

LE LANGAGE C++

MASTER 1

TYPES ET SOUS-TYPES

Papes et Soupapes...

Jean-Baptiste.Yunes@u-paris.fr
U.F.R. d'Informatique
Université de Paris

11/2021

LA SPÉCIALISATION

La **classe** :

- est une unité d'encapsulation
- est un modèle de construction, une description concrète, une implémentation
- définit un type de données

Comment **étendre** une classe ?

- Lui rajouter du (ou modifier) son comportement...

L'héritage consiste à personnaliser une classe existante de sorte qu'elle se conduise de façon particulière en :

- lui **ajoutant** des fonctionnalités
- **modifiant** le comportement des actions déjà existantes

L'héritage ne modifie pas la classe de base :

- il construit une **sous-classe**

- Supposons que l'on dispose d'une classe :

```
class Individu {  
    public:  
        const string getPrenom() const;  
};
```

```
#include "Individu.hpp"  
  
const string Individu::getPrenom() const {  
    return "Louis";  
}
```

- Pour lui ajouter des fonctionnalités :

```
#include "Individu.hpp"

class Sportif : public Individu {
public:
    const string getSportPratique() const;
};
```

Noter le
qualificatif public!

```
#include "Sportif.hpp"

const string Sportif::getSportPratique() const {
    return "Hockey sur gazon";
}
```

- Pour l'instant nous ne faisons que des dérivations publiques!!! (On verra plus tard pourquoi).

- Les définitions précédentes permettent d'écrire :

```
#include "Sportif.hpp"

int main() {
    Sportif s;

    cout << s.getPrenom() << " pratique le " <<
        s.getSportPratique() << endl;
    return 0;
}
```

```
[jby]> ./main
Louis pratique le Hockey sur gazon
[jby]>
```

- Ok, essayons de modifier quelques petites choses...

- Modifier des comportements (la redéfinition de fonctions membres) :

```
#include "Individu.hpp"

class Sportif : public Individu {
public:
    const string getSportPratique() const;
    const string getPrenom() const;
};
```

```
#include "Sportif.hpp"

const string Sportif::getSportPratique() const {
    return "Hockey sur gazon";
}

const string Sportif::getPrenom() const {
    return "Reginald"; // Pridmore, rip 1918
}
```

```
[jby]> ./main
Reginald pratique le
Hockey sur gazon
[jby]>
```

C'est une redéfinition

- Une redéfinition extensive :

```
[jby]> ./main  
Bonjour la compagnie  
[jby]>
```

```
class Individu {  
    public:  
        const string sePresente() const;  
}  
class IndividuFamiliier : public Individu {  
    public:  
        const string sePresente() const;  
};  
const string Individu::sePresente() const {  
    return "Bonjour";  
}  
const string IndividuFamiliier::sePresente() const {  
    return Individu::sePresente() + " la compagnie";  
}
```

C'est une
redéfinition

avec réutilisation

- Les deux fonctions membres cohabitent...

- Et avec les constructeurs ???

```
class Individu {  
    private:  
        const string prenom;  
    public:  
        Individu(const string prenom);  
        const string getPrenom() const;  
};
```

```
#include "Individu.hpp"  
  
Individu::Individu(const string prenom)  
    : prenom(prenom) {  
}  
  
const string Individu::getPrenom() const {  
    return prenom;  
}
```

- Et avec les constructeurs ???

```
class Sportif : public Individu {  
    private:  
        const string sport;  
    public:  
        Sportif(const string prenom, const string sport);  
        const string getSportPratique() const;  
};
```

Syntaxe similaire à l'initialisation de membres de type classe

```
#include "Sportif.hpp"
```

```
Sportif::Sportif(const string prenom, const string  
sport)  
    : Individu(prenom), sport(sport) {  
}  
const string Sportif::getSportPratique() const {  
    return sport;  
}
```

- Et avec les constructeurs ???

```
#include "Sportif.hpp"
```

```
Sportif::Sportif(const string prenom, const string  
sport)
```

```
    : Individu(prenom), sport(sport) {
```

```
}
```

```
const string Individu::getSportPratique() const {  
    return sport;
```

```
}
```

*Syntaxe similaire à l'initialisation
de membres de type classe*

- L'ordre des constructions est naturel :
 - on alloue la mémoire pour le tout
 - on initialise d'abord la classe de base
 - puis la classe en dessous, etc.

