Introduction aux méthodes Orientées Objets

Troisième 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

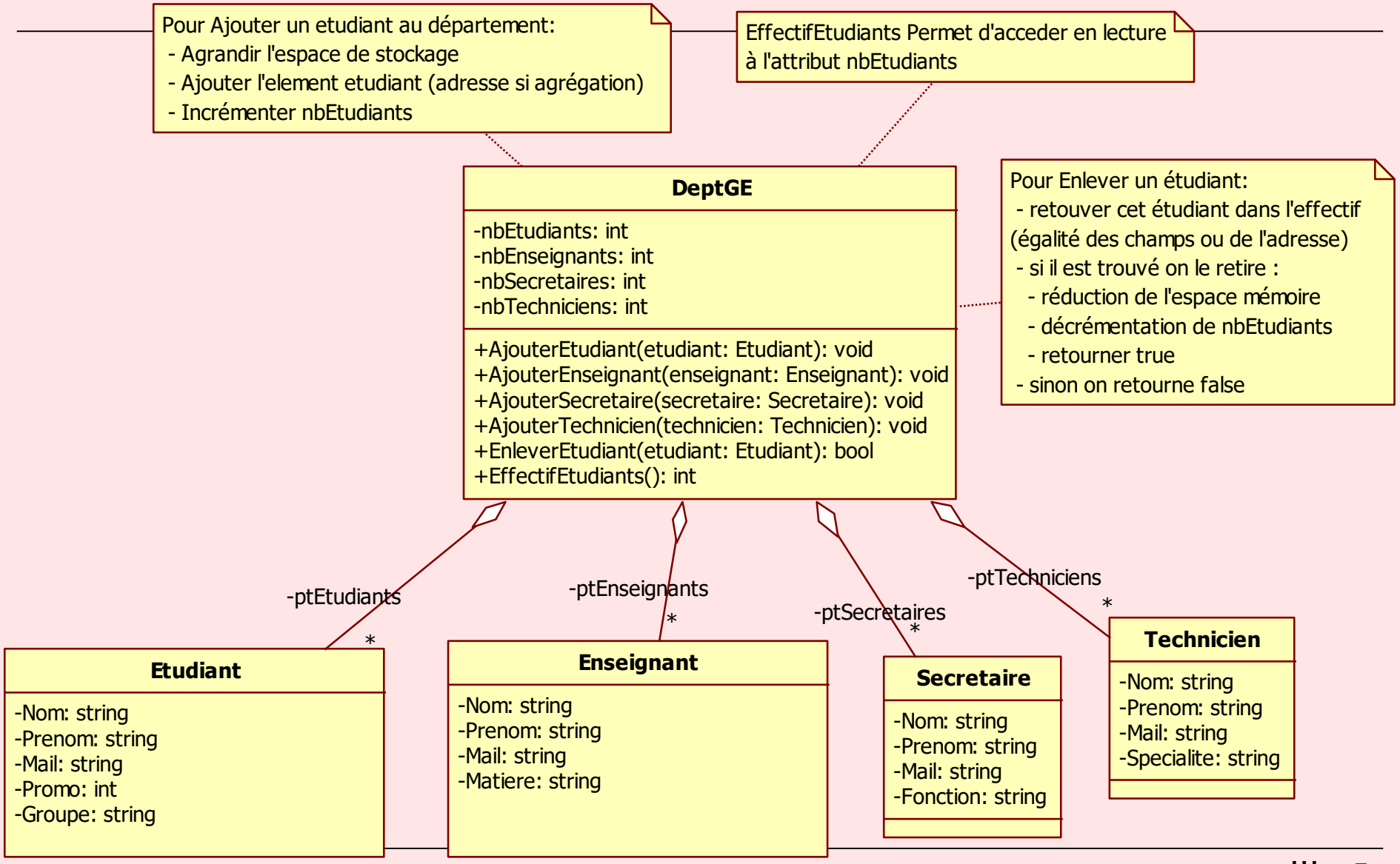
IV – Héritage – Plan

- Exemple sur exercice DeptGE
- Définitions
  - UML et C++
- Conventions C++
  - Ordre d'initialisation des objets
  - Ordre de destruction des objets

