Programmation Orientée Objet C++ — Episode #4

L’héritage

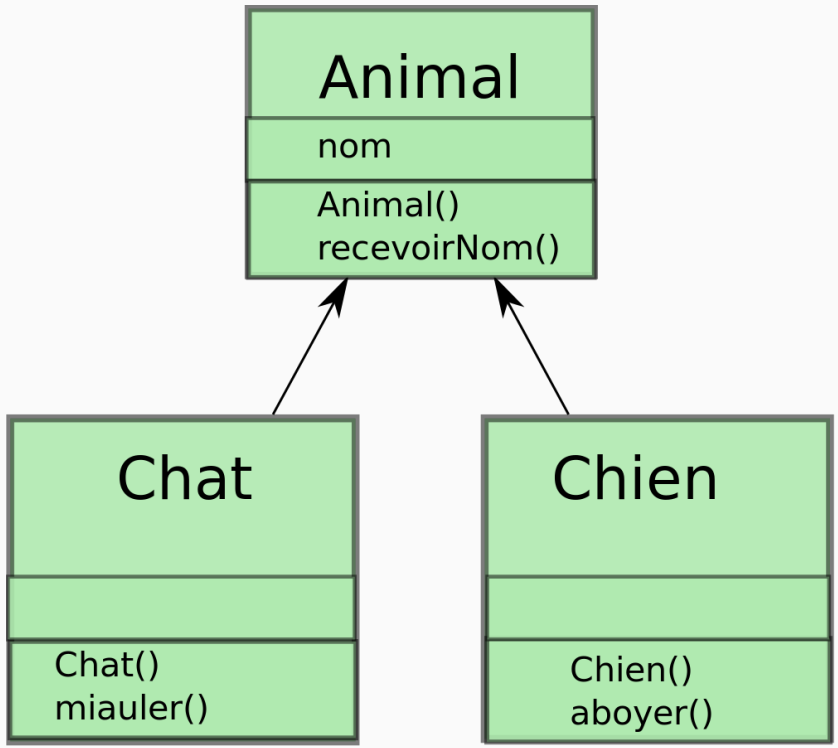
Le principe

Spécialisation
La classe `B` hérite de `A` si elle **spécialise** `A`.

Relation is a
`B` (classe **fille**) **est un** `A` (classe **mère — de base**).

Un exemple
Un *Rectangle* est un *Parallélogramme*.

Un peu de modélisation



Ne pas confondre avec *Composition*.

Concrètement

- *Tout* est récupéré : attributs, méthodes, constantes, ...
- Les ajouts sont gérés dans la classe fille.

La mise en oeuvre

La syntaxe

```
1  class A {
2    private:
3      int att;
4    public:
5      A(int _att) { att = _att, };
6      ~A();
7  };
8  class B : class A {
9    private:
10     string att_B;
11   public:
12     B(int _att, string _att_B) : A(_att)
13       { att_B = _att_B; }
14 };
```

L'exemple en code

```cpp
#include <iostream>
#include <string>

class Personne {
  std::string nom;
  public:
  Personne( std::string _nom );
  Personne( Personne const & p )
      ;
  ~Personne();

  std::string get_nom() const {
    return nom;
  }
};
std::ostream& operator<<(std::
    ostream &os, const Personne
    & p);
```

```cpp
#include "personne.hpp"

class Etudiant : public Personne
    {
  long INE;
  public:
  Etudiant( std::string _nom,
      long _INE );
  ~Etudiant();

  long get_INE() const;
};

std::ostream& operator<<(std::
    ostream &os, const Etudiant
    & e);
```

L'exemple en code

```cpp
#include "personne.hpp"

Personne::Personne(std::string _nom) : nom(_nom) {
  std::cout << "Constructeur Personne" << std::endl;
}

Personne::Personne(Personne const &p) {
  (*this).nom = p.get_nom();
}

Personne::~Personne() {
  std::cout << "Destructeur Personne" << std::endl;
}

std::ostream& operator<<(std::ostream &os, const Personne &p) {
  os << "Personne[ nom : " << p.get_nom() << " ]";
  return os;
}
```

