Programmation Orientée Objets

Origine

Programmation structurée ou procédurale limitée

Programme de + en + complexe

Systèmes d’exploitation graphiques

Approche « objet »

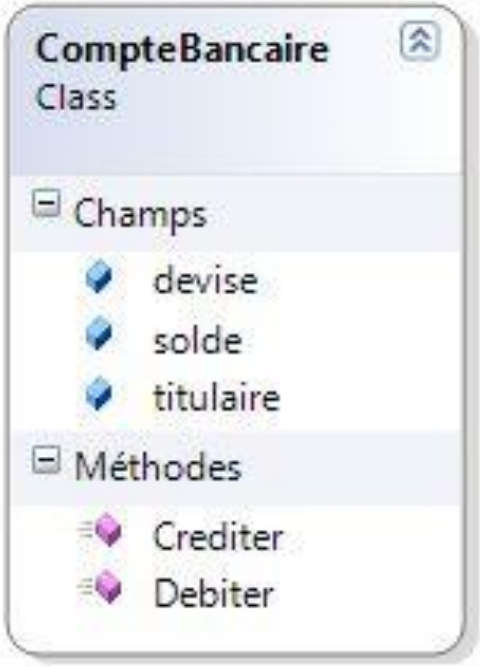
Décomposer un problème en un certain nombre d'entités indépendantes les unes des autres

Résoudre un problème :

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

Exemple : modélisons…

Un compte bancaire



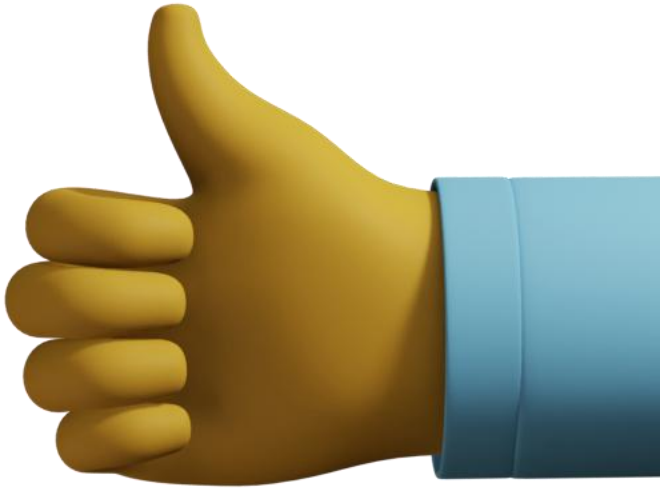
- En groupe, modélisez…

…une recette de cuisine !

