Introduction aux méthodes Orientées Objets

Deuxième partie

Modélisation avec UML 2.0

Programmation orientée objet en C++

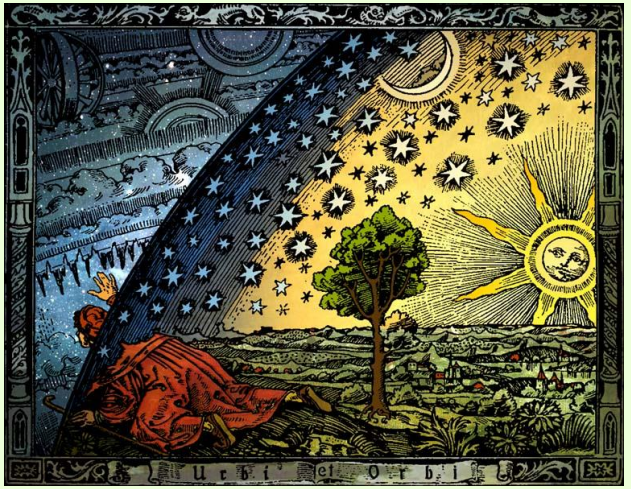
Pré-requis:

maitrise des bases algorithmiques (cf. 1ier cycle),
maitrise du C (variables, fonctions, pointeurs, structures)

Plan

Méthode Orientée Objet

- Les paradigmes « objet »
  - Encapsulation
    - Regrouper données et opérations
  - Héritage
    - Généralisation de classes
  - Polymorphisme
    - Découle de l'héritage, permet aux classes les plus générales d'utiliser les spécifications des classes plus spécifiques
  - Généricité (extension de l'encapsulation)
    - Modèle d'encapsulation, quelque soit le type des données



Universum, C. Flammarion

Définitions

- Objet
  - Instance de classe
  - Réalisation concrète d'une description abstraite
- Classe
  - Abstraction d'un concept, ne contenant que les détails nécessaires au système
  - Encapsulation des données et des opérations

Plan

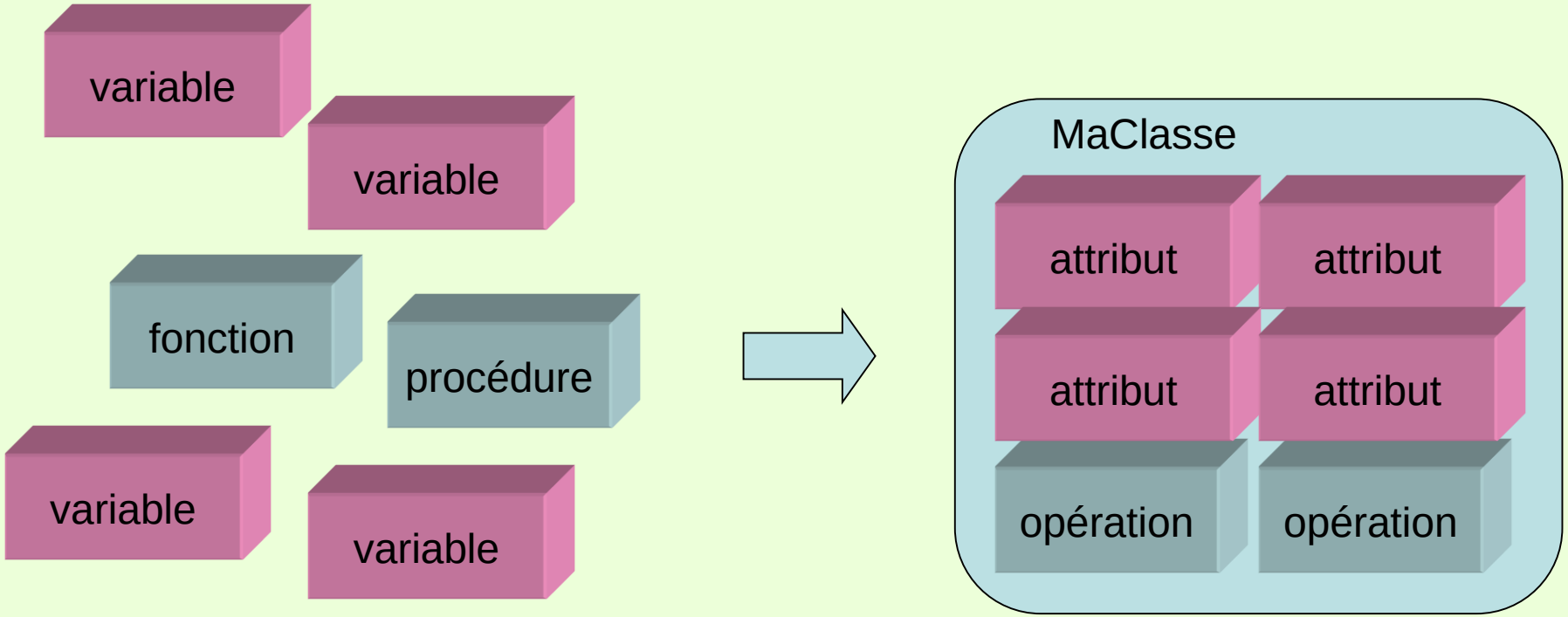
II – Encapsulation – Plan

- Rappels sur l'encapsulation
- Représentation UML et C++
  - Classes
  - Visibilité
  - Multiplicité
  - Liens entre classes
- Spécificités du C++
  - Fichiers *.h* et *.cpp*
  - Passage de paramètres :
    - Par valeur (copie), par adresse (pointeur), ***par référence***
  - Pointeur ***this***
  - Constructeurs
  - Destructeur
  - Surcharge d'opérateurs
    - Opérateur + , =

II – Encapsulation – Classes

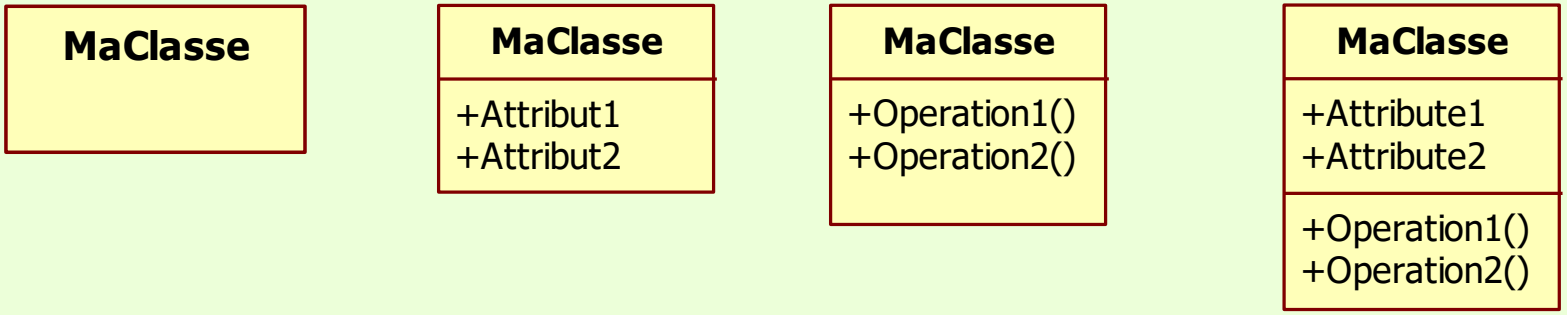
- Encapsuler = faire une classe

  = regrouper données et fonctions



II – Encapsulation – Classes

- Représentations des classes en UML



- En C++

Tout doit être déclaré !

```
class MaClasse
{
public:
type Attribut1;
type Attribut2;
void Operation1();
void Operation2();
MaClasse();
};
```

… il faudra préciser les types et les paramètres

Un constructeur n'est pas obligatoire en C++

…Premier exemple, exo 2

- Modéliser et concevoir un *PorteMonnaie* permettant de gérer une somme d'argent définie au départ. On pourra:
  - Ajouter de l'argent
  - Enlever de l'argent, à condition qu'il en reste suffisamment (signaler ce problème à l'utilisateur)
  - Afficher la somme restant dans le porte monnaie

Classe *PorteMonnaie*

```
class PorteMonnaie
{

public:
double Somme;
void Afficher()
  { cout << "Somme  =" << Somme << endl; }
void Ajouter( double argent )
  { Somme += argent; }
void Enlever( double argent )
  { if( Somme-argent >= 0)
        Somme -= argent;
    else
        cout << "Pas assez d'argent !" << endl;
  }
PorteMonnaie( double somme = 100)
  { Somme = somme;}
};
```

```
int main()
{
PorteMonnaie p(50);
p.Somme = 10;
p.Ajouter( 20 );
};
```

Ok…
Somme est *public*

…Premier exemple, exo 2

- Modéliser et concevoir un *PorteMonnaie* permettant de gérer une somme d'argent définie au départ.
  On pourra:
  - Ajouter de l'argent
  - Enlever de l'argent, à condition qu'il en reste suffisamment (signaler ce problème à l'utilisateur)
  - Afficher la somme restant dans le porte monnaie