L'exemple en code

```cpp
#include "etudiant.hpp"

Etudiant::Etudiant(std::string _nom, long _INE) : Personne(_nom)
    , INE(_INE) {
  std::cout << "Constructeur Etudiant" << std::endl;
}

Etudiant::~Etudiant() {
  std::cout << "Destructeur Etudiant" << std::endl;
}

long Etudiant::get_INE() const {
  return (*this).INE;
}

std::ostream& operator<<(std::ostream& os, const Etudiant &e) {
  os << "Etudiant[ nom : " << e.get_nom() << ", INE : " << e.
      get_INE() << " ]";
  return os;
}
```

La limitation des droits

Syntaxe

```
class Fille : <droits> Mere
```

Héritage `public` — le plus commun

```
class Fille : public Mere
```

Héritage `protected`

```
class Fille : protected Mere
```

Héritage `private`

```
class Fille : (private) Mere
```

La limitation des droits

Limitation des droits : appliquée à tout ce qui est **hérité**.

Ajouté/redéfini : règles habituelles.

Héritage `public` — le plus commun

```
class Fille : public Mere
```

Héritage `protected`

`protected` : accessible depuis la classe et toutes les classes qui en héritent.

Héritage `private`

```
class Fille : (private) Mere
```

En détails

`public`
Tout ce qui est hérité garde les mêmes droits.

`protected`
Tout ce qui est hérité **et** `public` devient `protected`.

`private`
Tout ce qui est hérité est `private`.

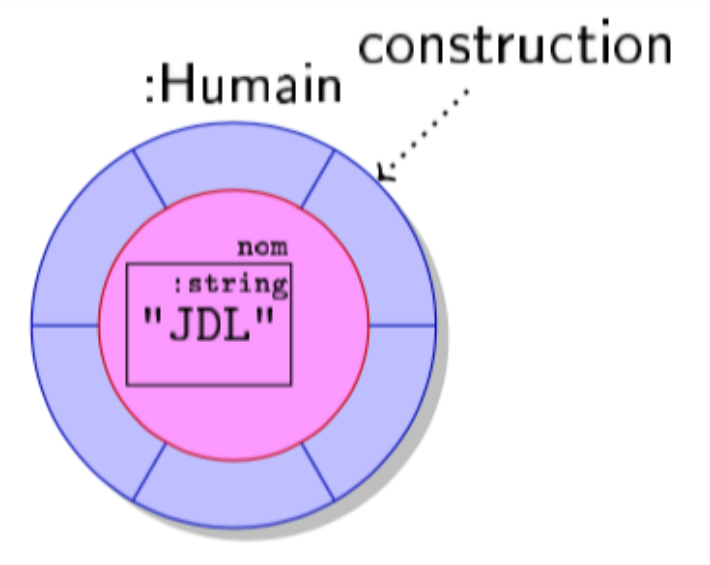
Le constructeur

Une instance de la classe `Etudiant` *contient* une instance de la classe `Personne`.

Cette instance doit être construite

Conséquence
Chaque constructeur de `Etudiant` doit invoquer un constructeur de `Humain`.

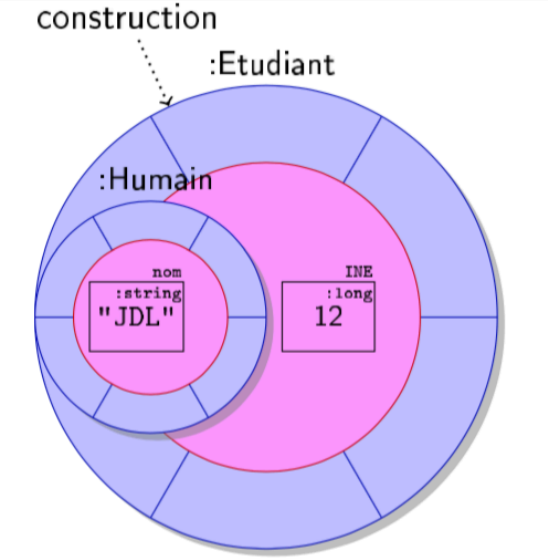
L’exemple en images




```cpp
#include "etudiant.hpp"

int main() {
  Etudiant e("Anthony", 12);
  return 0;
}
```