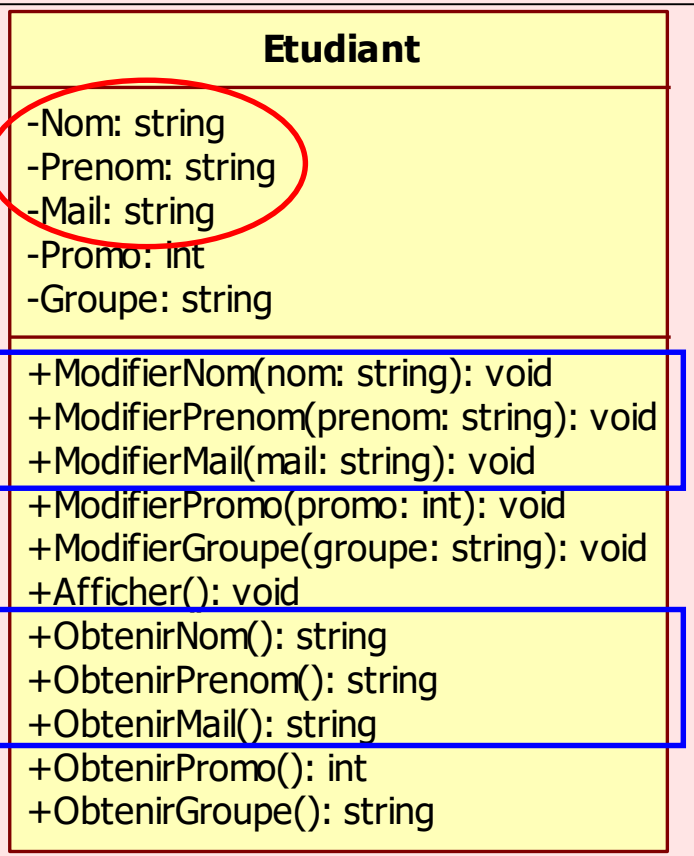
Exercice DeptGE

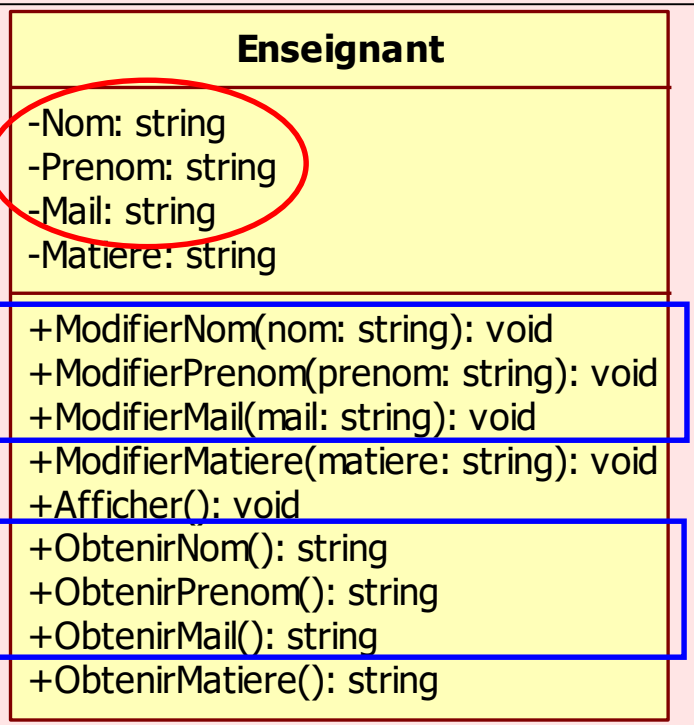
- Modéliser les différents effectifs du département GE. Pour cela, on modélisera les différents constituants ainsi :
  - un étudiant par: nom, prénom, promo, mail, groupe
  - un enseignant par: nom, prénom, mail, matière
  - une secrétaire par: nom, prénom, mail, fonction
  - un technicien par: nom,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