- Toute expression est possible dans l'appel au constructeurs de base :

```
class A {
    public:
        A(int, double);
};

class B : public A {
    public:
        B(char *, int, int);
};

B::B(char *s, int angle, int valeur)
    : A(valeur%3, sin(angle*3.14/2/90)) {
    ...
}
```

- Héritage et données membres de classe

```
class U {  
    public:  
        U(int);  
};
```

```
class A {  
    public:  
        A(int, double);  
};
```

```
class B : public A {  
    private:  
        U unChamp;  
    public:  
        B(char *, int, int);  
};
```

```
B::B(char *s, int angle, int valeur)  
    : A(valeur%3, sin(angle*3.14/2/90)), unChamp(valeur+4) {  
    ...  
}
```

classe
de base

membre de
classe

- Héritage et données membres de classe

```
class A {  
    public:  
        A(int, double);  
};  
  
class B : public A {  
    private:  
        A unA;  
    public:  
        B(char *, int, int);  
};  
  
B::B(char *s, int angle, int valeur)  
    : A(valeur%3, sin(angle*3.14/2/90)), unA(valeur+4, 3.0) {  
    ...  
}
```

classe
de base

membre de
classe

- Et les destructeurs ?
- La destruction s'opère dans l'ordre inverse des constructions...
- sous-classe d'abord
- super-classe, etc.
- puis désallocation pour finir...

```

class A {
    public:
        A() { cout << "A()" << endl; }
        ~A() { cout << "~A()" << endl; }
};

class B : public A {
    A a;
    public:
        B() { cout << "B()" << endl; }
        ~B() { cout << "~B()" << endl; }
};

class C : public B {
    public:
        C() { cout << "C()" << endl; }
        ~C() { cout << "~C()" << endl; }
};

int main() {
    B b;
    C c;
}

```

```

[Gibi:~]% ./main
A()
A()
B()
A()
A()
B()
C()
~C()
~B()
~A()
~A()
~B()
~A()
~A()
[Gibi:~]%

```

REDÉFINITION

VS

SURCHARGE

- Attention la redéfinition masque les définitions...