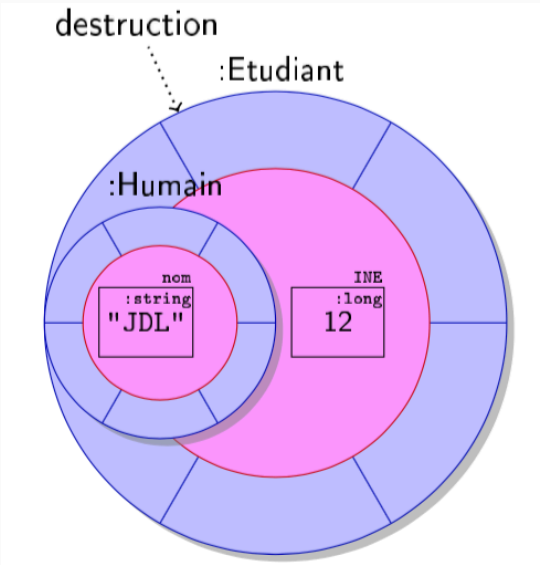
L’exemple en images




```cpp
#include "etudiant.hpp"

int main() {
  Etudiant e("Anthony", 12);
  return 0;
}
```

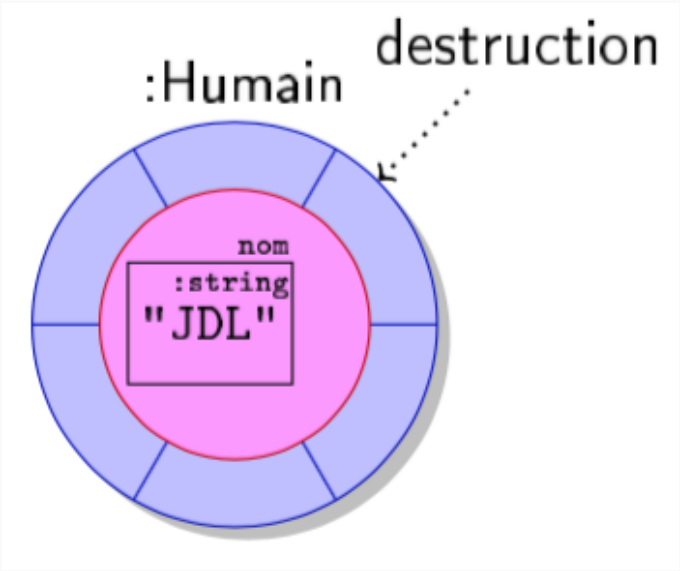
L’exemple en images




```cpp
#include "etudiant.hpp"

int main() {
  Etudiant e;
  return 0;
}
```

L’exemple en images




```cpp
#include "etudiant.hpp"

int main() {
  Etudiant e("Anthony", 12);
  Etudiant e1( e );
  return 0;
}
```

Destructeur et héritage

```
1  #include "destructeur.hpp"
2
3  int main() {
4    A a = B ( 1 );
5    return 0;
6  }
```

```
Constructeur A 1
Constructeur B 1
Constructeur A 11 copie A 1
Destructeur B 1
Destructeur A 1
Destructeur A 11
```

Gestion des variables automatique

- une instance de B (contenant une instance de A)
- une instance de A simple

Destructeur et héritage

```
1  #include "destructeur.hpp"
2
3  int main() {
4    A *a = new B(2);
5    delete a;
6    return 0;
7  }
```

```
Constructeur A 2
Constructeur B 2
Destructeur A 2
```

Il manque un destructeur !