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

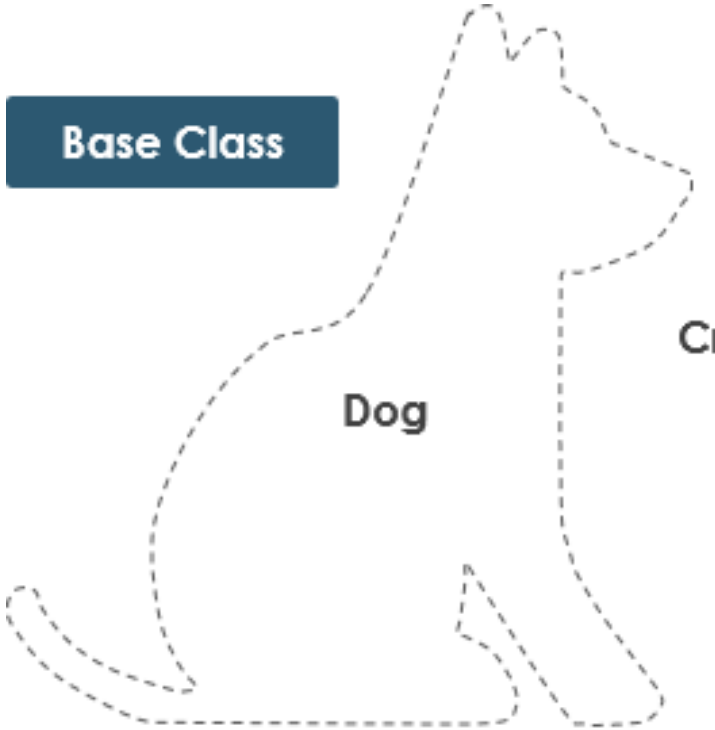
- Conception plus longue

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

Concepts de base

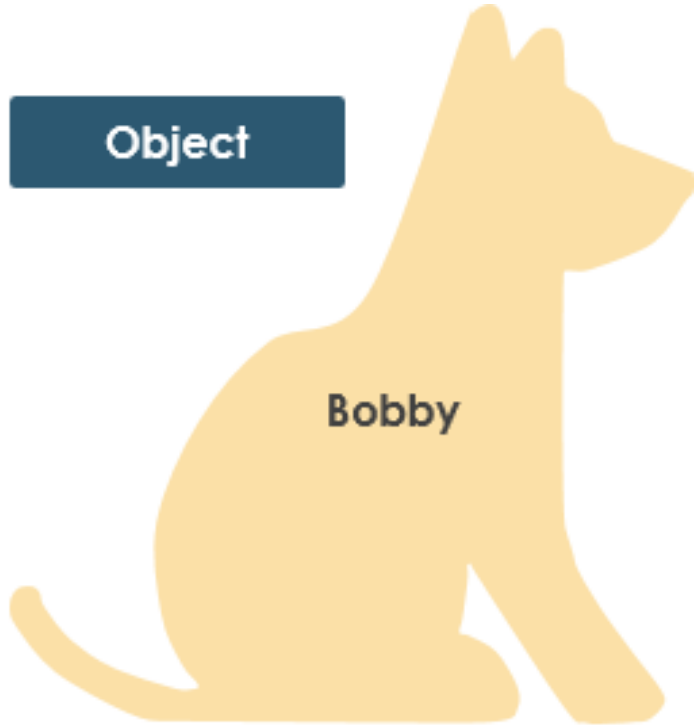
- **Classe** : un nouveau type de données que nous créons, composé de **membres** de 2 sortes :
  - **Attributs** : données qui décrivent l'état de la classe
  - **Méthodes** : fonctions pour agir sur les données
- **Objet** = instance de classe (les "variables" qui vont utiliser notre nouveau type)

Base Class



Create instance

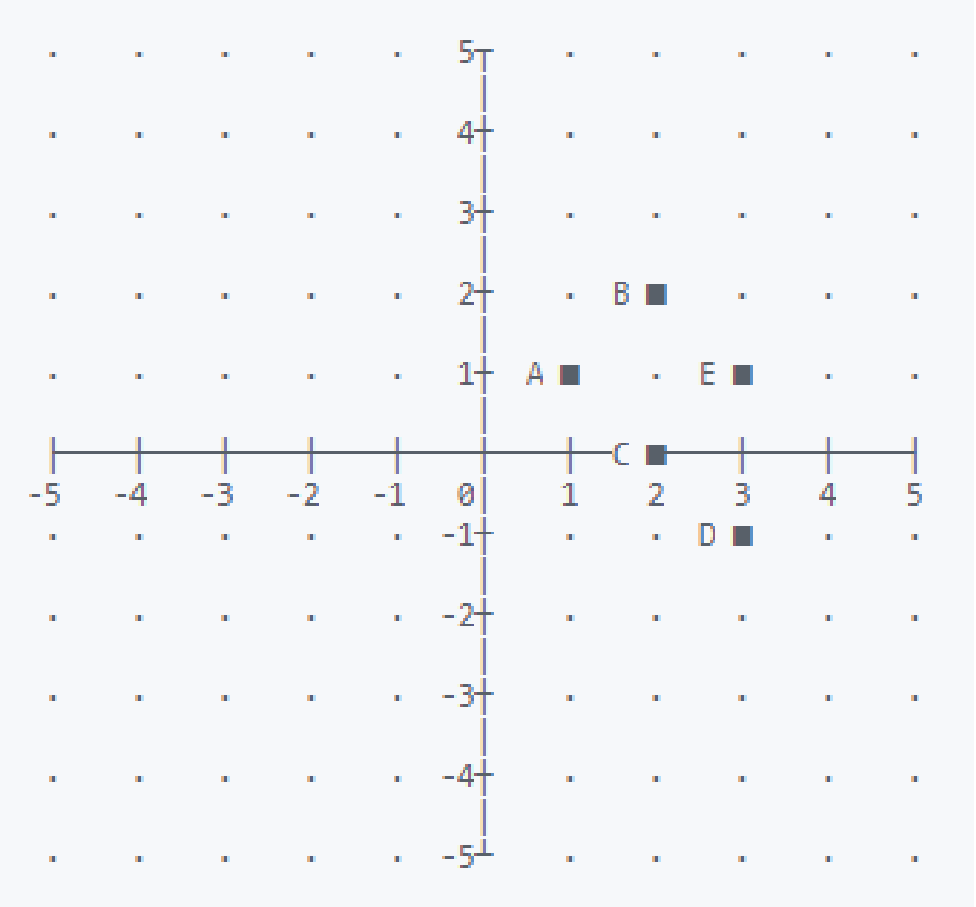
Object



Properties	Methods
Color	Sit
Eye Color	Lay Down
Height	Shake
Length	Come
Weight	

Property Values	Methods
Color: Yellow	Sit
Eye Color: Brown	Lay Down
Height: 17 in	Shake
Length: 35 in	Come
Weight: 24 pounds	