Exercice DeptGE



Exercice DeptGE





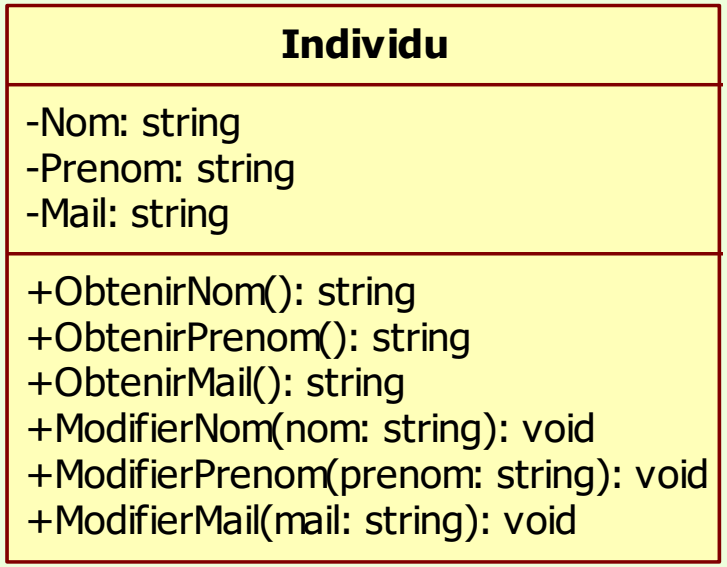
...

La même chose avec les classes:
- Secretaire
- Technicien

➔ Intérêts de regrouper les attributs et opérations identiques ? Comment ?

IV – Héritage (généralisation)

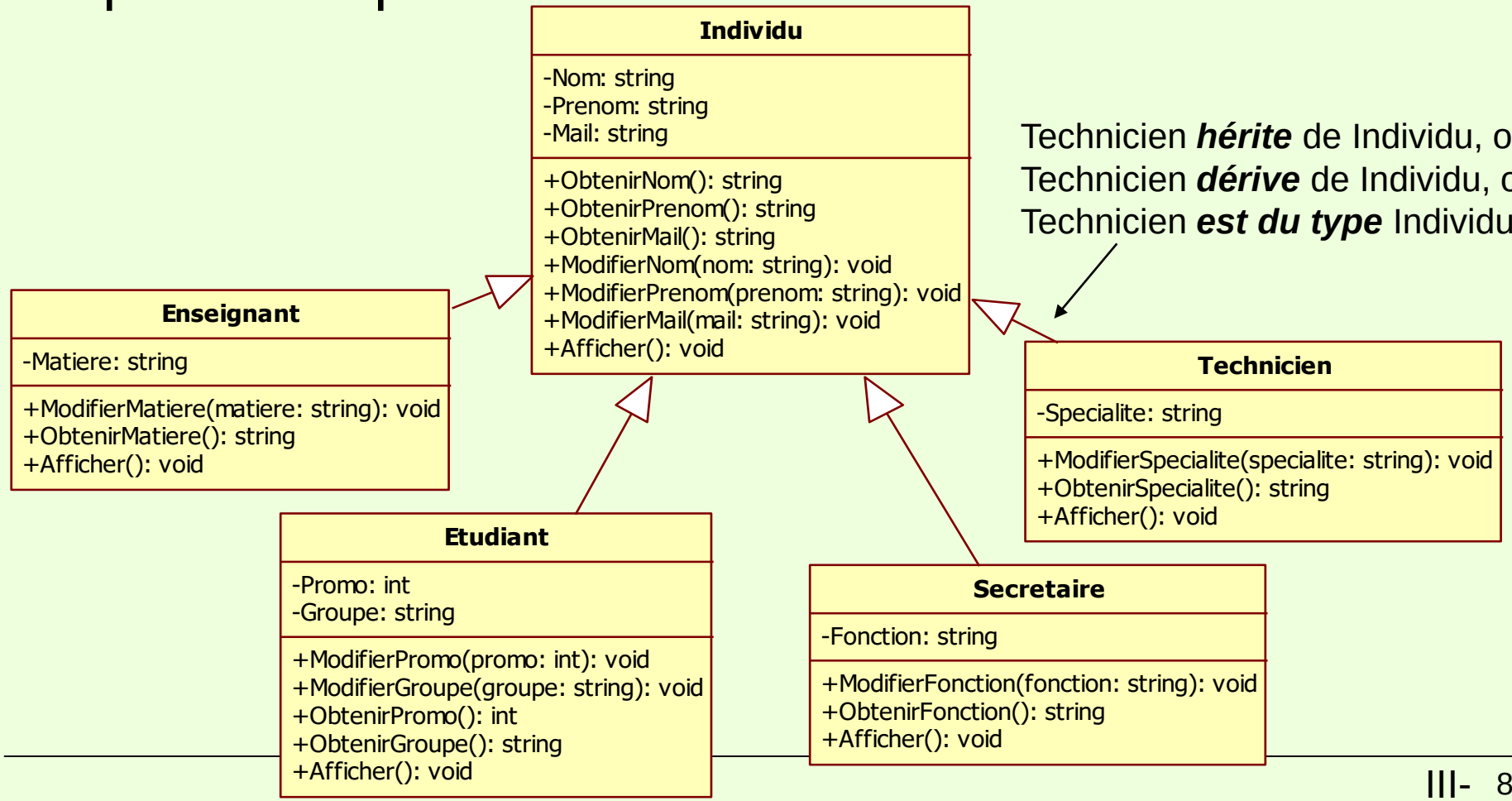
- Solution: faire une classe **plus générale*** (`Individu`) et <u>inclure</u> les fonctionnalités de cette classe dans les classes plus spécifiques:
  `Etudiant, Enseignant, Secretaire…`