Surcharge

Surcharge

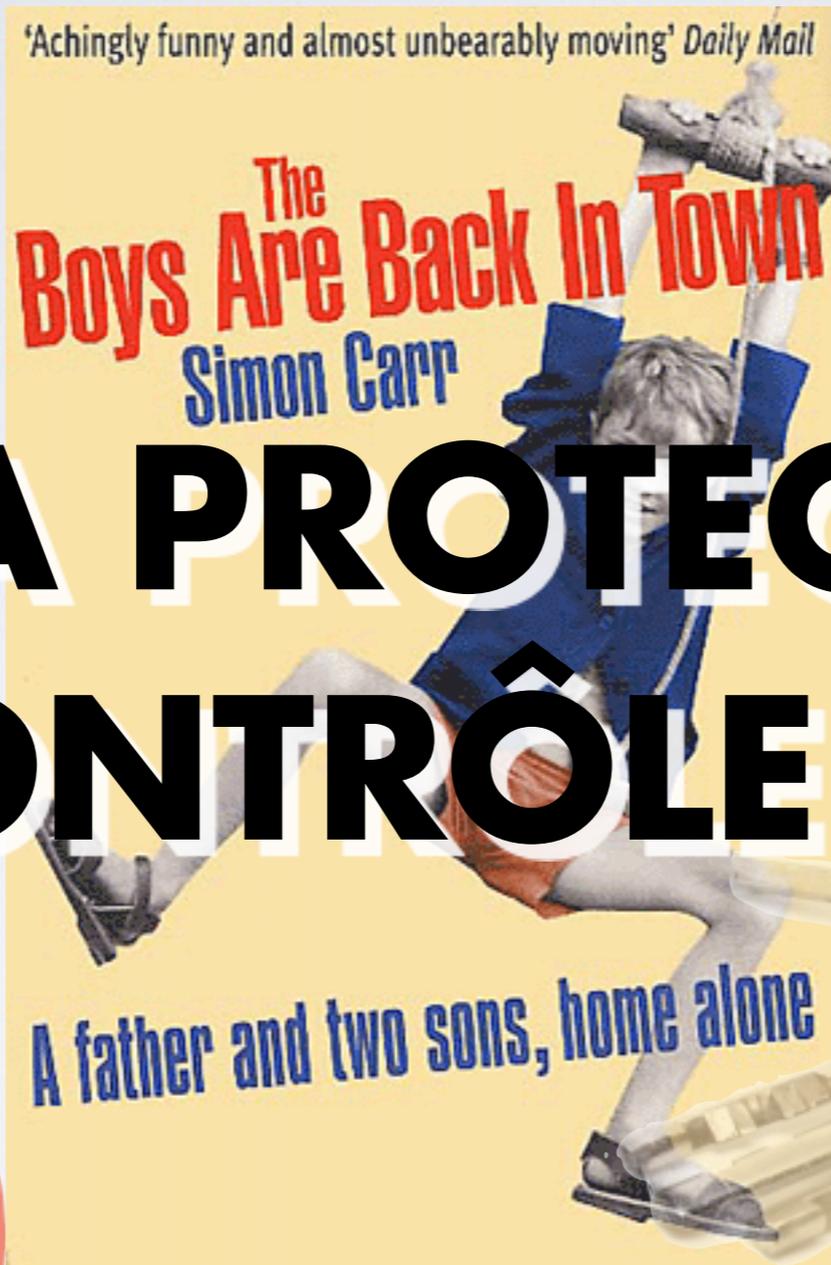
Redéfinition dans B par rapport à A

```
class A {
public:
    void m1(int i);
    void m1(char f);
    void m2(int i);
    void m2(char c);
};
```

```
class B : public A {
public:
    void m1(int i);
    void m1(float f);
};
```

```
int main() {
    B b;
    b.m1(3);           // B::m1(int)
    b.m1(3.4f);       // B::m1(float)
    b.m1('z');       // B::m1(int) avec conversion
    b.m2(3);          // A::m2(int)
    b.m2(3.4f);       // ambigu float~int ou float~char ?
    b.m2('z');       // A::m2(char)
    return 0;
}
```

Surcharge dans B



LA PROTECTION LE CONTRÔLE D'ACCÈS



- Souvenons-nous...
- `private` : visible **uniquement** depuis les fonctions membres de **la classe**
- `public` : visible depuis n'importe quelle partie du code
- Être dans une fonction membre d'une sous-classe ne signifie pas être dans la classe de base!!!!

```
class DeBase {
    private:
        int champPrive;
        void fonctionMembrePrivee();
    public:
        int champPublic;
        void fonctionMembrePublique();
};
```

dérivation publique

```
class SousClasse : public DeBase {
    private:
        int champPrive2;
        void fonctionMembrePrivee2();
    public:
        int champPublic2;
        void fonctionMembrePublique2();
};
```

```
void SousClasse::fonctionMembrePublique2() {
    champPrive = 666; // privé même pour la sous-classe
    champPrive2 = 666;
}

void f() {
    SousClasse sc;
    sc.champPrive; // privé c'est privé et puis c'est tout
    sc.champPublic;
}
```

- La protection telle que vue jusqu'ici peut-être considérée comme trop restrictive...
- Un troisième domaine :
 - `protected` : un membre (donnée ou fonction) protégé est :
 - de l'extérieur (*i.e.* ni dans la classe ni dans une sous-classe) considéré inaccessible
 - de l'intérieur (*i.e.* dans la classe ou dans n'importe quelle sous-classe) considéré comme accessible

```
class DeBase {
    private:
        int champPrive;
        void fonctionMembrePrivee();
    public:
        int champPublic;
        void fonctionMembrePublique();
};
```

dérivation publique

```
class SousClasse : public DeBase {
    private:
        int champPrive2;
        void fonctionMembrePrivee2();
    public:
        int champPublic2;
        void fonctionMembrePublique2();
};
```

```
void SousClasse::fonctionMembre....() {
    champPrive = 666; // privé même pour la sous-classe
    champPrive2 = 666;
}

void f() {
    SousClasse sc;
    sc.champPrive; // privé c'est privé et puis c'est tout
    sc.champPublic;
}
```

```
class DeBase {
    protected:
        int champPrive;
        void fonctionMembrePrivee();
    public:
        int champPublic;
        void fonctionMembrePublique();
};
```