→Il faudrait interdire l'accès direct à l'attribut gérant la somme d'argent
(être obligé de passer par les opérations *Ajouter* et *Enlever* pour le modifier)

```
void main()
{
PorteMonnaie p(50);
p.Somme = 10;
p.Somme = p.Somme - 20;
}
```

Classe *PorteMonnaie*

```
class PorteMonnaie
{
private:
        double Somme;
public:
void Afficher()
  { cout << "Somme  =" << Somme << endl; }
void Ajouter( double argent )
  { Somme += argent; }
void Enlever( double argent )
  { if( Somme-argent > 0)
        Somme -= argent;
    else
        cout << "Pas assez d'argent !" << endl;
  }
PorteMonnaie( double somme = 100)
  { Somme = somme;}
};
```

```
int main()
{
PorteMonnaie p(50);
p.Somme = 10;
p.Ajouter( 20 );
};
```

INTERDIT !!
Somme est *privée*

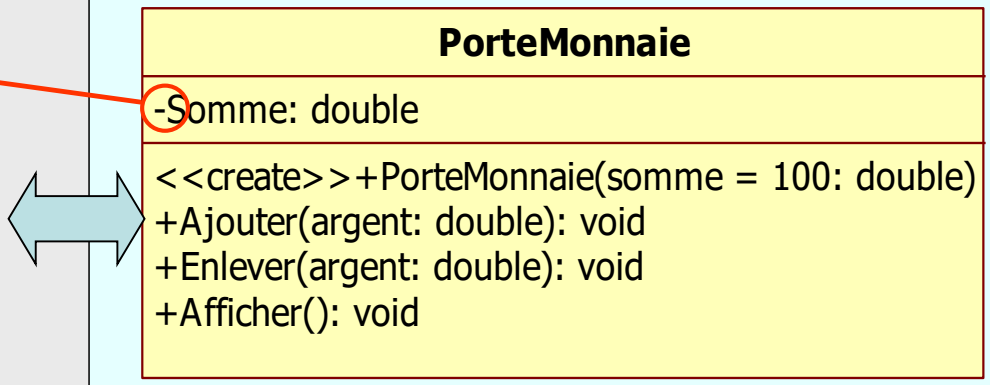
Ok, la visibilité privée autorise les **accès** aux champs pour les méthodes de la classe

Classe *PorteMonnaie*

C++

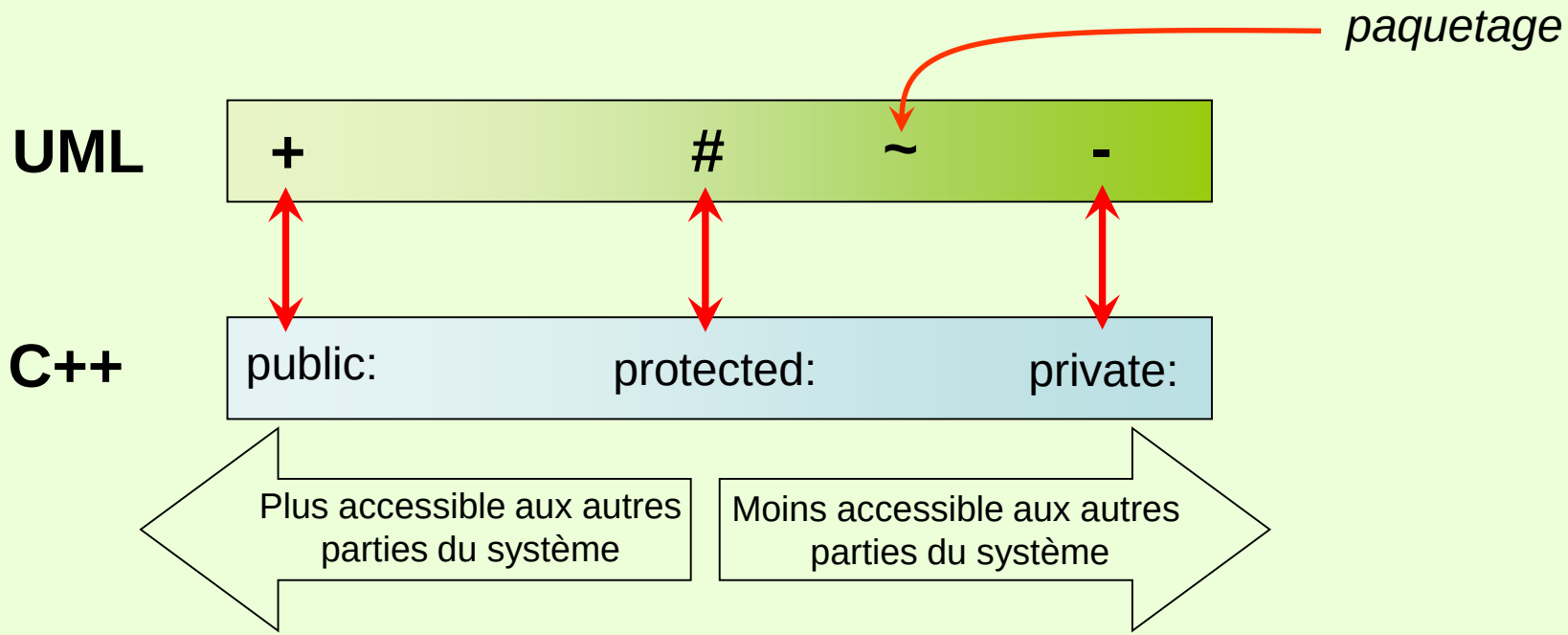
```
class PorteMonnaie
{
private:
        double Somme;
public:
void Afficher()
  { cout << "Somme  =" << Somme << endl; }
void Ajouter( double argent )
  { Somme += argent; }
void Enlever( double argent )
  { if( Somme-argent > 0)
        Somme -= argent;
    else
        cout << "Pas assez d'argent !" << endl;
  }
PorteMonnaie( double somme = 100)
  { Somme = somme;}
};
```

UML



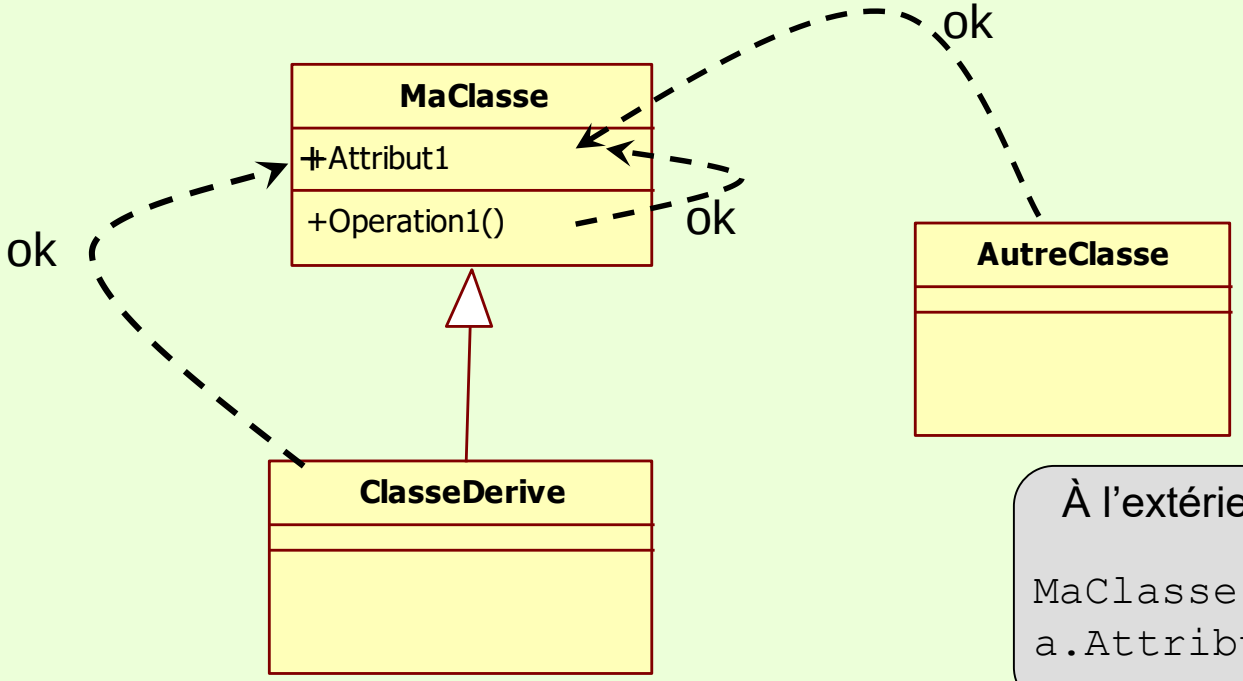
II – Encapsulation – **Visibilité**

- Visibilité des attributs et des opérations
  - 4 modificateurs en UML et 3 en C++



