LE LANGAGE C++
MASTER 1
GÉNÉRALISATION MULTIPLE
HÉRITAGE MULTIPLE

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

2020–2021

L'HERITAGE MULTIPLE

- Rappel : l'héritage peut être employé pour faire de la composition…
  - On veut donc agréger deux contenants (différents) prédéfinis pour en fabriquer un nouveau
    - Si les contenants à agréger sont du même type : une seule solution l'implémentation standard de la composition…
    - Sinon :

Mauvaise implémentation

```
class Magnetophone {
  public:
    void demarreEcoute() { ... };
    void demarreEnregistrement() { ... };
};

class Camera {
  public:
    void demarreProjection() { ... };
    void demarreEnregistrement() { ... };
};

class Camescope : private Magnetophone, private Camera {
  public:
    void enregistreFilm() {
      Magnetophone::demarreEnregistrement();
      Camera::demarreEnregistrement();
    }
    void regardeFilm() {
      Magnetophone::demarreEcoute();
      Camera::demarreProjection();
    }
};
```

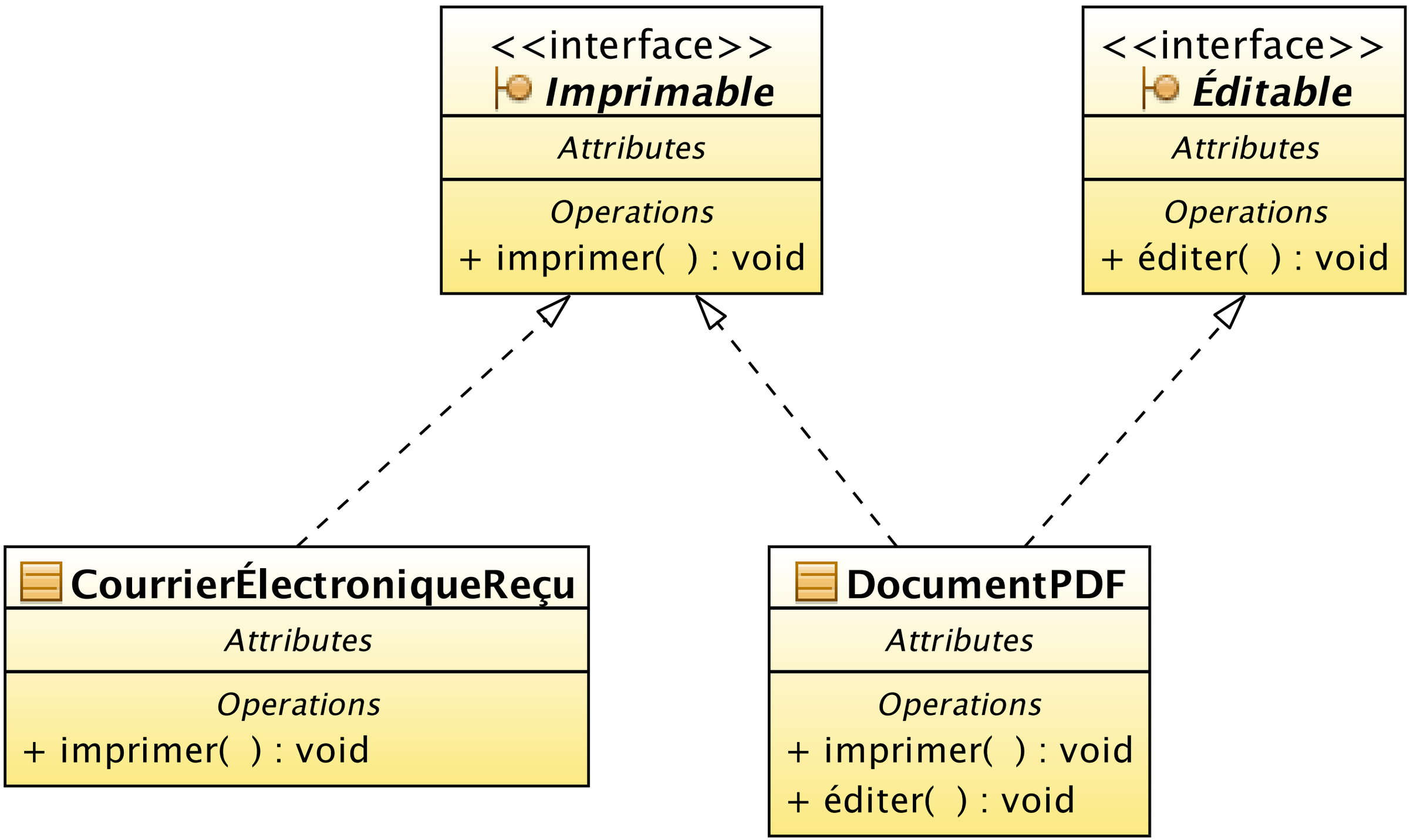
Meilleure implémentation

```
class Magnetophone  {
public:
  void demarreEcoute() { ... };
  void demarreEnregistrement() { ... };
};
class Camera  {
public:
  void demarreProjection() { ... };
  void demarreEnregistrement() { ... };
};
class Camescope {
private:
  Magnetophone  m;
  Camera c;
public:
  void enregistreFilm() {
    m.demarreEnregistrement();
    c.demarreEnregistrement();
  }
  void regardeFilm() {
    m.demarreEcoute();
    c.demarreProjection();
  }
};
```