dérivation publique

```
class SousClasse : public DeBase {
    private:
        int champPrive2;
        void fonctionMembrePrivee2();
    public:
        int champPublic2;
        void fonctionMembrePublique2();
};
```

```
void SousClasse::fonctionMembre....() {
    champPrive = 666; // accessible même dans la sous-classe
    champPrive2 = 666;
}

void f() {
    SousClasse sc;
    sc.champPrive; // privé c'est privé et puis c'est tout
    sc.champPublic;
}
```

- Et les amitiés, ça marche comment maintenant ?

dérivation publique

```
class DeBase {
    private:
        int champPrive;
    protected:
        int champProtege;
    public:
        int champPublic;
};

class SousClasse : public DeBase {
    private:
        int champPrive2;
    protected:
        int champProtege2;
    public:
        int champPublic2;
    friend void uneFonctionQuiTraine();
};
```

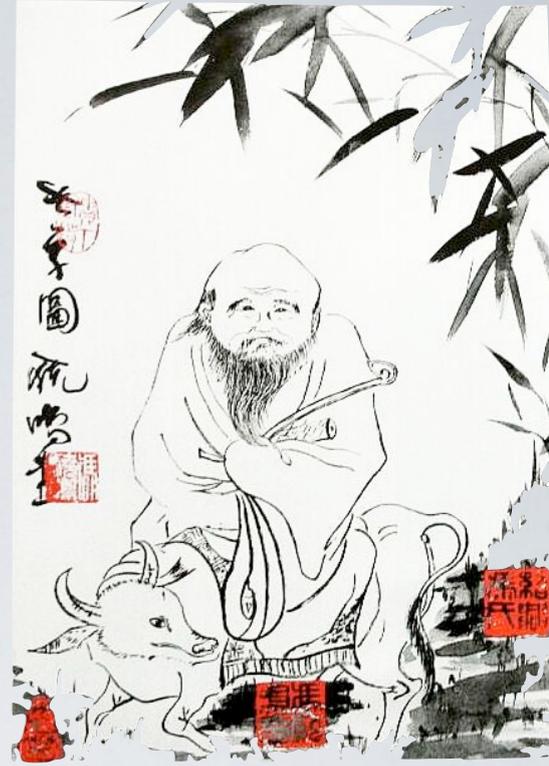
```
void uneFonctionQuiTraine() {
    SousClasse s;
    s.champPrive;
    s.champProtege;
    s.champPublic;
    s.champPrive2;
    s.champProtege2;
    s.champPublic2;
}
```

```
void uneAutreFonctionQuiTraine() {
    SousClasse s;
    s.champPrive;
    s.champProtege;
    s.champPublic;
    s.champPrive2;
    s.champProtege2;
    s.champPublic2;
}
```

- En résumé (**pour une dérivation publique**) :

Protection dans la classe de base	Accessibilité dans une fonction membre de la classe dérivée	Accessibilité dans une fonction amie de la sous-classe	Accessibilité par utilisation de la classe dérivée (hors cas précédents)	Protection vis-à-vis d'une nouvelle dérivation
private	non	non	non	private
protected	oui	oui	non	protected
public	oui	oui	oui	public

- Le conseil du vieux sage...
 - Attention à ne pas utiliser la dérivation pour *violer* l'encapsulation
 - en faisant réapparaître publiquement des choses qui devaient être cachées...
 - Attention donc en faisant apparaître des membres avec le qualificatif `protected`...
 - il ne faut jamais oublier que l'on peut dériver une classe...
- Les violations sont forcément conscientes... C'est déjà une chose...



```
class CompteEnBanque {
private:
    int position;
protected:
    void setPosition(int v);
public:
    int getPosition() const;
};
```

```
class Livret : public CompteEnBanque {
public:
    void calculeInterets();
};

void Livret::calculeInterets() {
    int v = getPosition();
    v *= 1.035;
    setPosition(v);
}
```

```
class CompteLibre : public CompteEnBanque {
public:
    void setPosition(int v);
};

void CompteLibre::setPosition(int v) {
    CompteEnBanque::setPosition(v);
}
```

```
void escroquerie() {
    CompteLibre cl;
    cl.setPosition(1000000000); // Je crois que cela me suffira (pour l'instant)
}
```