* : ou « moins spécifique »

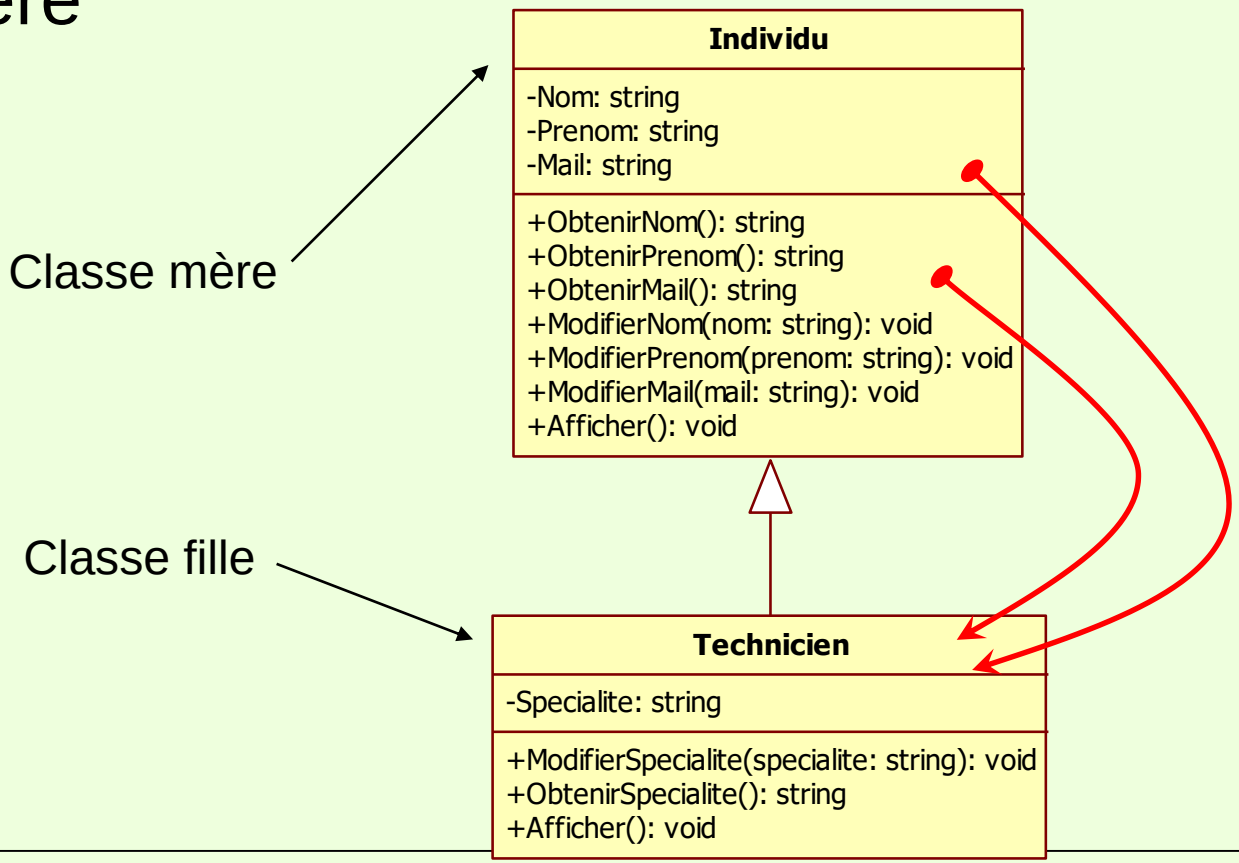
IV – Héritage (généralisation)

- La relation d'héritage (⟶) est la relation la plus forte pour lier deux classes



