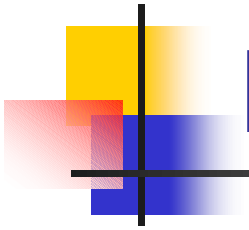
Programmation Orientée Objet

Héritage

Héritage



Exemple

Dans l'avion Paris-Berlin, on peut trouver les personnes suivantes :

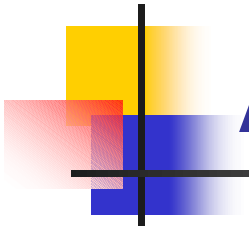
- Pierre : Pilote

  Paul : coPilote
- Anne : Hôtesse n°1

  Nathalie : Hôtesse n°2
- Laure : passager siège n°1

  Frédéric : passager siège n°2

  etc...



Attributs

- Chacune de ces personnes peut être représentée sous la forme d'objet.
- Chacun de ces objets appartient à une de ces catégories :
  - Pilote
  - Hôtesse
  - Passager

Organigramme

CPilote
•Prénom •N° de tel •Adresse •Age •Nbre d'heures de vole
•les constructeurs •les accesseurs •bool EstFatigue()

CHotesse
•Prénom •N° de tel •Adresse •Age •Nbre de langues
•les constructeurs •les accesseurs •bool ParlePlus3Langues()

CPassager
•Prénom •N° de tel •Adresse •Age •N° de son siège
•les constructeurs •les accesseurs •bool Chanceux()

Classe CPilote

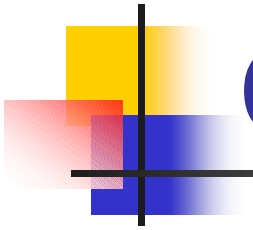
```
class CPilote
{
private:
    string Prenom;
    int NTel;
    string Adresse;
    int Age;
    int NbreHeureVole;
public:
    CPilote(...){...}
```

```
  string GetPrenom()
    {...}
  void SetPrenom(string prenom)
    {...}
   ...
   bool EstFatigue()
    {
      return (GetNbreHeureVole()>8);
    }
};
```

Classe CHotesse

```
class CHotesse
{
private:
    string Prenom;
    int NTel;
    string Adresse;
    int Age;
    int NbreLangues;
public:
    CHotesse(...){...}
```

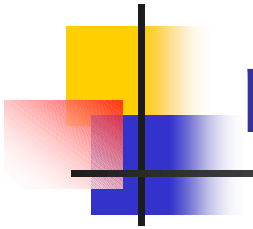
```
  string GetPrenom()
    {...}
  void SetPrenom(string prenom)
    {...}
   ...
   bool ParlePlus3Langues()
    {
        return (GetNbreLangues()>2 );
    }
};
```



Classe CPassager

```
class CPassager
{
private:
    string Prenom;
    int NTel;
    string Adresse;
    int Age;
    int NumSiege;
public:
    CPassager(...){...}
```

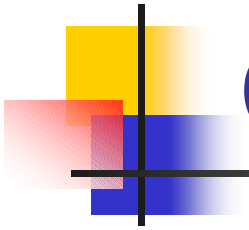
```
  string GetPrenom()
    {...}
  void SetPrenom(string prenom)
    {...}
   ...
   bool Chanceux()
    {
        return (GetNumSiege()!=13);
    }
};
```



main

Je peux maintenant créer mes objets :

```
{
        CPilote pilote("Pierre",…,5);
        CPilote coPilote("Paul",…,3);
        CHotesse hotesse1("Anne",…,4);
        CHotesse hotesse2("Nathalie",…,2);
        CPassager passager1("Laure",…,24);
        CPassager passager2("Frédéric",…,17);
        cout << pilote.GetPrenom() <<endl;
        cout << passager2. Dort() <<endl;
}
```



Ouf, enfin fini !!!

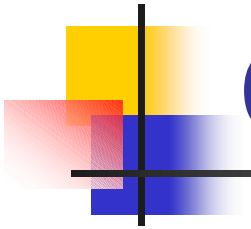
N'aurait on pas pu gagner du temps en remarquant et en exploitant que ces 3 classes avaient des attributs et des méthodes communes ?