Concepts de base - Exemple



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

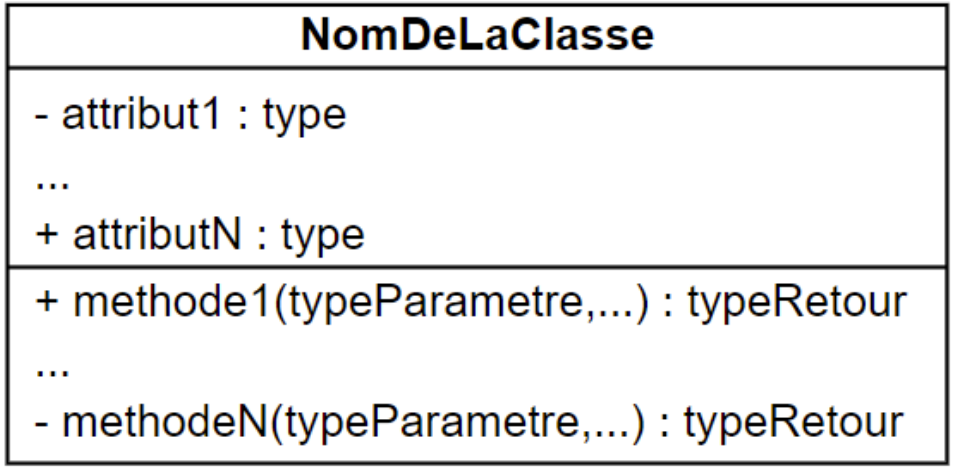
Point
+ x : int + y : int + lettre : char
+ afficher()

