord;  
}
```

```
// Utilisation du constructeur
```

```
int main () {  
    Point a(5, 2); ← // Appel du constructeur  
    a.afficher();    // Je suis en (5, 2)  
    a.deplacer(-2, 4);  
    a.afficher();    // Je suis en (3, 6)  
  
    Point b(1, -1); ← // Appel du constructeur  
    b.afficher();    // Je suis en (1, -1)  
  
    Point c; ← // Erreur de compilation  
    c.afficher();  
  
    return 0;  
}
```



// Le constructeur peut être surdéfini :

```
class Point {  
    private :  
        int x;  
        int y;  
    public :  
        Point ();  
        Point (int);  
        Point (int, int);  
};
```



```
Point::Point() {  
    x = y = 0;  
}
```

```
Point::Point(int val) {  
    x = y = val;  
}
```

```
Point::Point(int abs, int ord) {  
    x = abs;  
    y = ord;  
}
```

// Peut aussi s'écrire :

```
Point::Point(int abs, int ord) : x(abs), y(ord) {}
```

```
// Utilisation du constructeur
int main () {
    Point a(5, 2);        // Appel du constructeur Point(int, int)
    a.afficher();         // Je suis en (5, 2)
    a.deplacer(-2, 4);
    a.afficher();         // Je suis en (3, 6)

    Point b(1, -1);       // Appel du constructeur Point(int, int)
    b.afficher();         // Je suis en (1, -1)

    Point c;              ← // Appel du constructeur Point()
    c.afficher();         // Je suis en (0, 0)

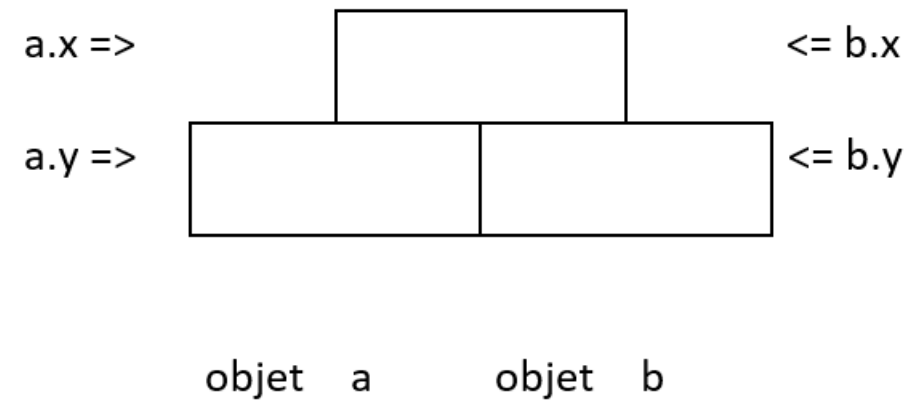
    return 0;
}
```



Attribut static

Attribut partagé par tous les objets de la classe :

```
class Point
{
    static int x;
    int y;
};
```



```
// Ne s'initialise qu'à l'extérieur de la classe
int Point::x = 1;
```

Destructeur

Un **destructeur** est une fonction membre qui est automatiquement appelée au moment de la "destruction" d'un objet, avant la libération de son espace mémoire :

- à la fin du bloc ou de la fonction pour les objets automatiques
- à la fin du programme pour les objets statiques,
- à l'aide de l'instruction **delete** ou à la fin du bloc ou de la fonction pour les objets dynamiques

Signature : `~NomDeLaClasse()`



Destructeur

Oui mais pour quoi faire ?

- Gérer les attributs static
- Gérer les attributs dynamiques

```
class ObjectCounter {  
    private:  
        static int count;  
    public:  
        ObjectCounter () {  
            cout << "++ creation : " << ++count << " objet(s)\n";  
        }  
        ~ObjectCounter () {  
            cout << "-- destruction : " << --count << " objet(s)\n";  
        }  
};  
  
int ObjectCounter::count = 0;
```



```
// Fonction qui crée 2 objets
void Creation() {
    ObjectCounter u,v;
    cout << "sortie de la fonction\n";
}

int main() {
    ObjectCounter a;
    cout << "appel de la fonction Creation\n";
    Creation();
    ObjectCounter b;
    return 0 ;
}
```

```
void Creation() {  
➡   ObjectCounter u,v;  
➡   cout << "sortie de la fonction\n";  
}  
  
int main() {  
➡   ObjectCounter a;  
➡   cout << "appel de la fonction Creation\n";  
➡   Creation();  
➡   ObjectCounter b;  
➡   return 0 ;  
}
```

++ creation : 1 objet(s)
appel de la fonction Creation
++ creation : 2 objet(s)
++ creation : 3 objet(s)
sortie de la fonction
-- destruction : 2 objet(s)
-- destruction : 1 objet(s)
++ creation : 2 objet(s)
-- destruction : 1 objet(s)
-- destruction : 0 objet(s)