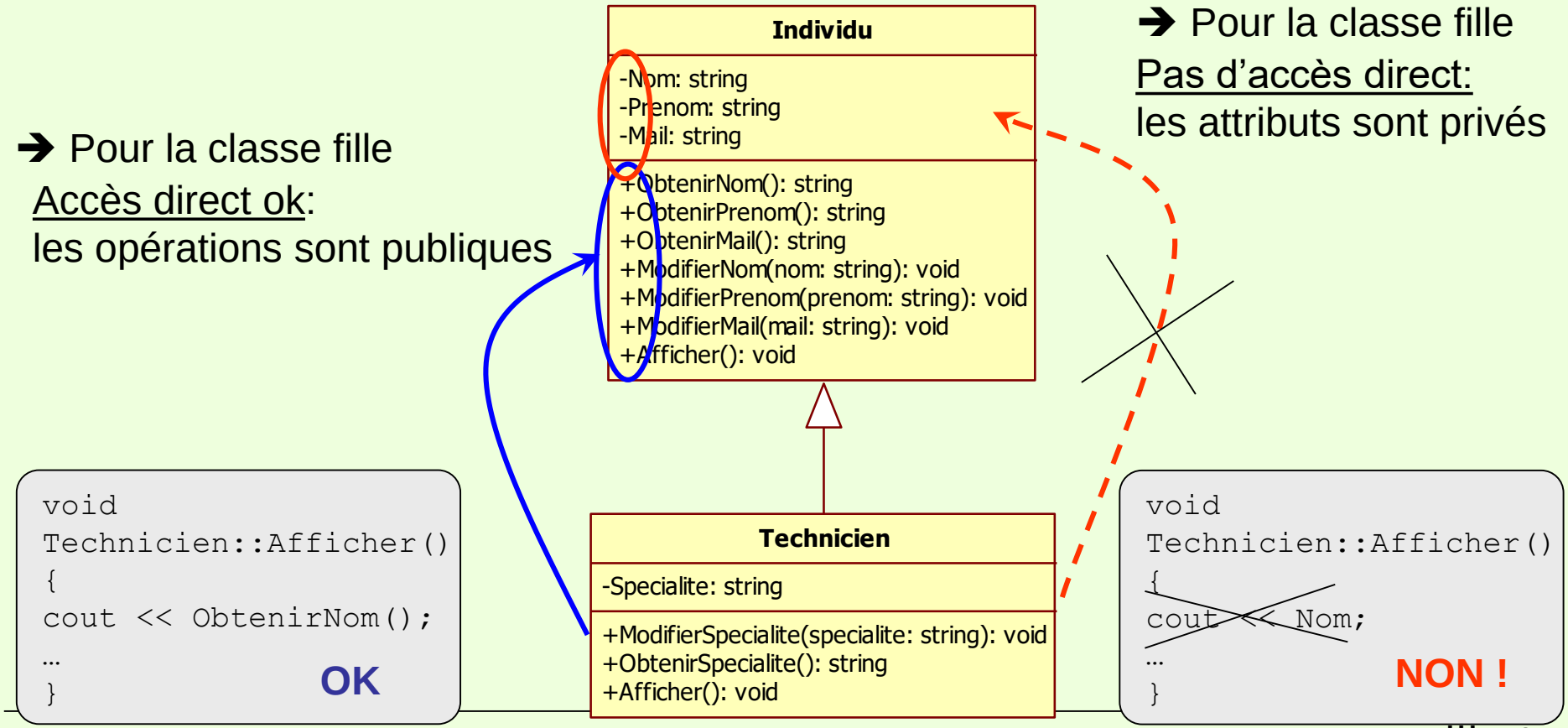
IV – Héritage (généralisation)

- Lorsqu'une classe hérite d'une autre classe, elle possède tous les attributs et opérations de la classe mère