Organigramme

CPilote
•Prénom •N° de tel •Adresse •Age •Nbre d'heures de vole
•les constructeurs •les accesseurs •bool EstFatigue()

CHotesse
•Prénom •N° de tel •Adresse •Age •Nbre de langues
•les constructeurs •les accesseurs •bool ParlePlus3Langues()

CPassager
•Prénom •N° de tel •Adresse •Age •N° de son siège
•les constructeurs •les accesseurs •bool Chanceux()



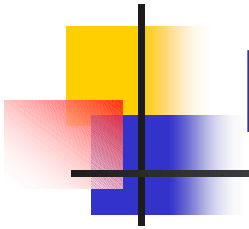
CPersonne

Ces caractéristiques communes peuvent représenter une personne :

CPersonne
•Prénom •N° de tel •Adresse •Age
•les constructeurs •les accesseurs

Classe CPersonne

```
class CPersonne
{
private:
    string Prenom;
    int NTel;
    string Adresse;
    int Age;

public:
    CPersonne(...){...}
    string GetPrenom(){...}
    void SetPrenom(string pren){...}
    int GetNTel(){...}
    void SetNTel(int nTel){...}
    string GetAdresse(){...}
    void SetAdresse(string adr){...}
    int GetAge(){...}
    void SetAge(int age){...}
};
```



Factorisation

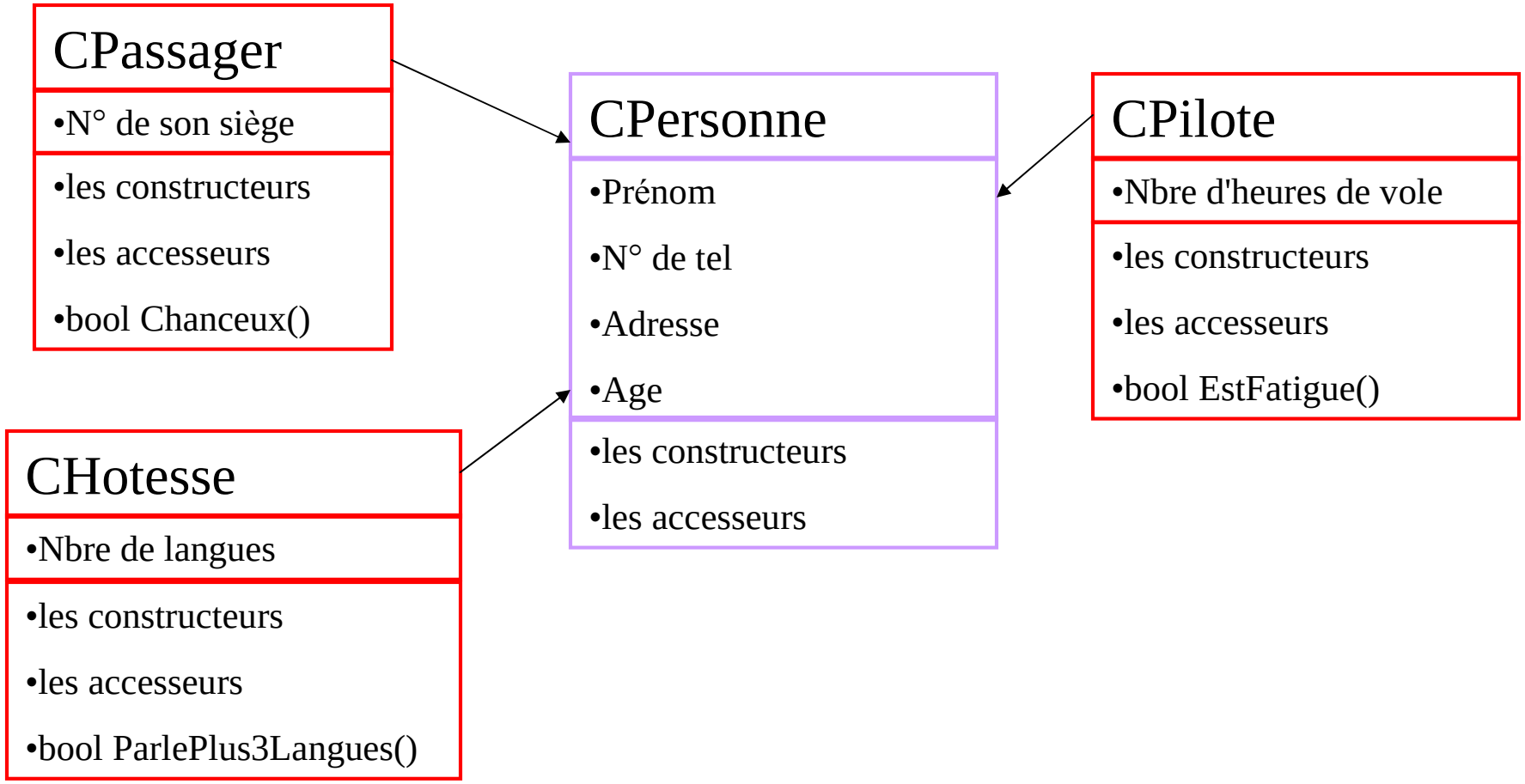
Maintenant, on peut dire que :