II – Encapsulation – Visibilité

- Visibilité *public*, (+)
  - Accessible directement par un objet
  - Accessible directement par une autre classe

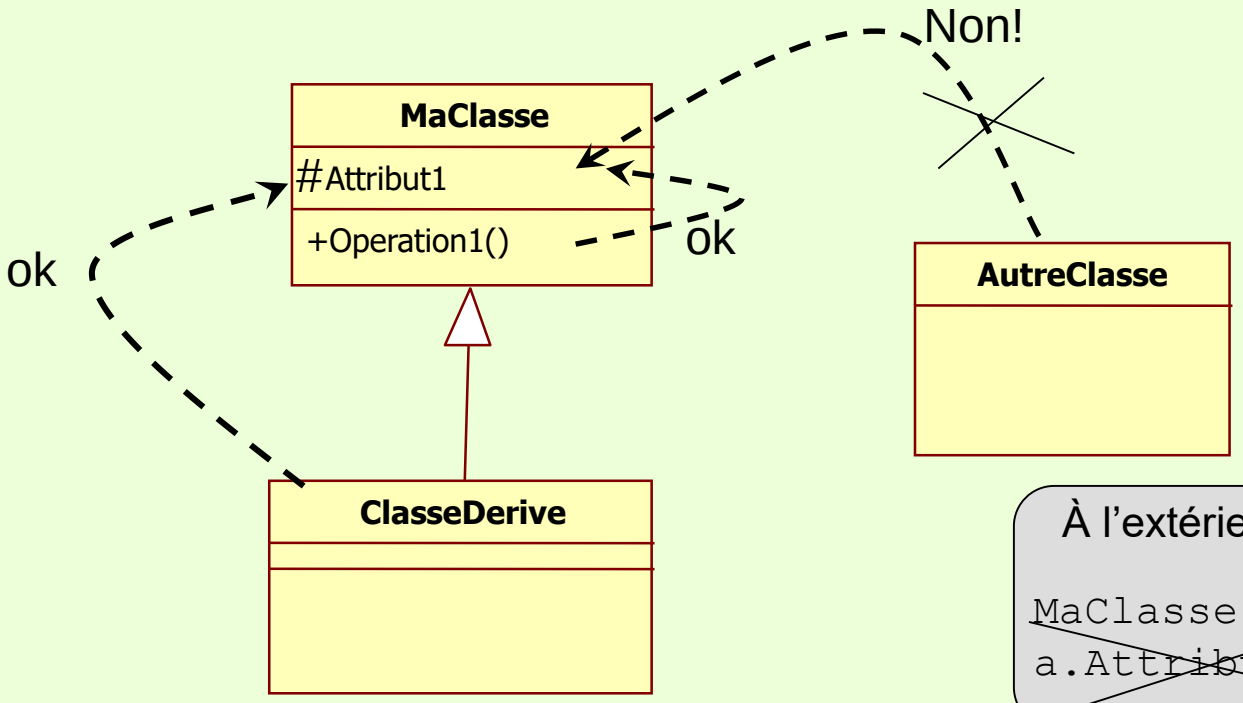


À l'extérieur de la classe

```
MaClasse a;
a.Attribut1=1;//OK
```

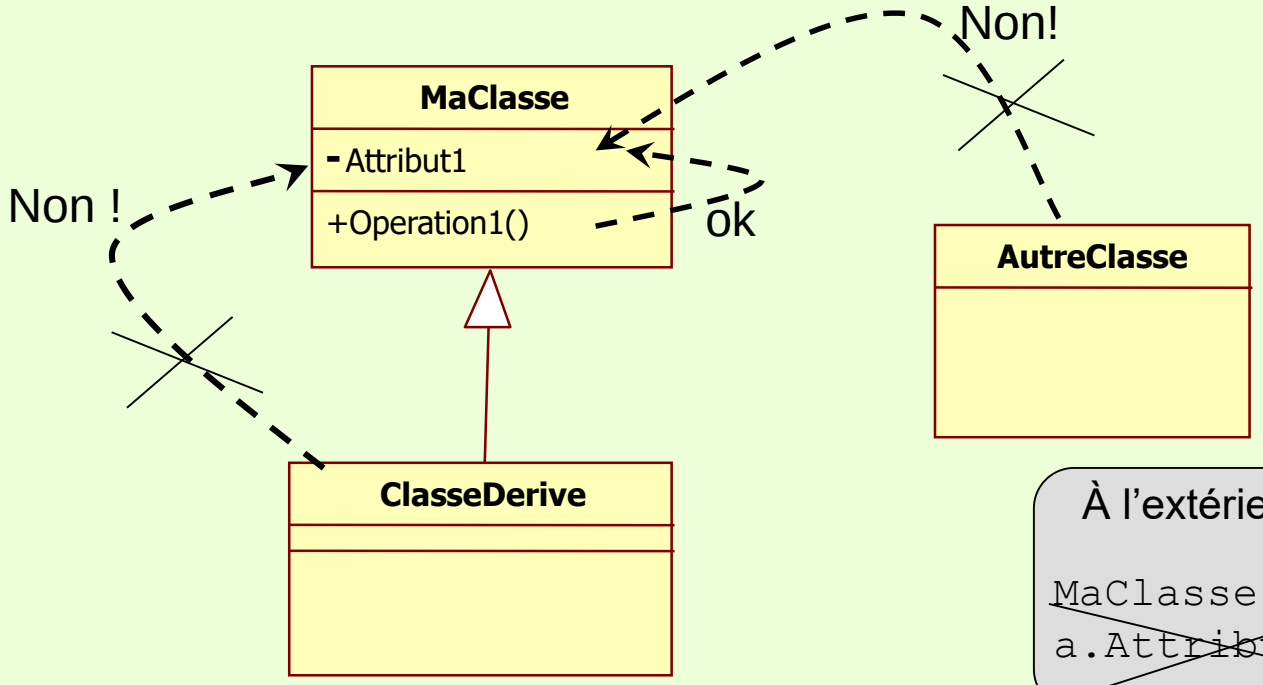
II – Encapsulation – Visibilité

- Visibilité *protected*, (#)
  - Non accessible directement par un objet
  - Accessible directement par une classe dérivée



II – Encapsulation – Visibilité

- Visibilité *private*, (-)
  - Non accessible directement par un objet
  - Non Accessible directement par une autre classe



II – Encapsulation – Visibilité

- Attributs publics…

Faut il utiliser des attributs publics ?

« Donner accès aux attributs d'une classe au reste du système ⇔ laisser la porte de sa maison ouverte et autoriser les passant à entrer sans sonner à la porte».

➔ **Usage désapprouvé par les concepteurs OO**

Mais…

➔ Il est préférable d'éviter les attributs publics,

➔ Usage autorisé pour les constantes (dialogues entre classes)

➔ Usage toléré pour les attributs n'affectant pas le comportement de la classe (ex. *Reel* et *Imag* de la classe *Complexe*)

II – Encapsulation – **Nom et type**

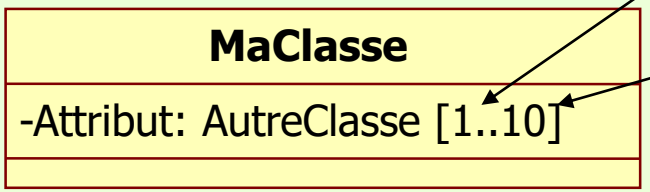
- Nom et type des attributs:
  - **Le nom** des attributs est une chaine quelconque en UML, pas en C++ (convention du langage: accents interdits, première lettre majuscule …)
    - Etre précis et clair pour le choix des noms
    - Deux attributs d'une même classe ne peuvent pas avoir le même nom
  - **Le type** des attributs est spécifié après le nom et les deux points « **:** ». Il peut s'agir :
    - D'un type classique (int, double, char, string, vector …)
    - D'une classe définie dans le système

II – Encapsulation – **Multiplicité**

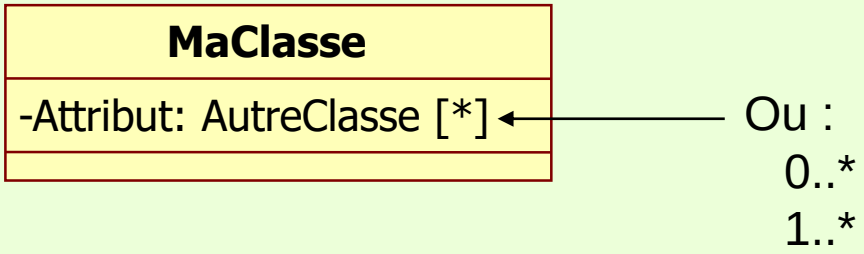
- Un attribut peut représenter un nombre quelconque d'objets de son type:

Déclaration de l'attribut comme un tableau

➔ On connaît le nombre d'éléments minimum et maximum

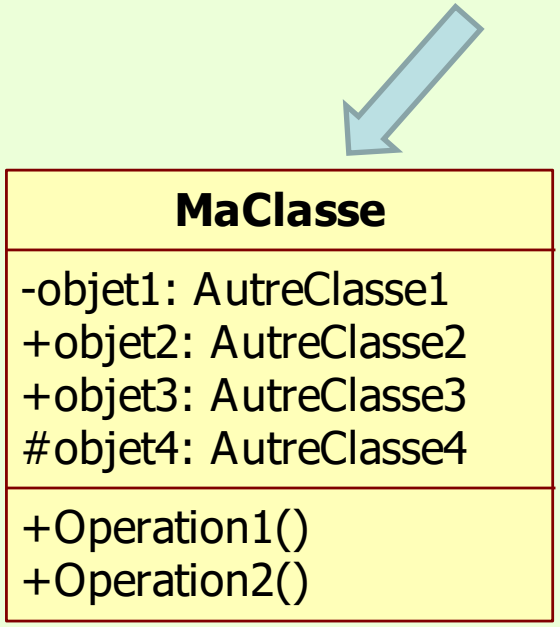


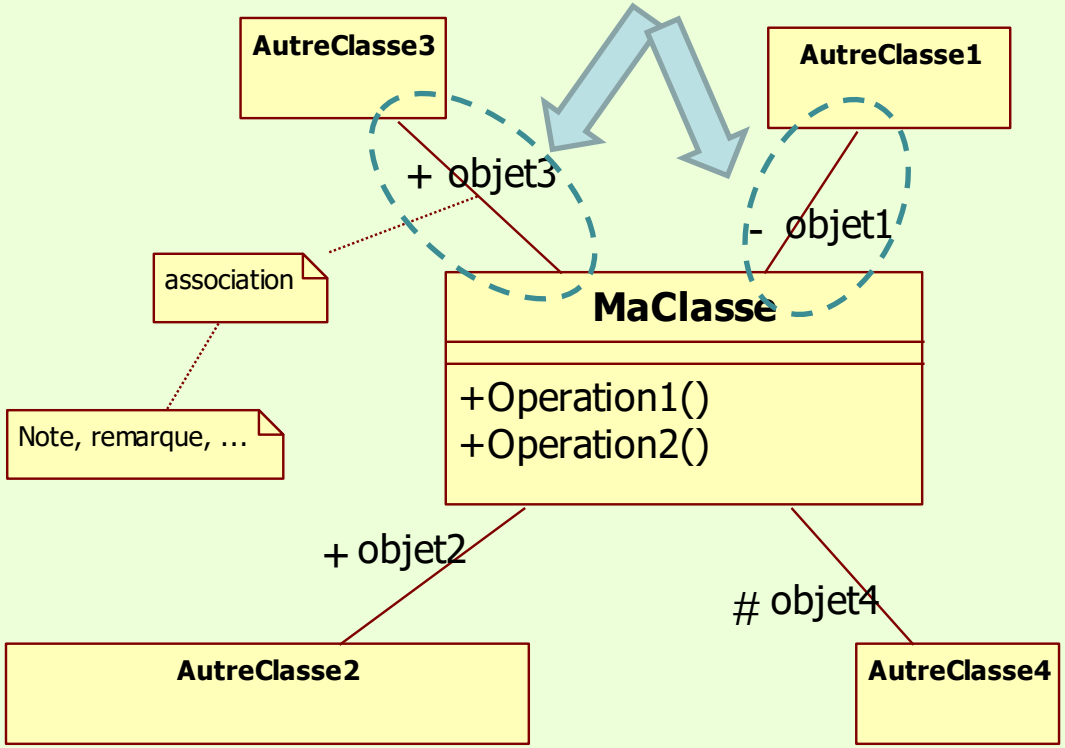
➔ On **ne** connaît **pas** le nombre d'éléments



II – Encapsulation –
Relations entre classes (UML)

- Attributs en **ligne** vs. attributs par **RELATIONS** association





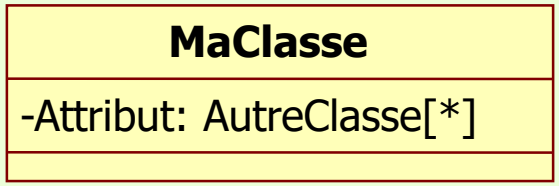
➔Représentation de la même chose (les mélanges de style sont autorisés)

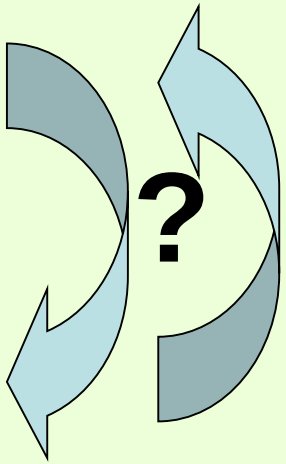
➔Attention à l'objectif du diagramme (clarté *vs.* relation entre classes)

II – Encapsulation – Relations entre classes (UML)

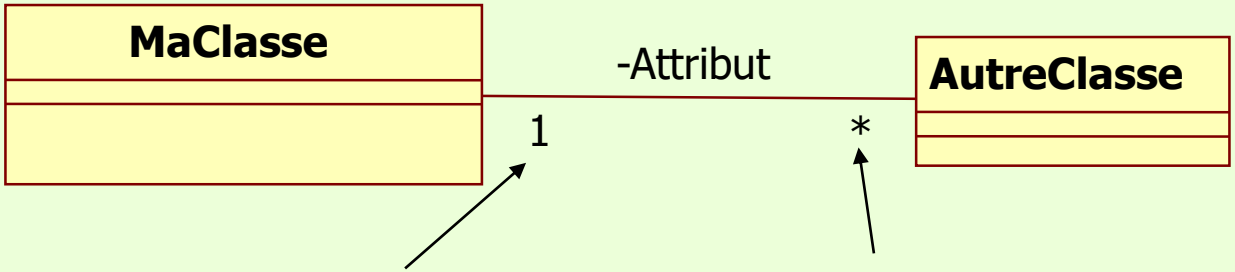
- Cas de la multiplicité

Attribut en ligne





Attribut par association



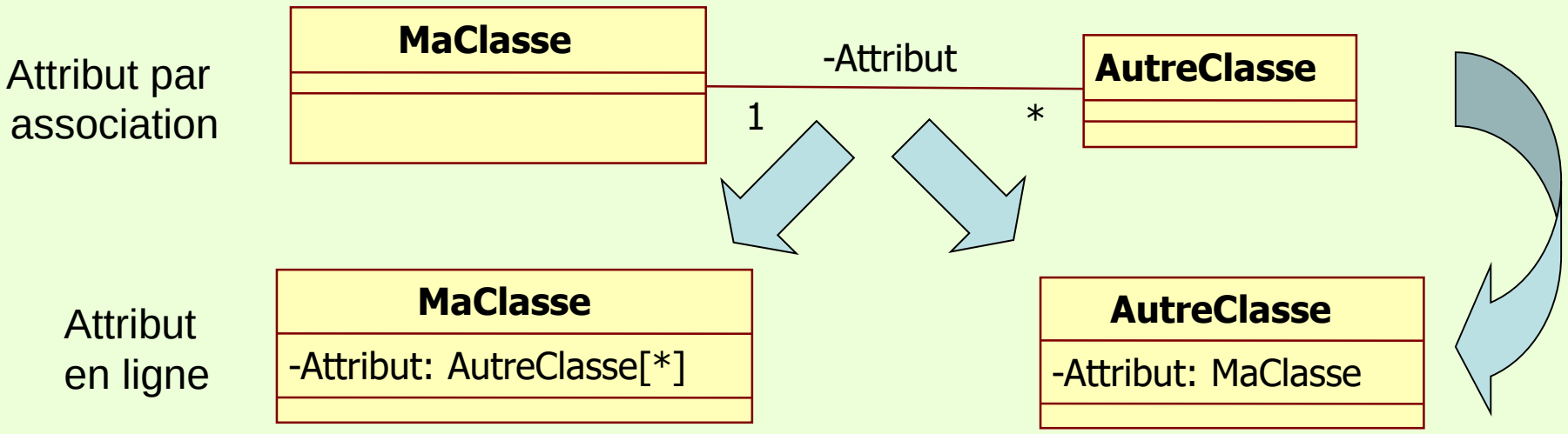
Le champs *Attribut* de *AutreClasse* est associé à **un** unique objet de type *MaClasse*

Le champs *Attribut de MaClasse* est associé à un nombre quelconque d'objets de type *AutreClasse*

➔Il y a un attribut nommé *Attribut* dans chacune des deux classes !