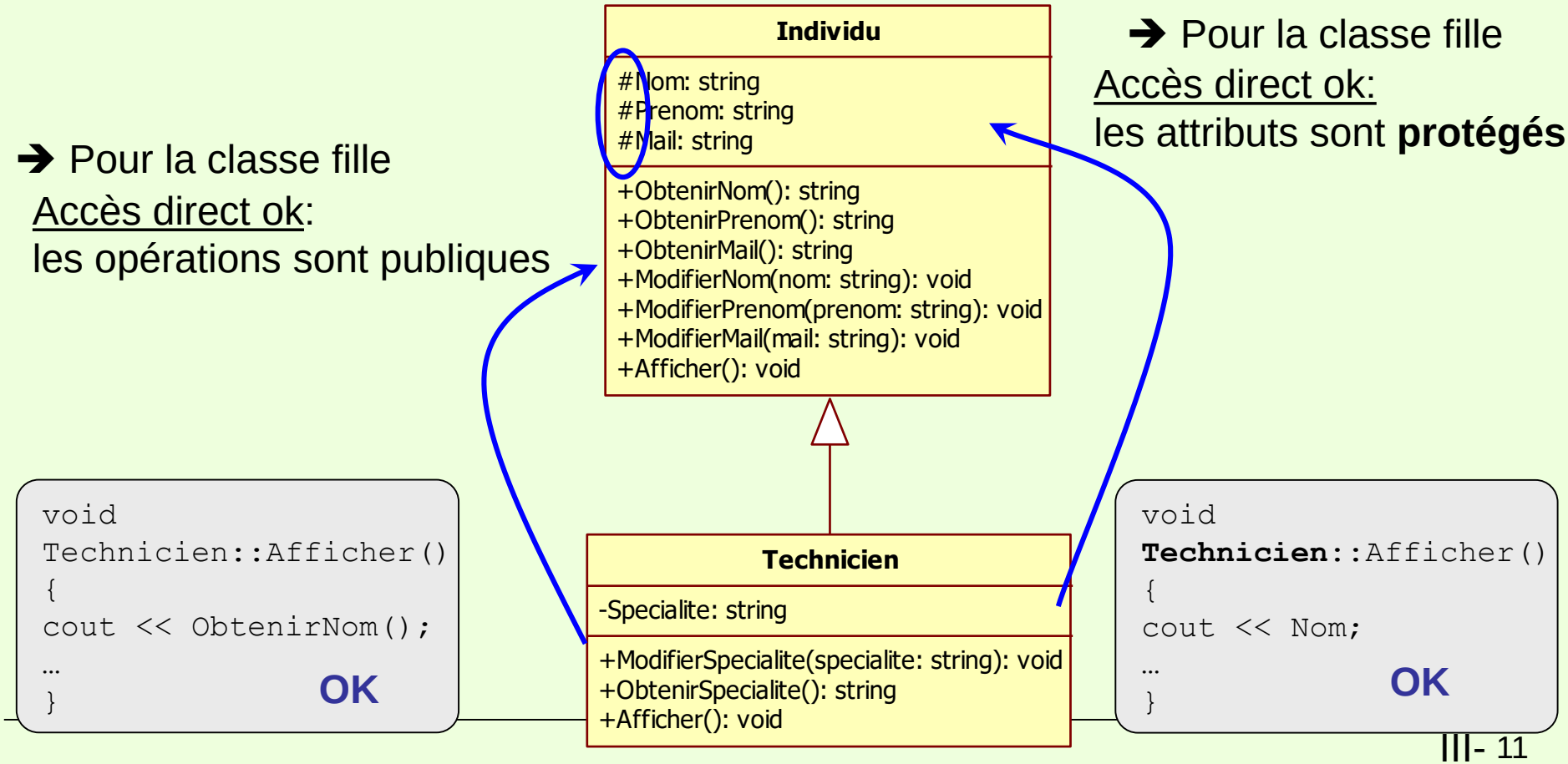
IV – Héritage (généralisation)

- La visibilité des attributs et des méthodes de la classe mère est appliquée à la classe fille



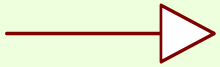
IV – Héritage (généralisation)

- La visibilité des attributs et des méthodes de la classe mère est appliquée à la classe fille