LA GÉNÉRALISATION MULTIPLE

- d'un point de vue conceptuel, il n'y a aucun problème à ce qu'un objet ait plusieurs types
  - on a déjà vu la *cascade* de généralisation

- on peut remonter plusieurs concepts disjoints d'une classe concrète existante


<<interface>>
Imprimable
Attributes
Operations
+ imprimer( ) : void
<<interface>>
Éditable
Attributes
Operations
+ éditer( ) : void
CourrierÉlectroniqueReçu
Attributes
Operations
+ imprimer( ) : void
DocumentPDF
Attributes
Operations
+ imprimer( ) : void
+ éditer( ) : void

C’est conceptuel

```
class Imprimable {
  public:
    virtual void imprimer()=0;
};

class Editable {
  public:
    virtual void editer()=0;
};

class CourrierElectroniqueRecu : virtual public Imprimable {
  public:
    virtual void imprimer() { /* just do it... */ }
};

class DocumentPDF : virtual public Imprimable,
                    virtual public Editable {
  public:
    virtual void editer() { /* do your best... */ }
    virtual void imprimer() { /* merci Johannes Gensfleisch
                                 zur Laden zum Gutenberg */ }
};
```

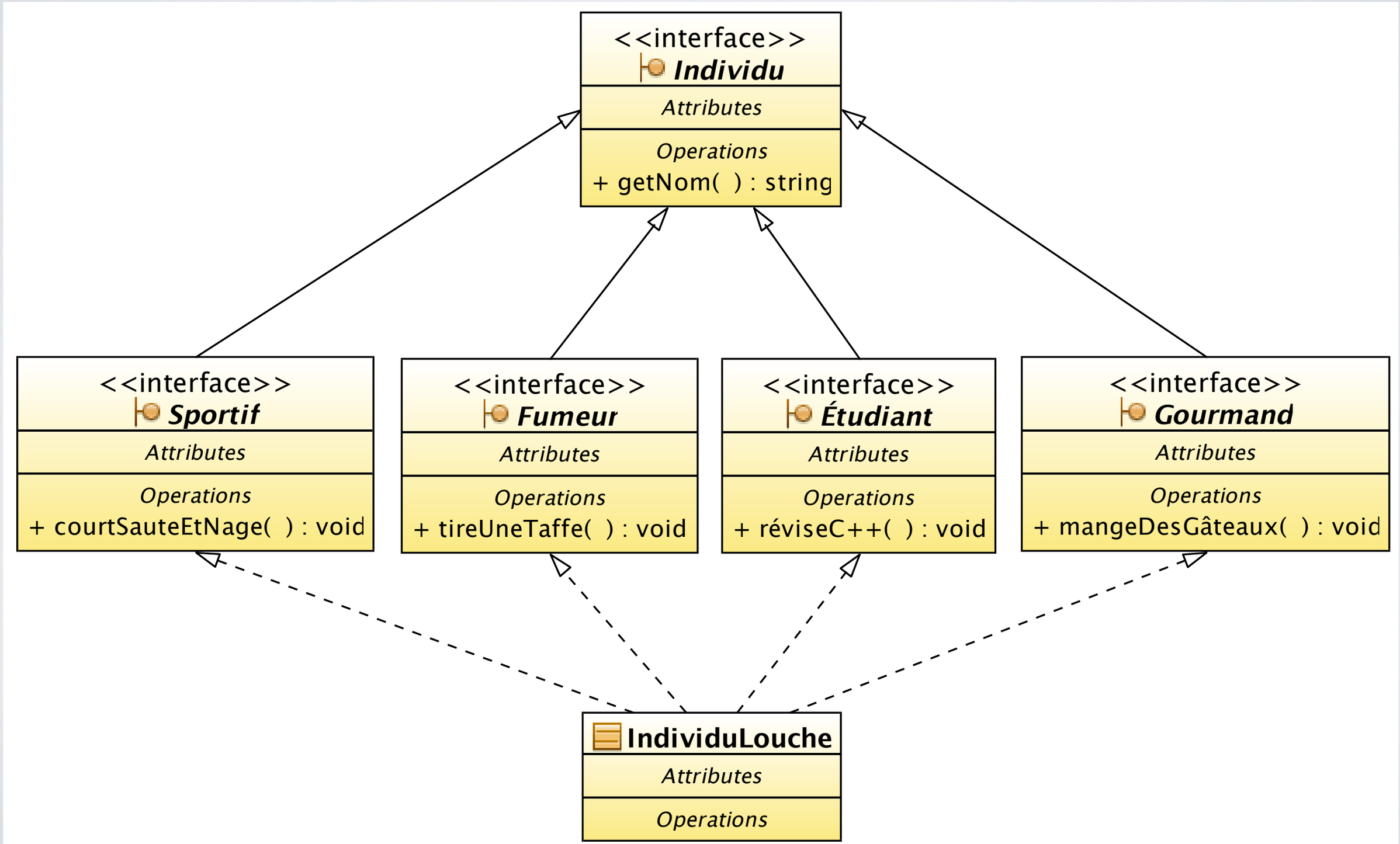
- Aucun problème théorique
  - puisqu'on *hérite* de rien (ou presque), *i*.e. pas d'implémentation, tout est abstrait
- Seul problème pratique, l'éventuel conflit de nom
  - deux super-types utilisent un même nom…
    - si les actions sont différentes : problème
    - sinon, ok…

```
class IPoint {
  public:
    virtual void affiche()=0;
};

class ICouleur {
  public:
    virtual void affiche()=0;
};

class PointColore : virtual public IPoint,
                    virtual public ICouleur{
  public:
    virtual void affiche() { /* OK */ }
};
```