- setPosition **en** protected **était-ce bien malin ?**
- **Pas de réponse absolue...**

- En C++, il est possible de modifier l'accessibilité, dans le sens que l'on souhaite, d'un membre hérité
- Attention (encore une fois) à ne pas *violer* l'encapsulation voulue...
- À n'utiliser qu'avec précaution et parcimonie...

• Changement de domaine et héritage

```
class DeBase {
    protected:
        int champProtege;
    public:
        int champPublic;
};

class SousClasse : public DeBase {
    protected:
        using DeBase::champPublic;
    public:
        using DeBase::champProtege;
};
```

```
void uneFonction() {
    DeBase o;
    o.champProtege; // bon ok c'est protégé, j'exagère un peu
    o.champPublic; // ok c'est public

    SousClasse s;
    s.champProtege; // le truc est devenu public! Bizarre mais bon...
    s.champPublic; // rien ne va plus, un truc visible devient invisible!
}
```

- Attention en C++ < 11, on ne peut pas vraiment empêcher la construction de sous-classe (sauf à «cacher» les constructeurs)
- En C++ >= 11 il est possible d'empêcher la construction de sous-classe

```
class X final {};
```

```
class Y : public X {};
```

LA COMPATIBILITÉ DES TYPES

- Si l'on considère la spécialisation comme un *ajout* structurel
- On doit pouvoir opérer à l'envers, *i.e.* supprimer les choses rajoutées. On resterait cohérent du point de vue des définitions, non ?
- La compatibilité des types et des classes...

- D'une manière générale
 - on peut toujours convertir une instance d'une classe donnée en un objet de sa super-classe
 - on peut toujours convertir un pointeur vers une instance d'une classe donnée en un pointeur vers un objet de sa super-classe
- C'est évidemment transitif (on remonte tant que l'on veut dans la hiérarchie des classes)

```
class DeBase {
    ...
};

class SousClasse : public DeBase {
    ...
};

class BienEnDessous : public SousClasse {
    ...
};

int main() {
    BienEnDessous unObjet;
    DeBase unObjetDeBase = unObjet;
    unObjet = unObjetDeBase; // strictly forbidden...

    BienEnDessous *p = &unObjet;
    DeBase *p2;
    p2 = p;
    p = p2; // formellement interdit
}
```

• Et les méthodes appelées ?

```
class DeBase {
public:
    void f() { cout << "DeBase" << endl; }
};

class SousClasse : public DeBase {
public:
    void f() { cout << "SousClasse" << endl; }
};

int main() {
    SousClasse unObjet;
    unObjet.f();
    DeBase instanceDeBase = unObjet;
    instanceDeBase.f();

    SousClasse *pSousClasse = &unObjet;
    pSousClasse->f();
    DeBase *pBase = pSousClasse;
    pBase->f();
    return 0;
}
```

```
[Gibi:~]% ./main
SousClasse
DeBase
SousClasse
DeBase
[Gibi:~]%
```

- En résumé :
- la méthode qui sera appelée à l'exécution est déterminée statiquement (à la compilation) - *early binding*
- la méthode appelée est celle apparaissant **dans le type** de la référence ou du pointeur, ou à défaut une méthode héritée (attention au masquage - redéfinition vs. surcharge)
- La liaison tardive (*late binding*) est possible...