- Objets : A(1, 1), B(2, 2), C(2, 0)…

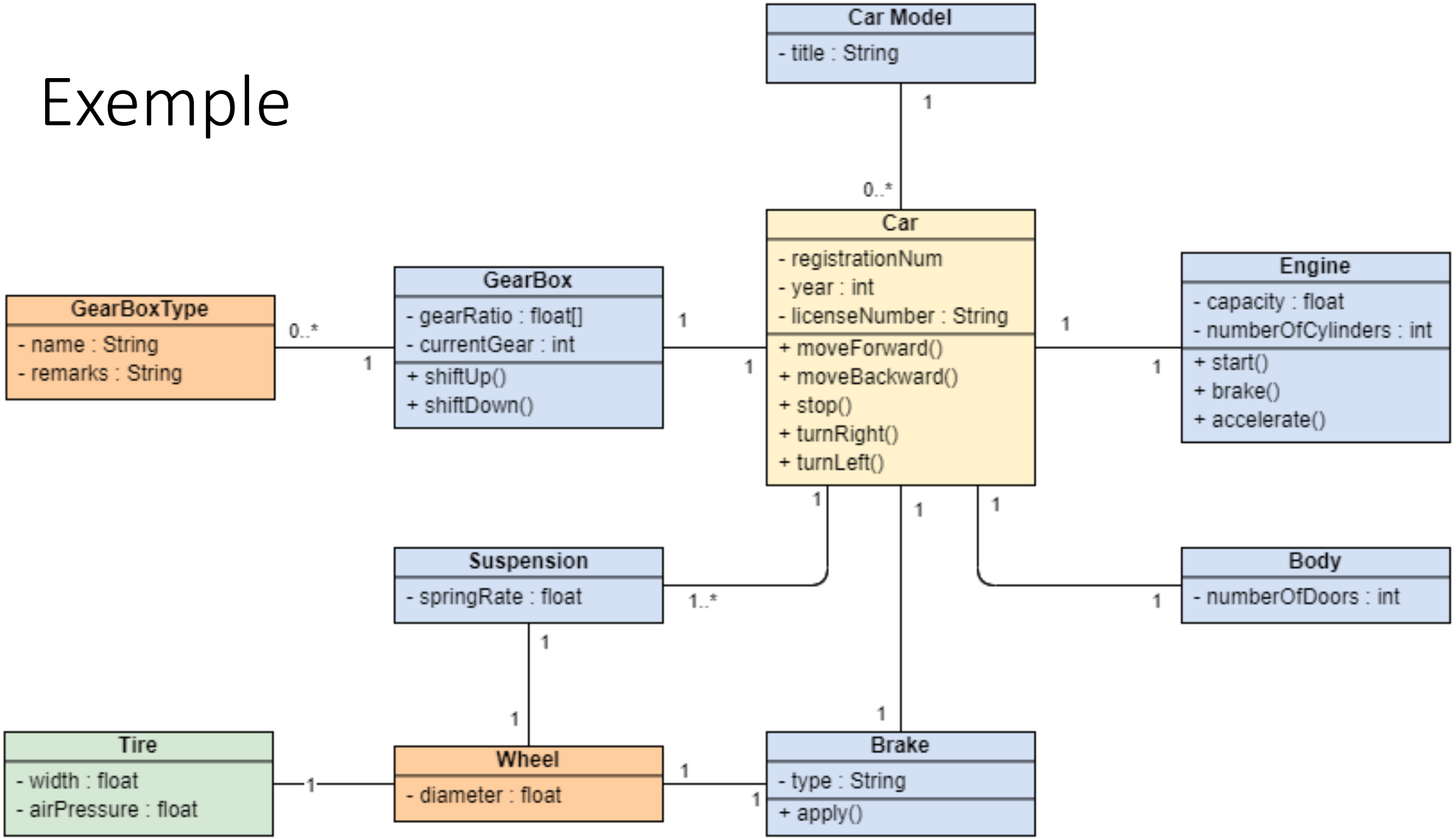
Diagramme de classes

- Unified Modeling Language (UML)

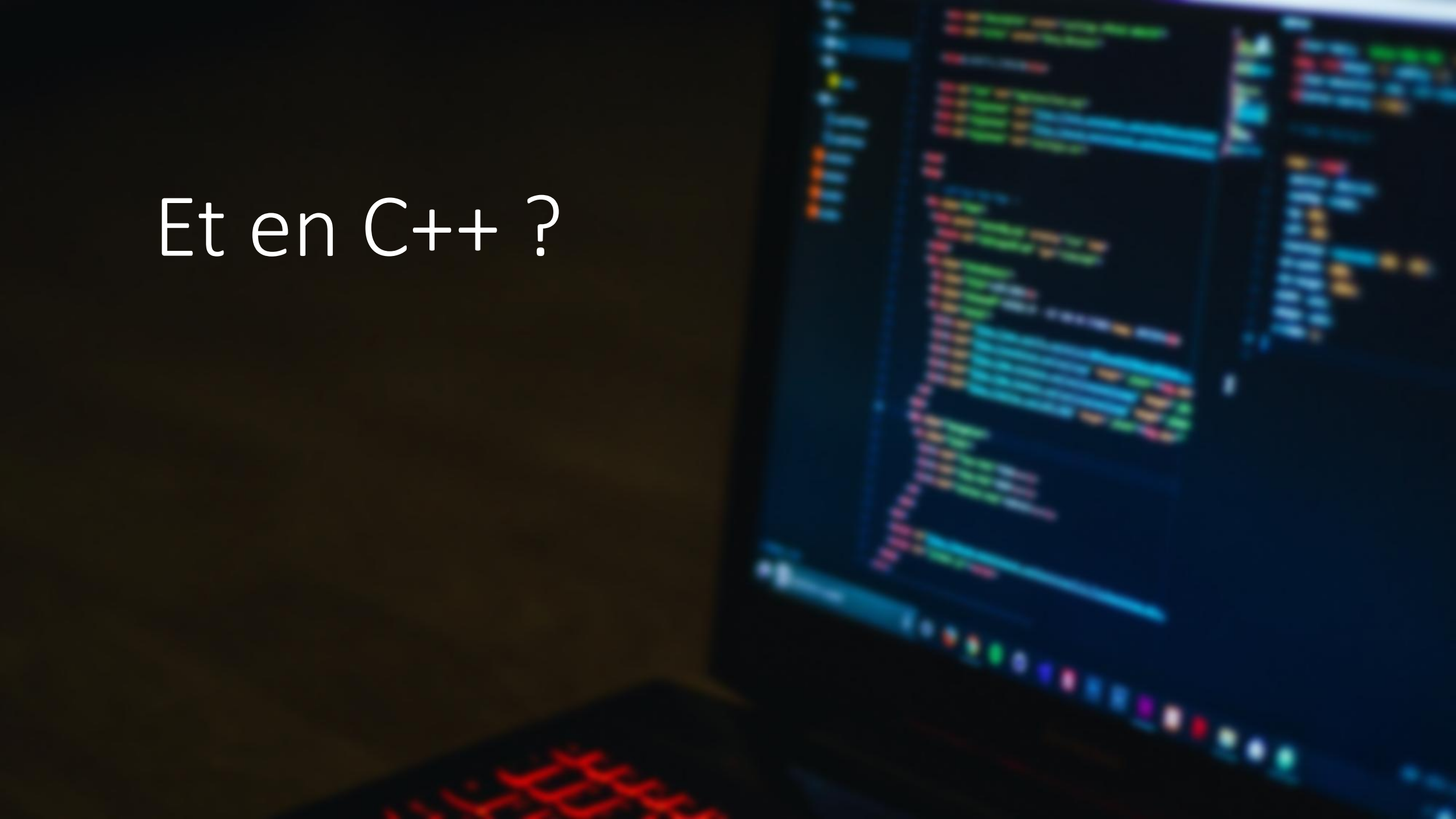
- Schéma présentant les classes d'un programme et leurs relations