Destructeur et héritage

```
1  #include "virtual.hpp"
2
3  int main() {
4    A *a = new B(2);
5    delete a;
6    return 0;
7  }
```

```
Constructeur A 2
Constructeur B 2
Destructeur B 2
Destructeur A 2
```

Dès qu'il pourrait y avoir héritage : destructeur **virtuel** — TBC

Héritage et surcharge

La surcharge de fonctions

```
1  #include <iostream>
2  using namespace std;
3
4  class Figure {
5    public:
6      Figure() { cout << "Constructeur Figure." << endl; }
7      ~Figure() { cout << "Destructeur Figure." << endl; }
8      void dessin() { cout << "Dessin Figure ?" << endl; }
9  };
10 class Rectangle : public Figure {
11   public:
12     Rectangle() { cout << "Constructeur Rectangle" << endl; }
13     ~Rectangle() { cout << "Destructeur Rectangle" << endl; }
14     void dessin() { cout << "Dessin Rectangle !" << endl; }
15 };
```

La surcharge de fonctions

```
1  #include "overload.hpp"
2
3  int main() {
4    Figure f;
5    f.dessin();
6
7    Rectangle r;
8    r.dessin();
9    r.Figure::dessin();
10 }
```

Héritage et transtypage

Upcasting — retrouver la `Personne`

```cpp
#include "etudiant.hpp"

int main() {
  using namespace std;
  Etudiant e("Anthony", 12);
  cout << e << endl;
  cout << (Personne) e << endl;
  cout << Personne(e) << endl;
  {
    Personne p_cp = e;
    cout << p_cp << endl;
  }
  return 0;
}
```

Upcasting — retrouver la `Personne`

```cpp
#include "etudiant.hpp"

int main() {
  using namespace std;
  Etudiant e("Anthony", 12);

  Personne &p = e;
  cout << p << endl;
  cout << &p << endl;
  cout << &e << endl;
  return 0;
}
```

Downcasting — retrouver l'`Etudiant`

```cpp
#include "etudiant.hpp"

int main() {
  using namespace std;
  Etudiant e("Anthony", 12);

  // Conversion implicite
  Personne &p = e;
  //Etudiant &ep = p;
  // error: initialisation invalide pour une référence du type
       « Etudiant& »
  // à partir d'une expression de type « Personne »

  Etudiant &ep = static_cast< Etudiant & >(p);
  cout << ep << endl;

  return 0;
}
```

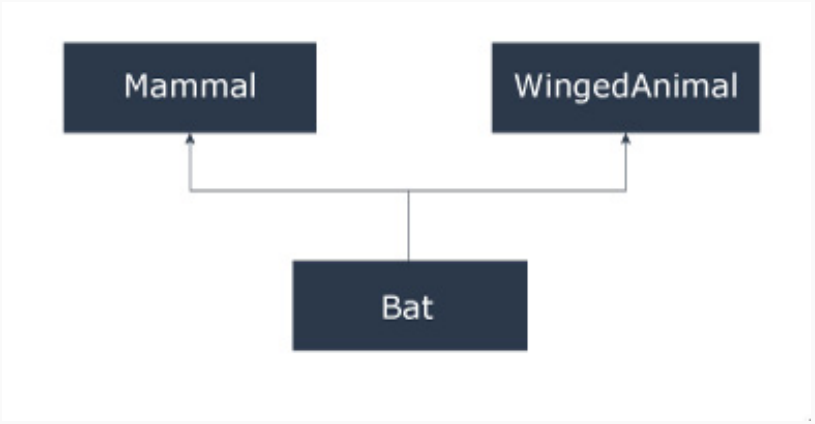
L’héritage multi{ple, -niveaux}

Héritage multi-niveaux

```
1  #ifndef SALARIE_HPP
2  #define SALARIE_HPP
3  #include "etudiant.hpp"
4
5  class Salarie : public Etudiant {
6    private:
7      long RSE;
8    public:
9      Salarie( std::string nom, long _INE, long _RSE ) :
10       Etudiant(nom, _INE), RSE(_RSE)
11       { std::cout << "Constructeur Salarie" << std::endl; }
12     ~Salarie()
13       { std::cout << "Destructeur Salarie" << std::endl; }
14 };
15 #endif
```

Héritage multiple

Une classe peut hériter de **plusieurs** classes.



Soulève de nombreux problèmes.

Exemple

```
1  #include <iostream>
2  using namespace std;
3  class Mammal {
4    public:
5      Mammal() { cout << "Constructeur Mammal." << endl; }
6      void f() { cout << "Fonction f() Mammal." << endl; }
7  };
8  class WingedAnimal {
9    public:
10     WingedAnimal() { cout << "Const. WingedAnimal." << endl; }
11     void f() { cout << "Fonction f() WingedAnimal" << endl; }
12 };
13 class Bat: public Mammal, public WingedAnimal {};
```

Problème engendré ?