• La liaison tardive.

la méthode est éligible à la liaison tardive (dynamique)

```
class DeBase {
public:
    virtual void f() { cout << "DeBase" << endl; }
};
class SousClasse : public DeBase {
public:
    virtual void f() { cout << "SousClasse" << endl; }
};
void uneFonction(DeBase &o) { o.f(); }
int main() {
    SousClasse unObjet;
    unObjet.f();
    DeBase instanceDeBase = unObjet;
    instanceDeBase.f();
    SousClasse *pSousClasse = &unObjet;
    pSousClasse->f();
    DeBase *pBase = pSousClasse;
    pBase->f();
    uneFonction(unObjet);
    return 0;
}
```

```
[Gibi:~]% ./main
SousClasse
DeBase
SousClasse
SousClasse
SousClasse
[Gibi:~]%
```

Le passage par référence permet la liaison tardive

- En résumé :
- l'appel à une méthode qualifiée de virtuelle (`virtual`) provoquera à l'exécution la recherche de la méthode appropriée - *late binding*
- la méthode appelée sera celle apparaissant **dans l'objet** désigné par la référence ou par le pointeur, ou à défaut une méthode héritée (attention au masquage - redéfinition vs. surcharge)
- Polymorphisme par sous-typage

- Définition :
- Une classe dont la définition contient au moins une méthode à liaison tardive est appelée **classe polymorphe**.

**CONSTRUCTEUR PAR
COPIE ET HÉRITAGE...**

- Qu'advient-il des constructions par copie en liaison avec l'héritage...
- de façon surprenante (?), lorsqu'une classe dérivée définit un constructeur par copie, C++ n'appelle pas automatiquement le constructeur de copie de la classe de base (même s'il existe)...
- si rien n'est indiqué, un appel au constructeur sans paramètre de la classe de base est tenté...

```
class A {
    public:
        A() { cout << "A()" << endl; }
        A(const A &a) { cout << "A(const A&)" << endl; }
};

class B : public A {
    public:
        B() { cout << "B()" << endl; }
        B(const B &b) { cout << "B(const &B)" << endl; }
};

void f(B unObjet)
{

}

int main()
{
    B b;
    cout << "Appel a f(B)" << endl;
    f(b);
    return 0;
}
```

rien
d'indiqué

```
[Gibi:~]% ./main
A()
B()
Appel a f(B)
A()
B(const &B)
[Gibi:~]%
```

```

class A {
    public:
        A() { cout << "A()" << endl; }
        A(const A &a) { cout << "A(const A&)" << endl; }
};

class B : public A {
    public:
        B() { cout << "B()" << endl; }
        B(const B &b) : A(b) { cout << "B(const &B)" << endl; }
};

void f(B b) {
}

int main()
{
    B b;
    cout << "Appel a f(B)" << endl;
    f(b);
    return 0;
}

```

explicite...

```

[Gibi:~]% ./main
A()
B()
Appel a f(B)
A(const A&)
B(const &B)
[Gibi:~]%

```

HÉRITAGE ET SOUS-TYPAGE

- L'héritage **public** observé jusqu'ici correspond à la spécialisation conceptuelle
 - autorise le sous/sur-typage, le polymorphisme, la liaison dynamique
- L'héritage peut servir à d'autres choses...

- L'héritage (privé) comme composition :
- une composition « ordinaire »

```
class Moteur {  
    public:  
        void demarre() { ... };  
};
```

```
class Voiture {  
    private:  
        Moteur leMoteur;  
    public:  
        void demarre() { leMoteur.demarre(); }  
};
```

- L'héritage comme composition :
- la composition obtenue par héritage...

```
class Moteur {  
    public:  
        void demarre() { ... };  
};
```

héritage privé!

```
class Voiture : private Moteur {  
    public:  
        void demarre() { Moteur::demarre(); }  
};
```

redéfinition

- L'héritage privé ne définit pas un sous-typage...

```
class Moteur {  
};  
  
class Voiture : private Moteur {  
};  
  
int main() {  
    Voiture v;  
    Moteur m;  
    m = v; // interdit!  
    Moteur *pm;  
    pm = &v; // interdit!  
    return 0;  
}
```

- ne pas l'employer!!!