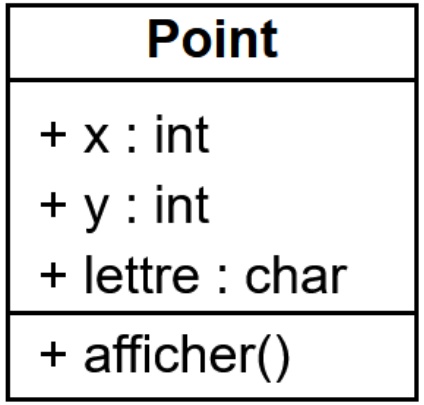
Exemple


Car Model
- title : String
1
0..*
Car
- registrationNum
- year : int
- licenseNumber : String
+ moveForward()
+ moveBackward()
+ stop()
+ turnRight()
+ turnLeft()
GearBoxType
- name : String
- remarks : String
0..*
1
GearBox
- gearRatio : float[]
- currentGear : int
+ shiftUp()
+ shiftDown()
1
1
1
1
Engine
- capacity : float
- numberOfCylinders : int
+ start()
+ brake()
+ accelerate()
1
1
1
1
1
Suspension
- springRate : float
1..*
1
Body
- numberOfDoors : int
1
1
1
Tire
- width : float
- airPressure : float
1
Wheel
- diameter : float
1
Brake
- type : String
+ apply()
1

Et en C++ ?

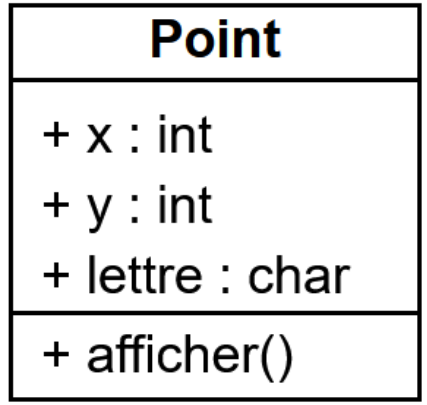


Déclaration




```
// Déclaration de la classe Point
class Point {     // Nom de la classe
    public :      // Déclaration des membres publics
        int x;
        int y;
        char lettre;
        void afficher();
};
```

Définition des méthodes #1

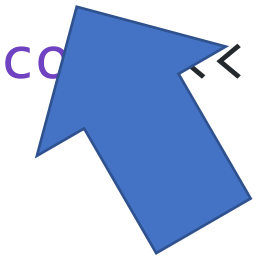



```
class Point {
    public :
        int x;
        int y;
        char lettre;
        // Définition inline
        void afficher() {
            std::cout << lettre << "(" << x << "," << y << ")";
        }
};
```

Définition des méthodes #2

Point
+ x : int + y : int + lettre : char
+ afficher()

```
// Définition en-dehors de la classe

void Point::afficher() {
    std::co    << lettre << "(" << x << "," << y << ")";
}
```



Bonnes pratiques

Déclarer chaque classe dans un fichier NomDeLaClasse**.hpp** ou **.h**

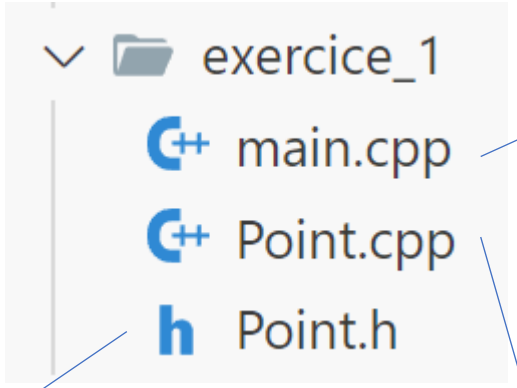
Définir les méthodes dans un fichier NomDeLaClasse**.cpp**

```
// Utilisation
int main() {
    Point p1, p2;
    p1.x = 5;
    p1.y = 2;
    p1.lettre = 'A';
    p1.afficher();          // A (5,2)
    p2.x = 3;
    p2.y = -6;
    p2.lettre = 'B';
    p2.afficher();          // B (3,-6)
    p1.afficher();          // A (5,2)
    return 0;
}
```