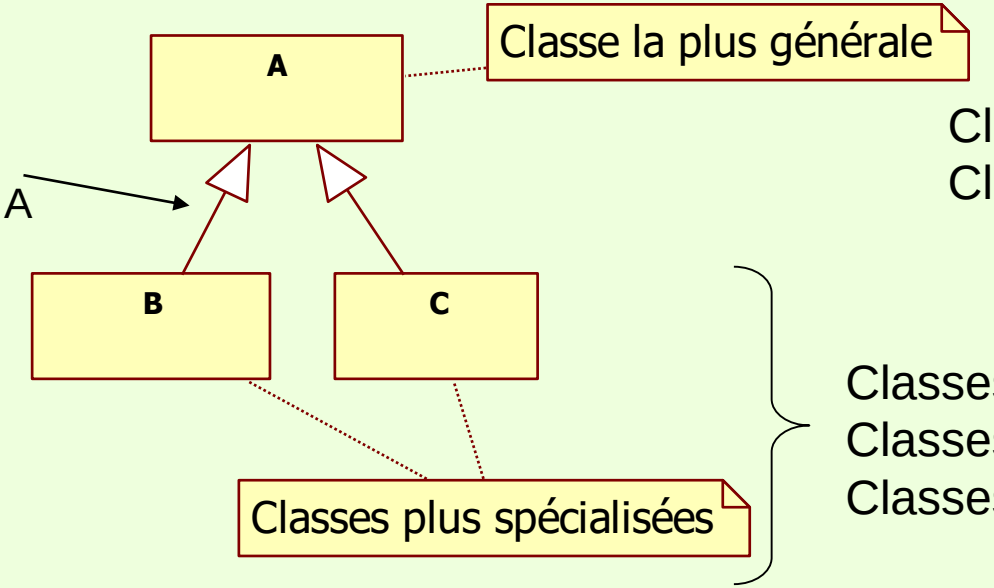


IV – Héritage, définition

- L'héritage (ou généralisation) permet de définir qu'une classe ***est du type*** d'une autre classe
- L'héritage est une relation unidirectionnelle

Symbole UML : 

Héritage (B hérite de A),
Dérivation (B dérive de A),
B est une spécialisation de A
B est du type de A



IV – Héritage, définition

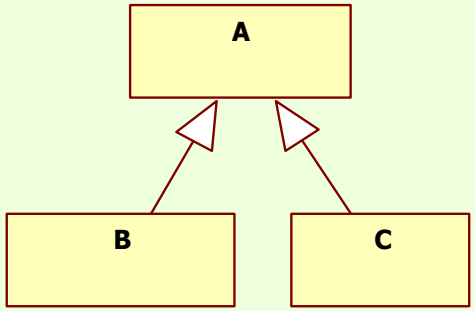
- La classe dérivée possède tous les membres (attributs et méthodes) de la classe mère, avec les même restrictions d'accès que celles définies dans la classe mère

visibilité dans la classe mère	**visibilité dans la classe fille**
public	public
protected	protected
private	*« private »* inaccessible

IV – Héritage, définition

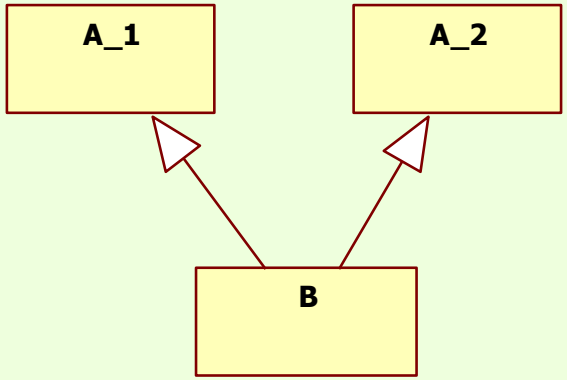
- Il existe la dérivation multiple

B hérite des membres de A
C hérite des membres de A



- Il existe l'héritage multiple

B hérite des membres de A_1 et de A_2



Exemple:
`Moto` hérite de `Objet2Roues` et `ObjetMotorisé`
`MotoCross` hérite de `Moto` et `MatérielToutTerrain`

IV – Héritage, mises en garde

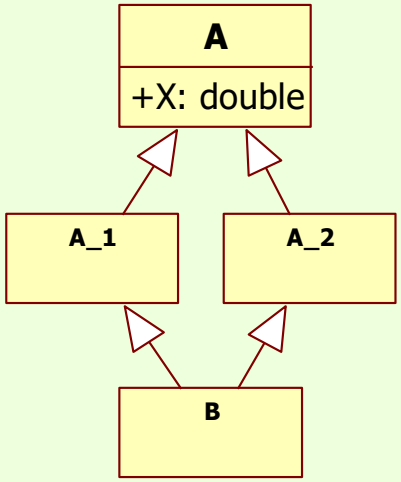
- Une relation d'héritage n'a un sens que si elle est vraie dans une seule direction

  Il est vrai qu'un étudiant est forcément un individu, mais un individu n'est pas forcément un étudiant ➔ OK

- L'héritage multiple peut être problématique *(conflits)*

  B possède t il deux fois les membres de A ?