On suppose que **`affiche`** signifie la même chose dans les deux interfaces


<<interface>>
Individu
Attributes
Operations
+ getNom( ) : string
<<interface>>
Sportif
Attributes
Operations
+ courtSauteEtNage( ) : void
<<interface>>
Fumeur
Attributes
Operations
+ tireUneTaffe( ) : void
<<interface>>
Étudiant
Attributes
Operations
+ réviseC++( ) : void
<<interface>>
Gourmand
Attributes
Operations
+ mangeDesGâteaux( ) : void
IndividuLouche
Attributes
Operations

- Pas de problèmes, la classe de base n'est présente qu'une seule fois...

```
class Base {
};

class SousClasse1 : virtual public Base {
};

class SousClasse2 : virtual public Base {
};

class ToutEnBas :
      virtual public SousClasse1,
      virtual public SousClasse2 {
};
```

- Oui mais, et les constructeurs…
- ben c’est affreux…

```
class Base {
  public:
    Base(int v) {}
};

class Classe1 : virtual public Base {
  public:
    Classe1() : Base(3) {}
};

class Classe2 : virtual public Base {
  public:
    Classe2() : Base(5) {}
};

class ToutEnBas : virtual public Base,
                  virtual public Classe1,
                  virtual public Classe2 {
  public:
    ToutEnBas() : Base(4), Classe1(), Classe2() {}
};
```

N'est appelé que si la partie **`Base`** n'existe pas...

N'est appelé que si la partie **`Base`** n'existe pas...

L'HÉRITAGE MULTIPLE ET LE TYPAGE

```
class Classe1  {
  public:
    virtual void f() {};
};

class Classe2 {
  public:
    virtual void g() {};
};

class T : public Classe1, public Classe 2 {
};

int main() {
  T unT;
  Classe1 *p1;
  p1 = &unT; // OK!
  Classe2 *p2;
  p2 = &unT; // OK!
  p1 = p2; // Non...
  p1 = dynamic_cast<Classe1 *>(p2); // oui si Classe1 polymorphe
  return 0;
}
```

```
class A {
public:
  virtual void f() {};
};
class Classe1 : virtual public A {};
class Classe2 : virtual public A {};
class T : public Classe1, public Classe2 {};

int main() {
  T unT;
  cout << "adresse de unT " << (&unT) << endl;
  Classe1 *p1;
  p1 = &unT;
  cout << "p1 " << p1 << endl;
  A *pA = p1;
  cout << "pa " << pA << endl;
  Classe2 *p2;
  p2 = &unT;
  cout << "p2 " << p2 << endl;
  p1 = dynamic_cast<Classe1 *>(p2); // non trivial!
 return 0;
}
```

```
yunes% ./p
adresse de unT 0x7fff521669c8
p1 0x7fff521669c8
pa 0x7fff521669c8
p2 0x7fff521669d0
yunes%
```

LES CONSEILS

Dr. Walbec Bunsen
The Muppet Show

- N'utilisez pas l'héritage multiple pour faire de la composition
- Réservez-le uniquement pour la généralisation multiple
  - Déclarez toutes les méthodes virtuelles
  - Déclarez tous les héritages comme virtuels
- Ne comptez plus sur l'adresse comme identifiant de l'objet