II – Encapsulation – Relations entre classes (UML)

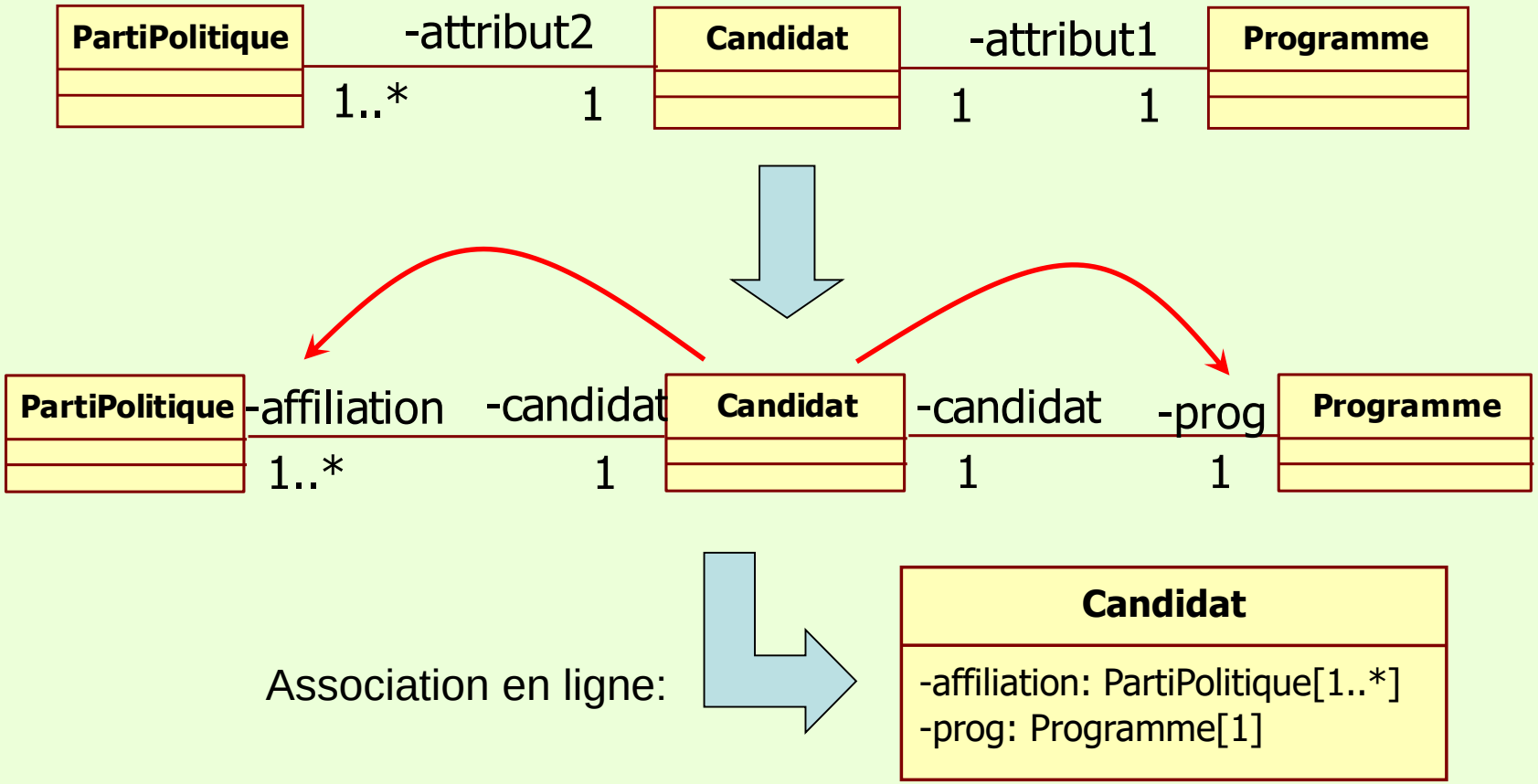
- Cas de la multiplicité



Exemple : *Modéliser les relations entre un candidat et son programme politique, puis avec les partis politiques*

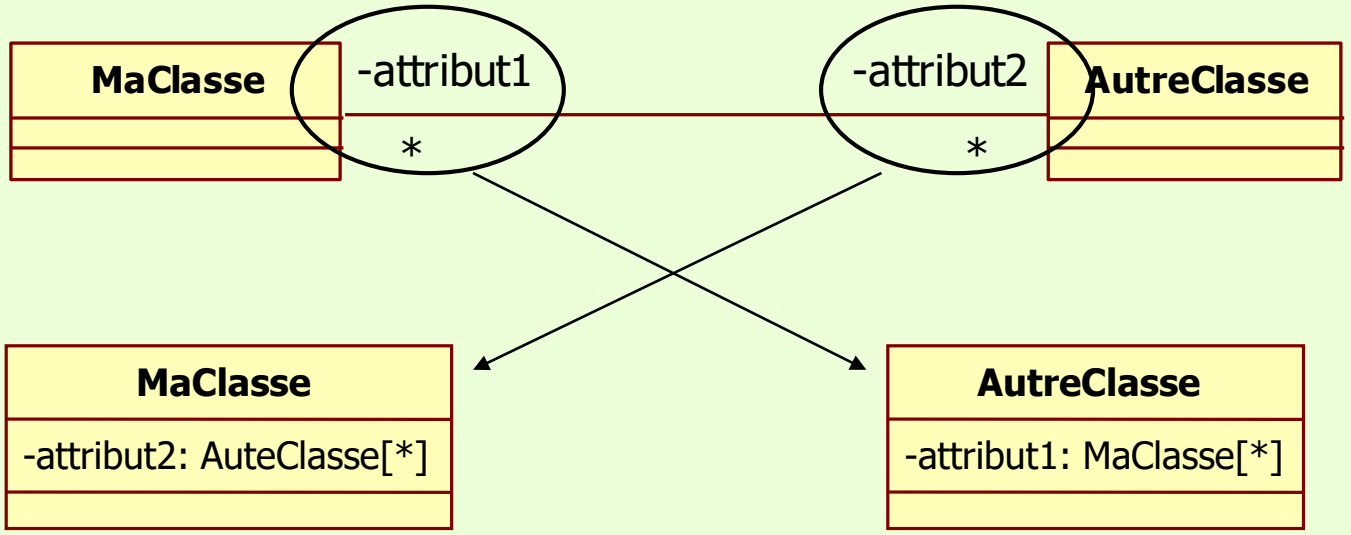
II – Encapsulation – Relations entre classes (UML)

- Exemple: Candidat, programme et partis politiques



II – Encapsulation – Relations entre classes (UML)

- Association

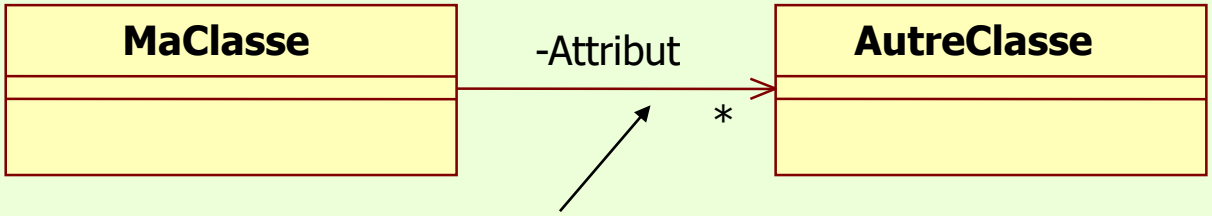


Comment rendre l'association unidirectionnelle ?
➔ rôle de la navigabilité

II – Encapsulation – Relations entre classes

- Navigabilité appliquée aux associations

UML



- *MaClasse* contient *Attribut* de type *AutreClasse*
- *AutreClasse* ne contient pas *Attribut* de type *MaClasse*

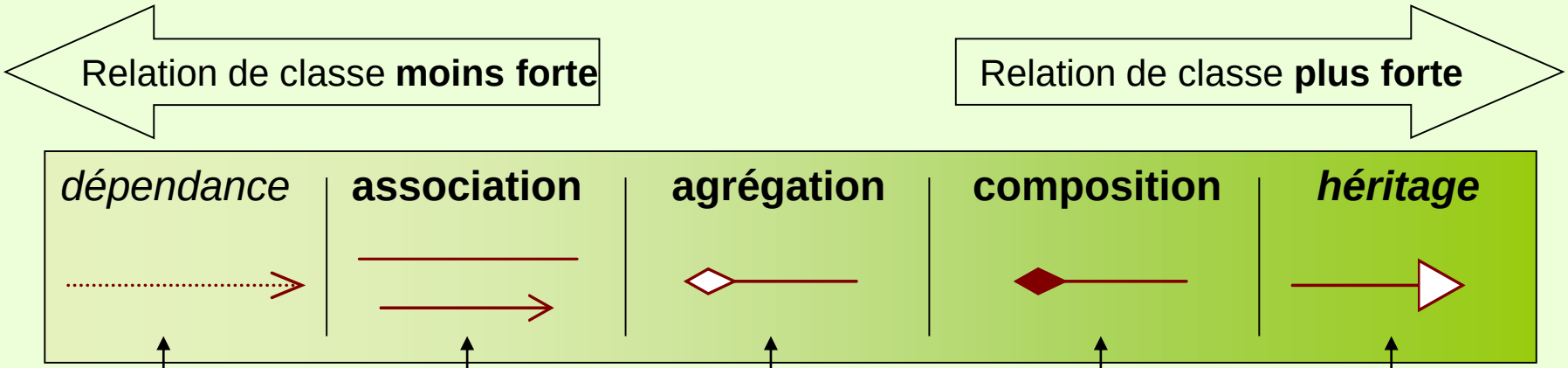
C++

```
class MaClasse
{
private:
  AutreClasse **Attribut;
…
};
```

```
class AutreClasse
{
private:
 // pas de champs Attribut
 // de type MaClasse
…
};
```