  B::A_1::X et B::A_2::X



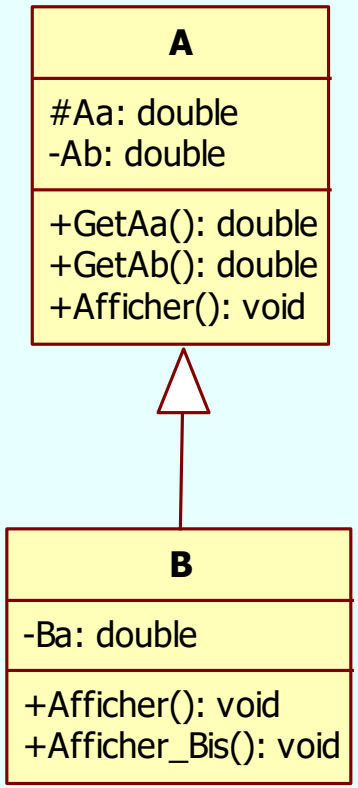
- L'héritage est mal adapté à la réutilisation d'implémentation avec évolutions

  Une classe fille est très fortement couplée à sa classe mère, si la classe mère évolue les classes filles devront probablement évoluer

  *(À moins d'avoir défini une **architecture d'objet** immuable)*

IV – Héritage en C++

UML



C++

```
class A
{
protected:
  double Aa
private:
  double Ab;
public:
  A() {};
  double GetAa()
   {return Aa;}
  double GetAb()
   {return Ab;}
  void Afficher()
   { cout<< Aa << Ab;}
};
```

```
class B : public A
{
private:
  double Ba;
public:
  B() {};
  void Afficher()
   {
   cout << Aa << GetAb();
   cout << Ba << endl;
   }
 void Afficher_Bis()
   {
   A::Afficher();
   cout << Ba << endl;
   }
};
```

IV – Héritage en C++

- Ordre des constructeurs
  - Le constructeur de la classe mère est exécuté avant celui de la classe fille
  - A la construction de la classe fille, il est possible de passer des paramètres au constructeur de la classe mère, sinon le constructeur par défaut de la classe mère est exécuté

```
class A
{
double X;
public:
  A()
   { cout << "A:defaut" << endl;}
  A(double a)
   { X=a;
     cout << "A:General" << endl;}
};
```

```
class B: public A
{
public:
  B()
   { cout << "B:defaut" << endl;}
  B(double a):A(a)
   { cout << "B:General" << endl;}
};
```

```
void main()
{
B ob1; //X=?
B ob2(3);//X=3
}
```

Affichage:
A:defaut
B:defaut

IV – Héritage en C++

- Ordre des constructeurs
  - Le constructeur de la classe mère est exécuté avant celui de la classe fille
  - A la construction de la classe fille, il est possible de passer des paramètres au constructeur de la classe mère, sinon le constructeur par défaut de la classe mère est exécuté

```
class A
{
double X;
public:
  A()
   { cout << "A:defaut" << endl;}
  A(double a)
   { X=a;
   cout << "A:General" << endl;}
};
```

```
class B: public A
{
public:
  B()
   { cout << "B:defaut" << endl;}
  B(double a):A(a)
   { cout << "B:General" << endl;}
};
```

```
void main()
{
B ob1; //X=?
B ob2(3);//X=3
}
```

Résultat:

A:General
B:General

IV – Héritage en C++

- Ordre des destructeurs
  - Le destructeur de la classe fille est exécuté avant le destructeur de la classe mère

```
class TableauDouble
{
double *Tab;
int Taille;
public:
  TableauDouble(int taille=1)
        :Taille(taille)
   { Tab = new double[Taille]; }
  double GetElement(int i) const
   { return Tab[i]; }
  void SetElement(int i, double v)
   { Tab[i] = v; }
  int GetTaille() const
   { return Taille; }
  ~TableauDouble() // destructeur
   { delete[] Tab;
   cout << "TD:Destructeur"<< endl;
   }
};
```

```
class VecteurZero:
        public TableauDouble
{
public:
  VecteurZero(int taille)
        :TableauDouble(taille)
    {for( int i=0; i<GetTaille(); i++)
        SetElement(i, 0);
    }
  ~VecteurZero() //Destructeur inutile
   {cout << "VZ:Destructeur"<< endl;}
};
```

```
void main()
{
VecteurZero a(10);
a.SetElement( 0, 1+a.GetElement(0));
}
```