Bonnes pratiques

Architecture

d'un projet/exercice :




```cpp
#include "Point.h"

// Utilisation
int main() {
    Point p1, p2;
    p1.x = 5;
    p1.y = 2;
    p1.lettre = 'A';
    p1.afficher();
    p2.x = 3;
    p2.y = -6;
    p2.lettre = 'B';
    p2.afficher();
    p1.afficher();
    return 0;
}
```

```cpp
// Déclaration de la classe Point

#ifndef Point_H
#define Point_H

class Point {
    public :
        int x;
        int y;
        char lettre;
        void afficher();
};

#endif
```

```cpp
#include "iostream"
#include "Point.h"

// Déclaration en-dehors de la classe
void Point::afficher() {
    std::cout << lettre << "(" << x << "," << y << ")\n";
}
```

Bonnes pratiques

Compilation : `g++ *.cpp -o exo1.exe`

Spécificateurs d’accès

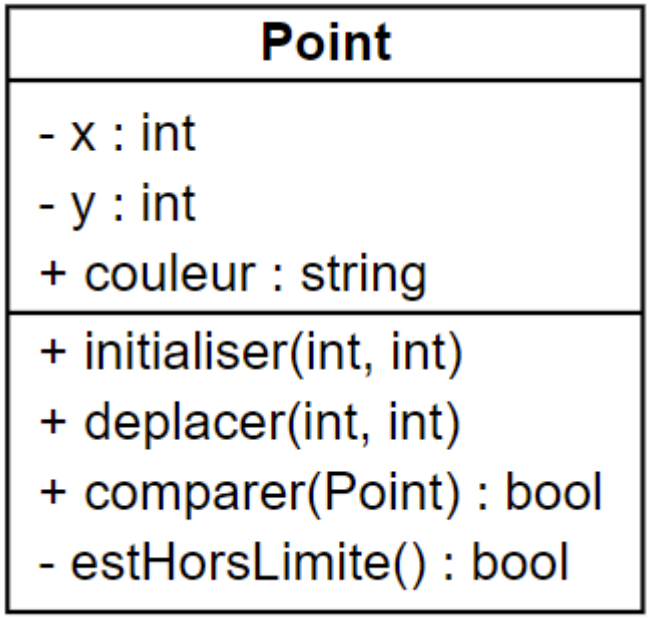
- Permet de réglementer la visibilité des attributs et des méthodes.

- 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.

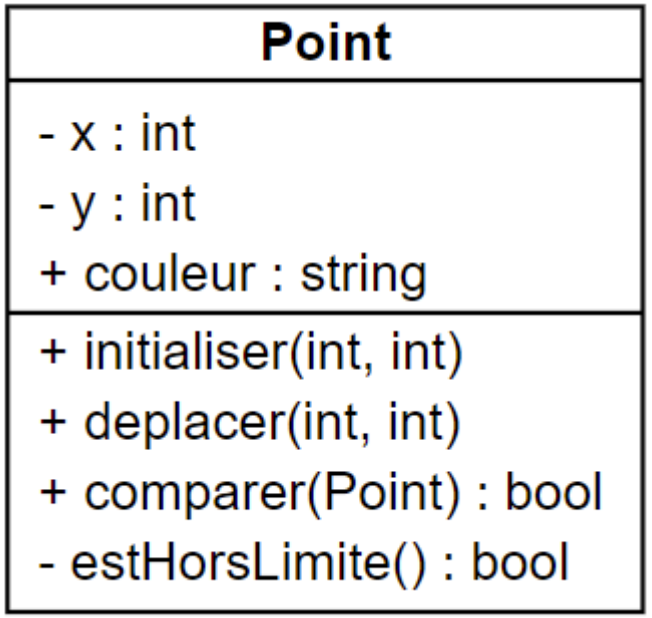
- L'attribut **x** est privé (-) :
  - On ne peut ni y accéder ni le modifier depuis l'extérieur de la classe (il est invisible).
  - On peut le modifier depuis les méthodes initialiser(), deplacer(), comparer() et estHorsLimite() car elles sont à l'intérieur de la classe.



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).