II – Encapsulation – Relations entre classes

- UML : 5 types de relations entre classe



Relation momentanée entre 2 classes
Fleche en pointillés

Une classe contient une référence à un ou plusieurs objets d'une autre classe

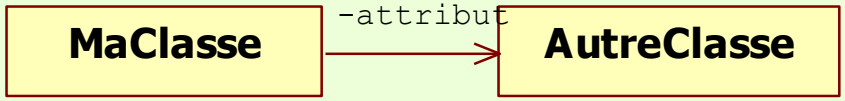
Une classe détient, et peut partager, des objets d'une autre classe

Une classe détient et ne partage pas des objets d'une autre classe

Une classe est du type d'une autre classe
(prochaine partie)

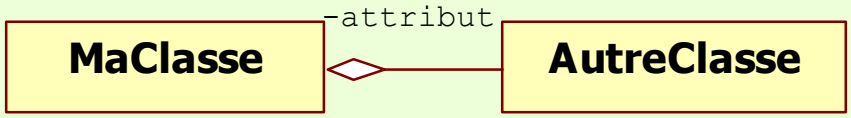
II – Encapsulation – Relations entre classes (UML)

- Association



  - *MaClasse* contient une référence à un objet de type *AutreClasse* grâce à *attribut,* elle ne modifie pas *attribut*
  - *MaClasse* **fonctionne avec un objet** de *AutreClasse*

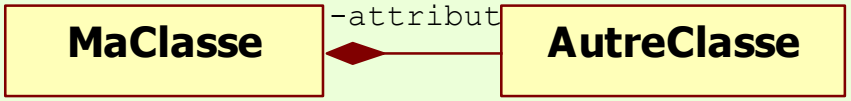
- Agrégation



  - *MaClasse* détient l'objet *attribut* et peut le partager
  - *MaClasse* est propriétaire du contenu de *attribut*

  Modification, création, suppression… d'éléments possibles

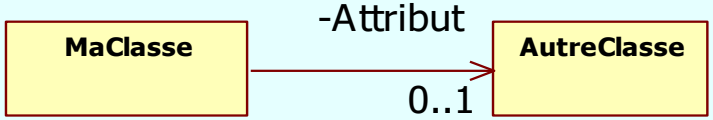
- Composition



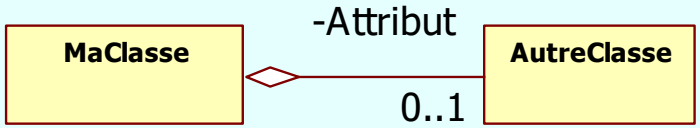
  - *MaClasse* détient et gère l'objet *attribut*: création et destruction
  - *attribut* **n'est pas partagé** avec une autre classe
  - *attribut* est une partie interne de *MaClasse*

II – Encapsulation – Les relations en C++

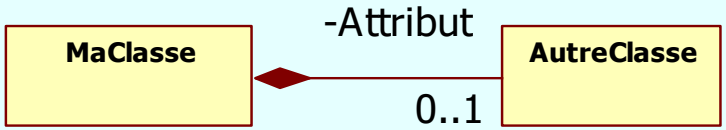
- Association



- Agrégation



- Composition

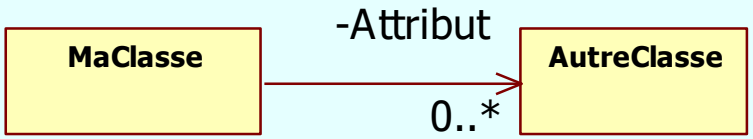



```
class MaClasse
{
private: // ou autre
  AutreClasse *Attribut;
…
};
```

```
class MaClasse
{
private: // ou autre
  AutreClasse Attribut;
…
};
```

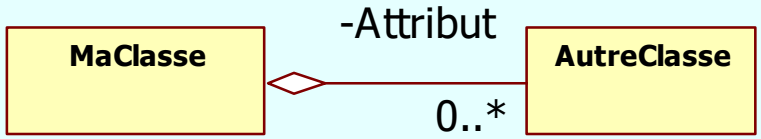
II – Encapsulation – Relations et multiplicité en C++

- Association



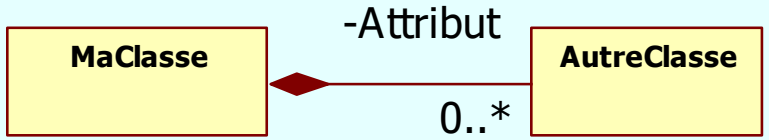

```
class MaClasse
{
private: // ou autre
  AutreClasse **Attribut;
…
};
```

- Agrégation




```
class MaClasse
{
private: // ou autre
  AutreClasse *Attribut;
…
};
```

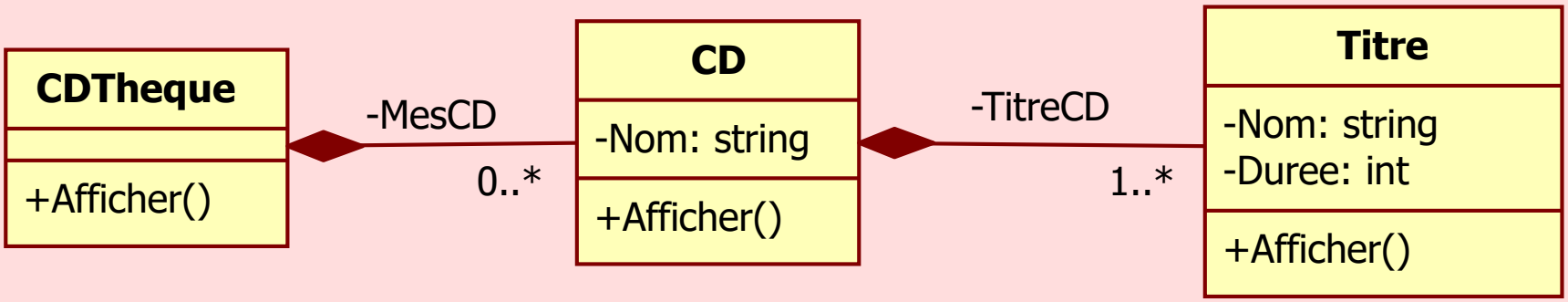
- Composition




```
class MaClasse
{
private: // ou autre
  AutreClasse Attribut[10];
…
};
```

Exemple de relations: CDtèque

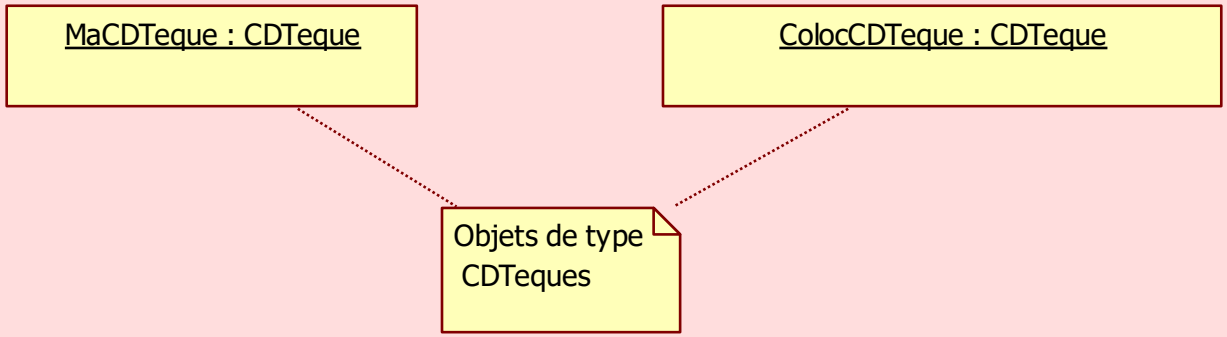
- Modéliser votre « CD thèque » étant donné que vous avez plusieurs CD et que sur chaque CD il y a plusieurs titres.
  On souhaite:
  - Afficher le nom de tous les CD de la CDthèque
  - Afficher tous les titres d'un CD et la durée de chaque titre