Exemple

```cpp
#include <iostream>
using namespace std;
class Mammal {
  public:
    Mammal() { cout << "Constructeur Mammal." << endl; }
    void f() { cout << "Fonction f() Mammal." << endl; }
};
class WingedAnimal {
  public:
    WingedAnimal() { cout << "Const. WingedAnimal." << endl; }
    void f() { cout << "Fonction f() WingedAnimal" << endl; }
};
class Bat: public Mammal, public WingedAnimal {};
```

```cpp
#include "mammal.hpp"

int main() {
  Bat b;
  return 0;
}
```

Exemple

```cpp
#include <iostream>
using namespace std;
class Mammal {
  public:
    Mammal() { cout << "Constructeur Mammal." << endl; }
    void f() { cout << "Fonction f() Mammal." << endl; }
};
class WingedAnimal {
  public:
    WingedAnimal() { cout << "Const. WingedAnimal." << endl; }
    void f() { cout << "Fonction f() WingedAnimal" << endl; }
};
class Bat: public Mammal, public WingedAnimal {};
```

```cpp
#include "mammal.hpp"

int main() {
  Mammal m;
  WingedAnimal w;
  m.f();
  w.f();
  return 0;
}
```

Exemple

```cpp
#include <iostream>
using namespace std;
class Mammal {
  public:
    Mammal() { cout << "Constructeur Mammal." << endl; }
    void f() { cout << "Fonction f() Mammal." << endl; }
};
class WingedAnimal {
  public:
    WingedAnimal() { cout << "Const. WingedAnimal." << endl; }
    void f() { cout << "Fonction f() WingedAnimal" << endl; }
};
class Bat: public Mammal, public WingedAnimal {};
```

```cpp
#include "mammal.hpp"

int main() {
  Bat b;
  b.f();
  return 0;
}
```

Exemple

```cpp
#include <iostream>
using namespace std;
class Mammal {
  public:
    Mammal() { cout << "Constructeur Mammal." << endl; }
    void f() { cout << "Fonction f() Mammal." << endl; }
};
class WingedAnimal {
  public:
    WingedAnimal() { cout << "Const. WingedAnimal." << endl; }
    void f() { cout << "Fonction f() WingedAnimal" << endl; }
};
class Bat: public Mammal, public WingedAnimal {};
```

```cpp
#include "mammal.hpp"

int main() {
  Bat b;
  b.Mammal::f();
  b.WingedAnimal::f();
  return 0;
}
```

Un concept complexe

L'héritage en diamant

```
1  #include <iostream>
2  using namespace std;
3  class Animal {
4    public:
5      Animal() { cout << "Constructeur Animal." << endl; }
6  };
7  class Mammal : public Animal {
8    public:
9      Mammal() { cout << "Constructeur Mammal." << endl; }
10     void f() { cout << "Fonction f() Mammal." << endl; }
11 };
12 class WingedAnimal : public Animal {
13   public:
14     WingedAnimal() { cout << "Constructeur WingedAnimal." <<
           endl; }
15     void f() { cout << "Fonction f() WingedAnimal" << endl; }
16 };
17 class Bat: public Mammal, public WingedAnimal {}
18 };
```

Quand hériter ?

- `Rectangle` avec `h` et `l`, accesseurs et mutateurs.
- **Post-condition** : hauteur et largeur sont **librement** modifiables.
- `Carré` hérite de `Rectangle`
  mathématiquement cohérent : tout carré est un rectangle.

Le constat
Une instance de type `Carré` devrait être utilisable partout où un `Rectangle` est attendu.

Quand hériter ?

Le problème

- Dans un `Carré`, hauteur et largeur ne peuvent pas être **librement** modifiables.
- Si un `Carré` est utilisé là où un `Rectangle` est attendu : comportements **incohérents**.

Une mauvaise solution

Modifier les mutateurs de `Carré` : ne respecte pas la post-condition des mutateurs de `Rectangle` !