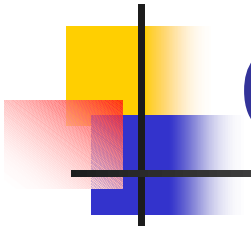
- un pilote est une personne
- une hôtesse est aussi une personne
- un passager est aussi une personne.

Donc

un pilote, une hôtesse et un passager possèdent aussi un prénom, un n° de téléphone, une adresse, et un age.

Organigramme





Classe CPilote

```
class CPilote : public CPersonne
{
private:
    int NbreHeureVole;
public:
    CPilote(...){...}

    int GetNbreHeureVole()
        {return NbreHeureVole;}
    void SetNbreHeureVole(int nb)
        {NbreHeureVole=nb;}
```

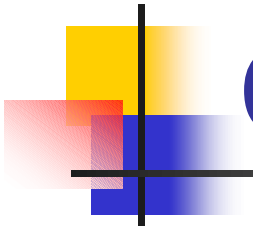
```
    bool EstFatigue()
    {
      return (GetNbreHeureVole()>8);
    }
};
```

Héritage

La classe Fille hérite des attributs et des méthodes de la classe Mère :

```
class Mère
{



};
```

```
class Fille : public
   Mère
{



};
```



Classe CHotesse

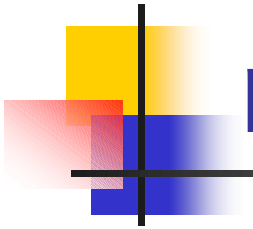
```
class CHotesse : public CPersonne
{
private:
    int NbreLangues;
public:
    CHotesse(...){...}

    int GetNbreLangues()
        {return NbreLangues}
    void SetNbreLangues(int nbre)
        {NbreLangues=nbre;}
```

```
    bool ParlePlus3Langues()
    {
        return (GetNbreLangues()>2
    );
    }
};
```

Classe CPassager

```
class CPassager : public
    CPersonne
{
private:
    int NumSiege;
public:
    CPassager(...){...}

    int GetNumSiege()
        {return NumSiege}
    void SetNumSiege(int num)
        {NumSiege=num;}
```

```
    bool Chanceux()
    {
        return (GetNumSiege()!=13 );
    }
};
```



main

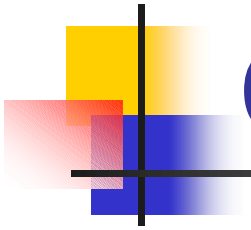
A l'usage, rien n'a changé :

```
{
        CPilote pilote("Pierre",…,5);
        CPilote coPilote("Paul",…,3);
        CHotesse hotesse1("Anne",…,4);
        CHotesse hotesse2("Nathalie",…,2);
        CPassager passager1("Laure",…,24);
        CPassager passager2("Frédéric",…,17);
        cout << pilote.GetPrenom() <<endl;
        cout << passager2. Dort() <<endl;
}
```

Constructeur de CPilote

```
CPilote(string prenom, int nTel, string adresse, int age,
            int nbreHeure)
{
    SetPrenom(prenom);
    SetNumTel(nTel);
    SetAdresse(adresse);
    SetAge(age);
    SetNbreHeureVole(nbreHeure);
}
```

Les constructeurs de CHotesse et de CPersonne divergent de celui de CPilote sur le dernier paramètre.