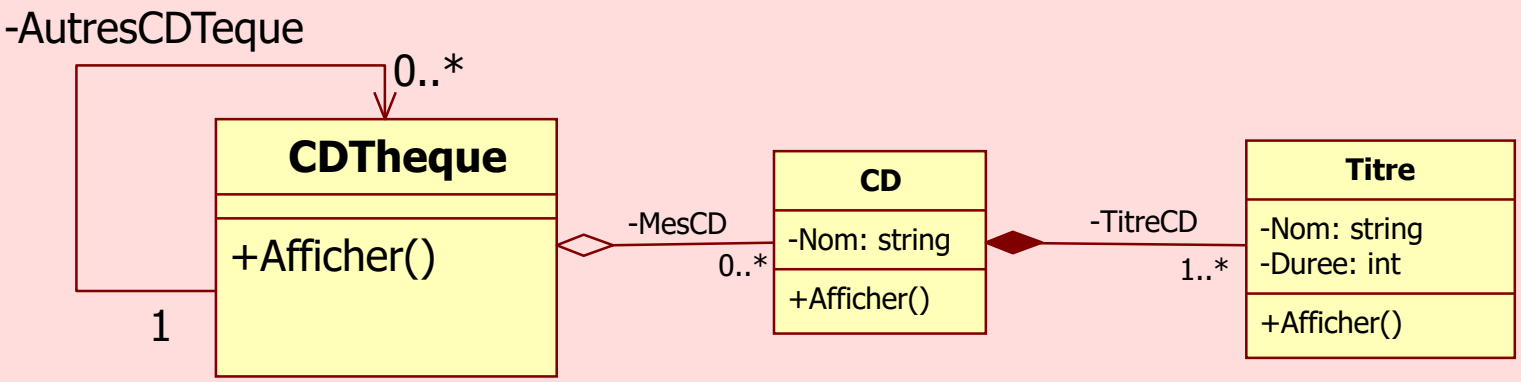


Exemple de relations: CDtèque

- Modifier votre « CD thèque » :
  vous avez un colocataire avec une autre CDthèque

Diagramme d'objets





Exemple de relations: CDtèque

- C++ de l'exemple CDteque

```
class CDTeque
{
private:
  CD *MesCD;
  CDTeque
    **AutresCDTeque;
public:
  CDTeque();
  void Afficher();
};
```

```
class CD
{
private:
  string Nom;
  Titre *TitreCD;
public:
  CD();
  void Afficher();
};
```

```
class Titre
{
private:
  string Nom;
  int Duree;
public:
  Titre();
  void Afficher();
};
```

Que manque t'il à ces classes pour pouvoir fonctionner ?

Le nombre d'éléments des tableaux !

Des méthodes d'ajout/suppression/modification d'éléments CDTeque, CD et Titre

À faire … ou utiliser ***vector<>***

II – Encapsulation – C++

- En C++, pour la lisibilité, séparation de:
  - La définition de la classe ➔ ***.h***
  - L'implémentation des méthodes ➔ ***.cpp***

.h

```
class Complexe
{
public:
   double Reel;
   double Imag;

   Complexe();
   Complexe Plus(Complexe z);
};
```

.cpp

opérateur d'appartenance

```
Complexe::Complexe()
   {
   Reel=0;
   Imag=0;
   }
Complexe Complexe::Plus(Complexe z)
   {
    Complexe s;
     s.Reel = Reel + z.Reel;
     s.Imag = Imag + z.Imag;
    return s;
   }
```

II – Encapsulation – C++

Complexe.h

```
#ifndef _Complexe_h_
#define _Complexe_h_

#include …… //si besoin

class Complexe
{
public:
  double Reel;
  double Imag;

  Complexe();
  Complexe Plus(Complexe z);
};
#endif
```

Complexe.cpp

```
#include "Complexe.h"

#include …… //si besoin

Complexe::Complexe()
  {
  Reel=0;
  Imag=0;
  }
Complexe Complexe::Plus(Complexe z)
  {
   Complexe s;
    s.Reel = Reel + z.Reel;
    s.Imag = Imag + z.Imag;
   return s;
  }
```

Classe complexe

Complexe.h

```
#ifndef _Complexe_h_
#define _Complexe_h_

class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe();
  Complexe Plus(Complexe z);
  Complexe MuliplierPar
        (Complexe z);
  Complexe DiviserPar
        (Complexe z);
  Complexe Conjuger();
  void Affiche();
  void AffichePolar();
};
#endif
```

Complexe.cpp

```
#include "Complexe.h"
#include <cmath>

Complexe::Complexe() { … }
Complexe Complexe::Conjuguer() {… }
Complexe Complexe::Plus(Complexe z) {… }
void Complexe::Affiche() {…}
Complexe Complexe::MultiplierPar(Complexe
z) {… }

Complexe Complexe::DiviserPar(Complexe z)
{
Complexe s;
s = MultiplierPar( z.Conjuguer() );
s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
return s;
}
```

Classe complexe

Complexe.h

```
#ifndef _Complexe_h_
#define _Complexe_h_

class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe();
  Complexe Plus(Complexe z);
  Complexe MuliplierPar
        (Complexe z);
  Complexe DiviserPar
        (Complexe z);
  Complexe Conjuger();
  void Affiche();
  void AffichePolar();
};
#endif
```

Complexe.cpp

```
#include "Complexe.h"
#include <cmath>
 …
Complexe Complexe::DiviserPar(Complexe z)
{…}

void Complexe::AffichePolar()
{
cout << sqrt(Reel * Reel + Imag * Imag);
cout << "ei(" <<  atan(Imag / Reel);
cout  << endl;
}
```

C++ - passage de paramètres

- Passage de paramètres par valeur (copie)
  - Idem que langage C
- Passage de paramètres par adresse (pointeur)
  - Idem que langage C
- Passage de paramètres par référence
  - Nouveau !

C++ - passage de paramètres

- Passage de paramètres par valeur (copie)

```
void fonction(double a)
{
  a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(x);
  cout << x;
}
```

→ x vaut 1

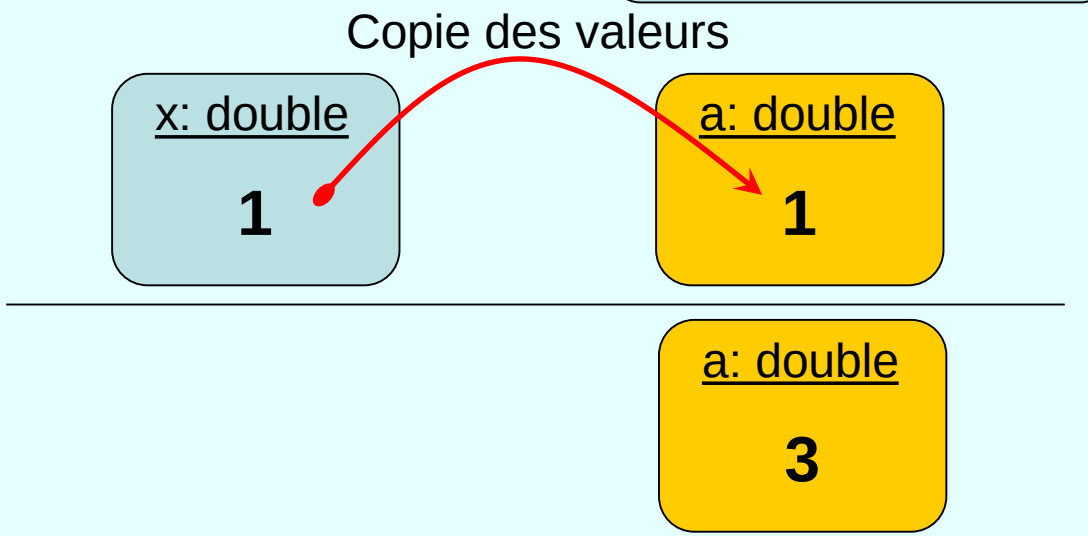
En mémoire :

A l'appel de **`fonction(x);`**

⇩

Dans *fonction*:

`a = 3;`



➔ Pas de modification de la valeur de x dans la fonction

C++ - passage de paramètres

- Passage de paramètres par adresse (pointeur)

```
void fonction(double *a)
{
  *a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(&x);
  cout << x;
}
```

→ x vaut 3

En mémoire :

Copie de l'adresse de x ⇔ &x

&x

x: double

1

a: double*

&x

A l'appel de **`fonction(&x);`**

⇩

Dans *fonction*:

`*a = 3;`

x: double

3

a: double*

&x

Mettre 3 dans ce que pointe a

➔ Modification de la valeur de x par la fonction

C++ - passage de paramètres

- Passage de paramètres par référence

```
void fonction(double &a)
{
  a = 3;
}
```

```
void main(void)
{
  double x = 1;
  fonction(x);
  cout << x;
}
```

→ x vaut 3

En mémoire :

extraction de l'adresse de x

&x

x: double

1

a: double&

&x

a est une référence de x

A l'appel de **`fonction(x);`**

⇩

Dans *fonction*:

`a = 3;`

x: double

3

a: double&

&x

Mettre 3 dans ce que référence a

➔ Modification de la valeur de x par la fonction

C++ - passage de paramètres

- Passage de paramètres par valeur (copie)
  - **Copie** des valeurs des variables dans les paramètres
    2 espaces mémoires différents