Je préfère même
le Comic Sans MS

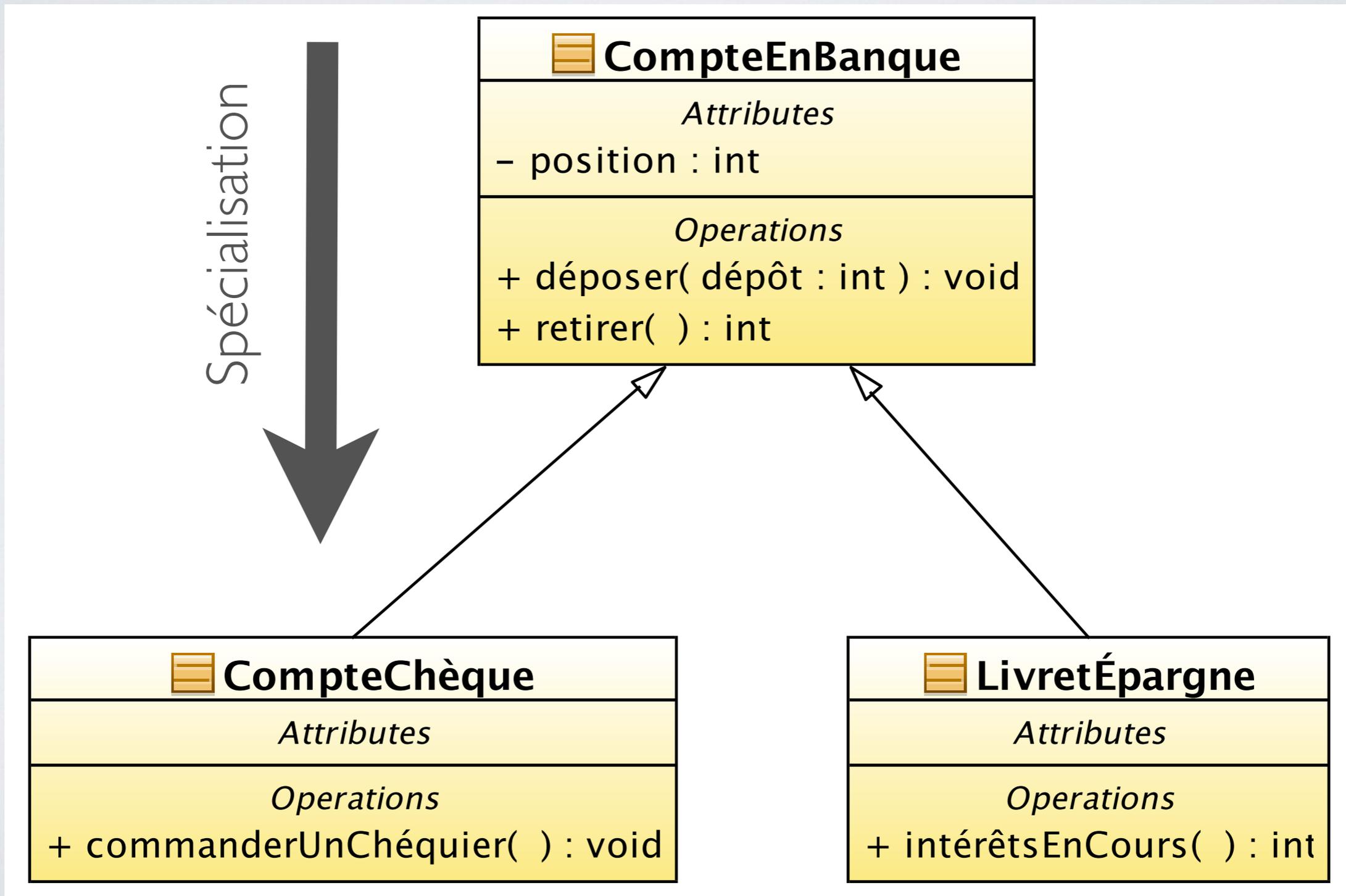
J'ai dit **NON!**

- L'héritage protégé existe aussi, il ne définit pas non plus de sous-typage...
- C'est aussi une composition
 - Elle autorise une visibilité différente des membres hérités
- Son usage est rare (très rare)...
- Je n'ai pas d'exemple convaincant (mais je ne suis pas le seul)...

- Comment tout cela fonctionne...

Statut du membre hérité dans la classe dérivée		Droit d'accès du membre dans la classe de base		
		private	protected	public
Héritage	private	private	private	private
	protected	private	protected	protected
	public	private	protected	public

- Représentation UML d'une spécialisation :



- Le conseil du vieux sage...

- Pour la **composition**

- implémentez-la avec des **données membres**

- Pour la **spécialisation** (extension d'une classe concrète)

- vous pouvez utiliser l'**héritage**