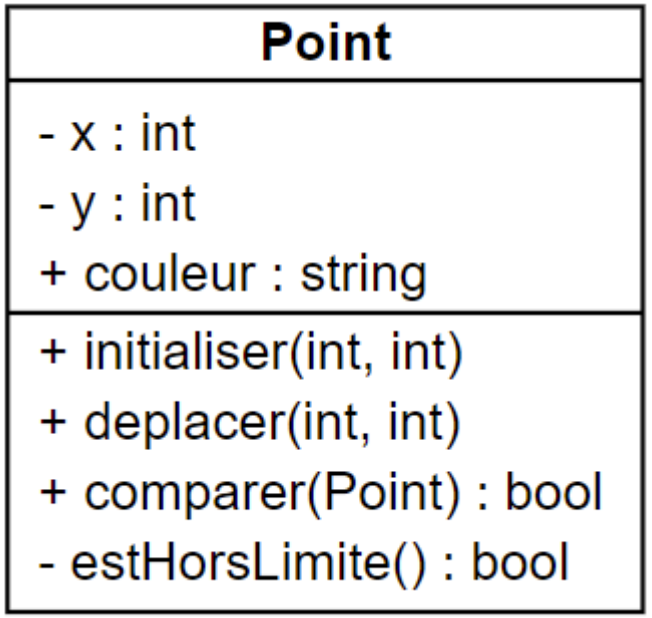
    Les deux objets partageant la classe Point, dans cette méthode, **le x de l’objet passé en paramètre sera visible**, on pourra comparer les attributs x des deux objets Point.



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(), deplacer(), comparer() et estHorsLimite() (récursivité) car elles sont à l'intérieur de la classe.