- Passage de paramètres par adresse (pointeur)
  - Copie de l'adresse d'une variable (pas des valeurs)
  - Modification de la syntaxe pour l'accès au contenu
  - Permet de passer et manipuler des tableaux

- Passage de paramètres par référence
  - Extraction de l'adresse d'une variable
    - création d'une référence, pas de copie des valeurs
  - Transparent à l'utilisation et à la programmation d'une fonction (pas de changement de syntaxe)
    - Simplement préciser les paramètres passés par référence
  - Ne permet pas de passer ni de manipuler des tableaux

C++ - Passage de paramètres

- Comparaisons

	copie	**pointeur**	**référence**
rapidité	lent	rapide	rapide
utilisation	très simple	attention aux règles	simple
manipulation de tableau	non	oui	non
mémoire utilisée/param.	un objet	une adresse	une référence
modification paramètres	non	oui	oui
polymorphisme	*non*	*oui*	*oui*

C++ - mot clé « const »

- But du mot clé: préciser et garantir l'interdiction de modification d'un objet
  - Applicable aux paramètres (pointeur et référence)
  - Applicable aux opérations des classes

```
void fonction(const double &a)
{
  a = 3;
}
```

Erreur de compilation :
la référence « a » est constante
et ne peut pas être modifiée

```
void Complexe::AffichePolar() const
{ … }
```

Cette opération ne modifie pas les attributs de l'objet courant

Classe complexe

Complexe.h

Version avec référence et mot clé *const*

```
#ifndef _Complexe_h_
#define _Complexe_h_

class Complexe
{
public:
  double Reel;
  double Imag;
  Complexe();
  Complexe(const double &re, const double &im);
  Complexe Plus(const Complexe &z) const;
  Complexe MuliplierPar(const Complexe &z) const;
  Complexe DiviserPar(const Complexe &z) const;
  Complexe Conjuger() const;
  void Affiche() const;
  void AffichePolar() const;
};
#endif
```

Classe complexe

Complexe.cpp

Version avec référence et mot clé *const*

```
#include "Complexe.h"
#include <cmath>
…
Complexe Complexe::DiviserPar(const Complexe &z) const
{
Complexe s;
s = MultiplierPar( z.Conjuguer() );
s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
return s;
}

void Complexe::AffichePolar() const
{
cout << sqrt(Reel * Reel + Imag * Imag);
cout << "ei(" <<  atan(Imag / Reel);
cout  << endl;
}
```

Doit être déclarée *const*

Classe complexe

Complexe.cpp

Version avec référence et mot clé *const*

```
#include "Complexe.h"
#include <cmath>
…
Complexe Complexe::DiviserPar(const Complexe &z) const
{
Complexe s;
s = MultiplierPar( z.Conjuguer() );
s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
return s;
}

void Complexe::AffichePolar() const
{
cout << sqrt(Reel * Reel + Imag * Imag);
cout << "ei(" <<  atan(Imag / Reel);
cout  << endl;
}
```

Doit être déclarée *const*

C++ : pointeur *this*

- *this* est un **pointeur privé** automatiquement inclus dans les classes (il n'apparaît pas dans la définition de la classe)
- *this* pointe toujours l'objet courant
  - pointeur du type de la classe (ex.: « Complexe * »)
  - initialisation automatique sur l'objet courant (ou appelant)

```
Complexe::Complexe()
{
  this->Reel=0; // ⇔ Reel=0;
  this->Imag=0; // ⇔ Imag = 0;
}
Complexe Complexe::DiviserPar(const Complexe &z) const
{
  Complexe s;
  s = this->MultiplierPar( z.Conjuguer() );
  s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
  s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
  return s;
}
```

this permet d'accéder à **tous** les éléments de l'objet courant

Rappel : (*this).Reel ⇔ this->Reel

C++ : constructeurs

- En C++, un constructeur :
  - est nécessaire à la classe (un constructeur minimum par classe, un constructeur *implicite* sans paramètre est ajouté si aucun n'est présent, un constructeur de copie est aussi toujours présent)
  - porte le nom de la classe
  - ne retourne rien de rien (même pas void !)
  - est surchargeable : Une classe peut avoir plusieurs constructeurs,

    Le type et le nombre des paramètres passés lors de la construction d'un objet permettent de désigner sans ambiguïté le constructeur à exécuter
  - est exécuté automatiquement et uniquement à la création d'un objet :

```
MaClasse a; // Execution du constructeur de MaClasse
```

NB: un constructeur n'est pas forcement public (ex. des architectures basées sur les pointeurs objets)

C++ : constructeurs

- Rôle du constructeur :
  - Doit permettre d'initialiser l'objet créé de façon à le rendre utilisable par le système. En clair initialise (tous) les champs de l'objet.
  - (le compilateur se charge de la création en mémoire des champs statiques)
- Remarque : on peut passer des instructions spécifiques lors de la création des champs statiques

```
class Complexe
{public:
  double Reel;
  double Imag;
  Complexe();
   …
};
```

Complexe.cpp

```
Complexe::Complexe():Reel(0), Imag(0){ }
```

Reel et Imag seront initialisés à 0

Attention à l'ordre !

C++ : constructeurs

- Les constructeurs du C++ :
  - Constructeur sans paramètres (ou « par défaut »)
    - Il est conseillé d'en faire un pour les définitions d'objet :

      MaClasse a;
    - Déclaration : `MaClasse();`
  - Constructeur(s) « utilisateur »
    - Autant que nécessaire, attention aux ambiguïtés d'exécution
    - **Surcharge le constructeur par défaut implicite**
    - Déclaration (exemples) :
      - `MaClasse( int a );`
      - `MaClasse( double x1, double x2, double x3=0);`
      - …
  - Constructeur de copie
    - S'il n'existe pas, automatiquement réalisé par le compilateur (copie des valeurs de tous les champs, *pb* avec les pointeurs…)
    - Déclaration : `MaClasse( const MaClasse &a);`

C++ : destructeur

- En C++, un destructeur :
  - n'est pas obligatoire
    - Si la classe n'en possède pas, le compilateur en ajoute un « simple » (donc problème si allocations d'attributs dynamiques)
    - Obligatoire dans le cas de gestion de l'allocation d'attributs dynamiques (pointeurs, tableaux dynamiques, listes, …)
  - porte le nom de la classe précédé de ~, ex: `~MaClasse();`
  - ne retourne rien de rien (même pas void)
  - n'a pas de paramètre
  - est généralement *public*
  - est **unique**
  - est exécuté automatiquement à la destruction des objets
  - peut être appelé explicitement

    delete a; delete[] a; ~~a.~MaClasse();~~

    Pas recommandé…

NB: un destructeur n'est pas forcement public (cf. t-49) mais devrait toujours être virtuel

C++ - opérateurs

- Il est possible de surcharger les opérateurs du C++ : =, +, -, /, (), [], ==, +=, …

possibilité introduite dans Algol68

  - Le nom de la fonction porte le nom ***operator*** suivi du symbole ou mot clé à (re)définir

```
Complexe Complexe::operator/(const Complexe &z) const
{
  Complexe s;
  s = this->MultiplierPar( z.Conjuguer() );
  s.Reel /= z.Reel*z.Reel + z.Imag*z.Imag;
  s.Imag /= z.Reel*z.Reel + z.Imag*z.Imag;
  return s;
}
```

```
void main(void)
{
Complexe a(1,1);
Complexe b(1,-4);
Complexe z;
z = a / b;
}
```

Par défaut un *operator*= est ajouté à toutes les classes

➔ à surcharger en cas d'attribut dynamique

C++ - opérateur =

- La surcharge de l'opérateur égal '=' est obligatoire si la classe gère elle-même des espaces mémoires (allocation et libération)
- La déclaration de *operator*= respecte une syntaxe stricte:

```
Complexe& Complexe::operator=(const Complexe &z);
```

  - La déclaration permet d'écrire : a=b=c=d;
  - L'implémentation doit permettre de gérer : a=a; …
- L'implémentation de *operator*= respecte un canevas précis:

```
Tableau& Tableau ::operator=(const Tableau &t)
{ if( &t != this) // « t » est il l'objet courant?
    {
    //allocation et copie des éléments de t dans l'objet courant
    }
  return *this;  // on retourne le contenu de l'objet courant
}
```

Plan

I. Premier exemple

II. Définitions de l'Orienté Objet

III. Encapsulation, concept de classe

IV. Héritage (généralisation) *… troisième partie …*

V. Polymorphisme

VI. Généricité (modèles de classe)

Exercice

- Modéliser les différents effectifs du département GE. Pour cela, on modélisera les différents constituants ainsi :
  - un étudiant par: nom, prénom, promo, mail, groupe
  - un enseignant par: nom, prénom, mail, matière
  - une secrétaire par: nom, prénom, mail, fonction
  - un technicien par: nom, prénom, mail, spécialité