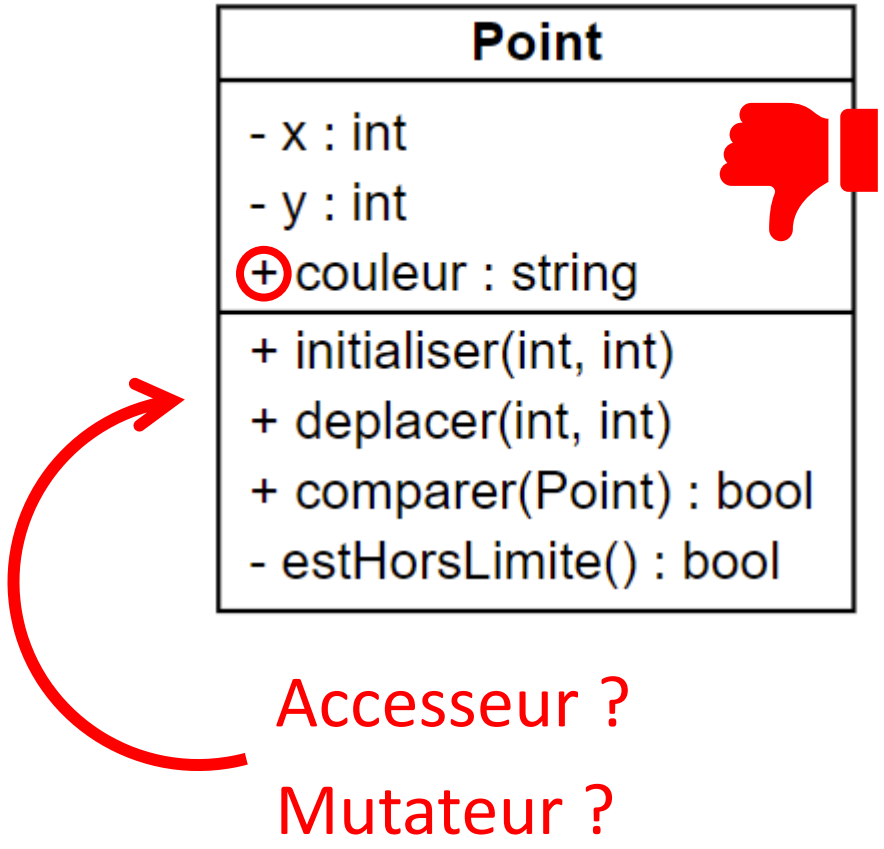


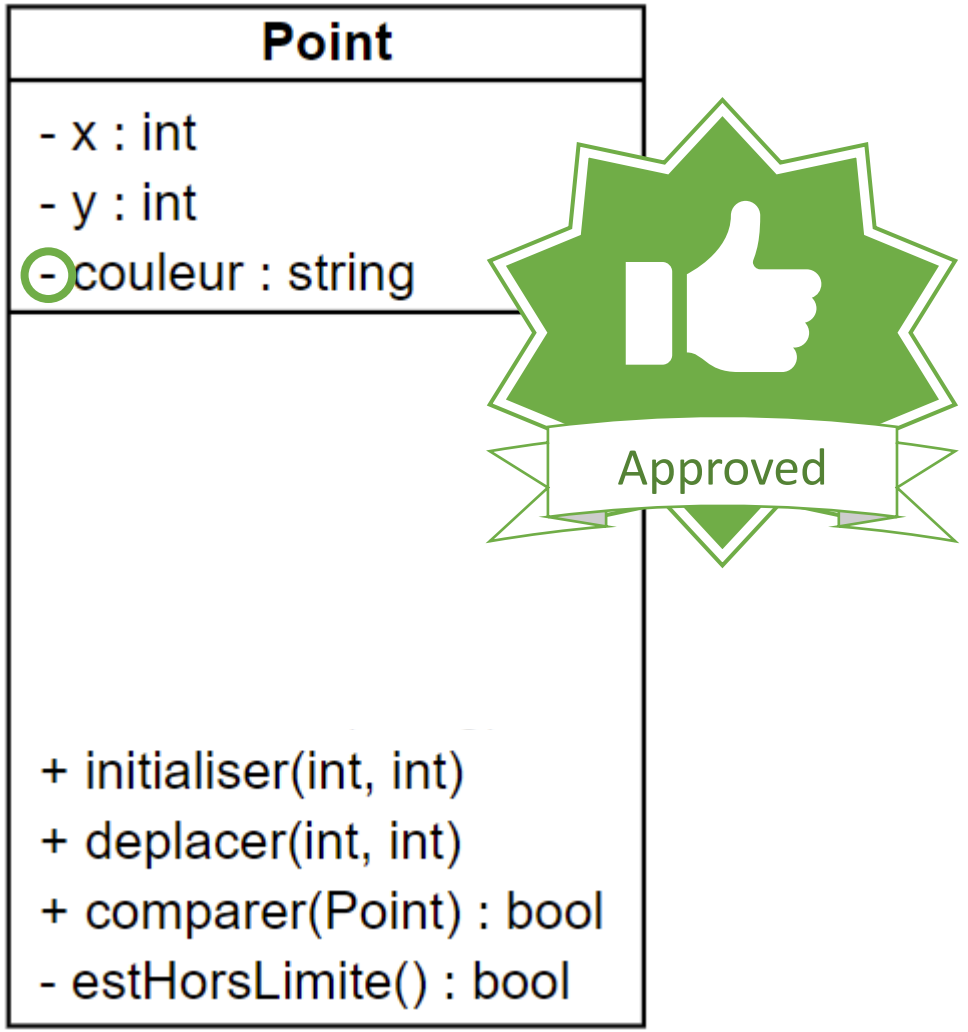
Encapsulation

- 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


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

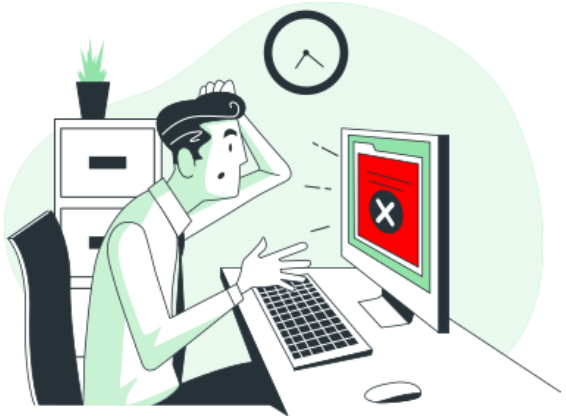

Point
- x : int
- y : int
- couleur : string
+ initialiser(int, int)
+ deplacer(int, int)
+ comparer(Point) : bool
- estHorsLimite() : bool
Approved

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);
};
```

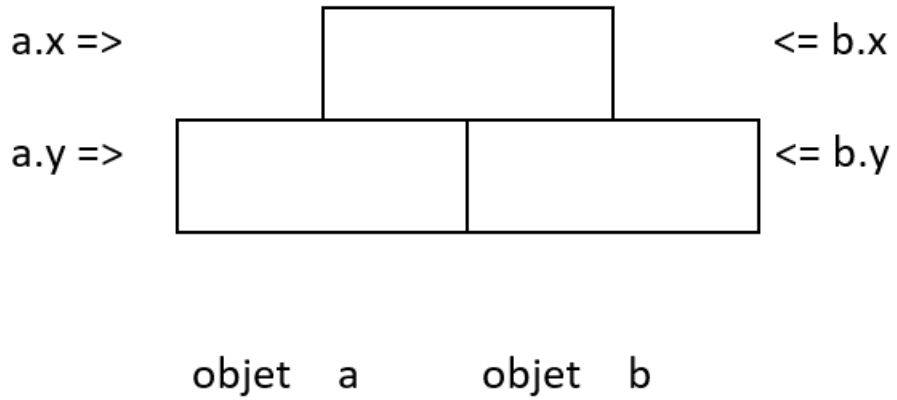
```
Point::Point() {
    x = y = 0;
}

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) {}
```

```cpp
// 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)
```