Constructeur de CPersonne

Pour CPersonne, on a besoin d'initialiser que ses attributs

```
CPersonne(string prenom, int nTel, string adresse, int age)
{
    SetPrenom(prenom);
    SetNumTel(nTel);
    SetAdresse(adresse);
    SetAge(age);
}
```

Héritage du constructeur

J'utilise le constructeur de CPersonne pour m'aider à "construire" CPilote.

```
CPilote(string prenom, int nTel, string adresse, int age, int
    nbreHeure)
    : CPersonne(prenom,nTel,adresse,age)
{
    SetNbreHeureVole(nbreHeure);
}
```

Exemple d'usage :

```
CPilote pilote1("Pierre",0321175413,"home",54,9);
```

Héritage

La classe Fille hérite de la classe Mere :

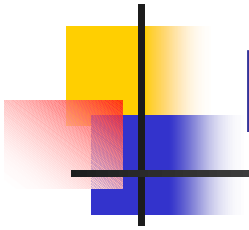
```
class Mere
{
   Mere(type1 par1,
        type2 par2)
   {
      ...
   }
};
```

```
class Fille : public Mère
{
   Fille(type1 par1,
         type2 par2,...)
      : Mere(par1,par2)
   {
      ...
   }
};
```

Héritage du constructeur

```
CHotesse(string prenom, int nTel, string adresse, int age, int nbreLangue)
    : CPersonne(prenom,nTel,adresse,age)
{
    SetNbreLangue(nbreLangue);
}
```

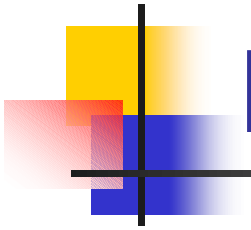
```
CPassager(string prenom, int nTel, string adresse, int age, int numSiege)
    : CPersonne(prenom,nTel,adresse,age)
{
    SetNumSiege(numSiege);
}
```



Public/Private

public: Tous les attributs ou fonctions situés sous le mot clé sont accessibles en dehors de l'objet ou depuis n'importe quelles fonctions de la classe.

private: Tous les attributs ou fonctions situés sous le mot clé ne sont accessibles que depuis les fonctions de l'objet.



Protected

Tous les attributs ou fonctions situés sous le mot clé protected: ne sont accessibles que depuis les méthodes de la classe mère et de ses filles.

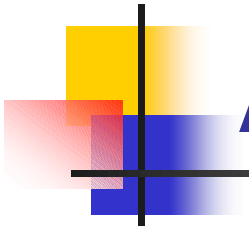
Remarque : c'est une sorte de private: étendue aux classes filles.

Classe CPersonne

Ø Supposons qu'on ajoute un protected à la classe CPersonne

```
class CPersonne
{
private:
    string Prenom;
    int NTel;
    string Adresse;
    int Age;

public:
    CPersonne(...){...}
```

```
protected:
    string GetPrenom(){...}
    void SetPrenom(string pren){...}
    int GetNTel(){...}
    void SetNTel(int nTel){...}
    string GetAdresse(){...}
    void SetAdresse(string adr){...}
    int GetAge(){...}
    void SetAge(int age){...}
};
```



Accessibilité

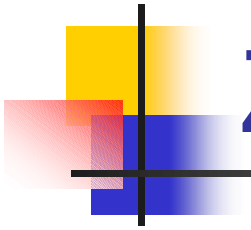
Les méthodes de la classe CPersonne:

GetPrenom, SetPrenom, GetNTel, SetNTel, GetAdresse, SetAdresse, GetAge, SetAge

ne sont accessibles que dans

- CPersonne, CPilote, CHotesse, CPassager
- Toutes classes qui héritent de CPilote, CHotesse et CPassager
- et ainsi de suite....

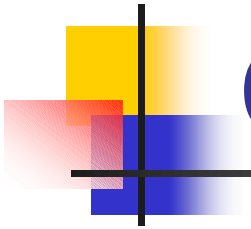
Héritage Multiple



Zoo

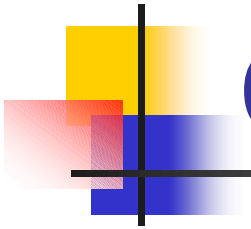
Supposons la gestion d'un zoo.

- Pour chaque animal, on décide de conserver son nom ainsi que son age.
- En plus, notre zoo gère quelques espèces protégées. Pour chacun d'eux, on conserve un numéro d'identification ainsi que le nombre d'individu encore vivant sur Terre.
- Enfin, chaque espèce possède ses propres caractéristiques.



Conception

- Comment gérer "simplement" :
  - un dauphin
  - un panda ⇒ espèce protégée
  - ...
- ⇒ héritage : créer une classe contenant le prénom, l'age et les informations sur l'espèce protégée.
- ⇒ Chaque classe (pour chaque espèce) en hérite.



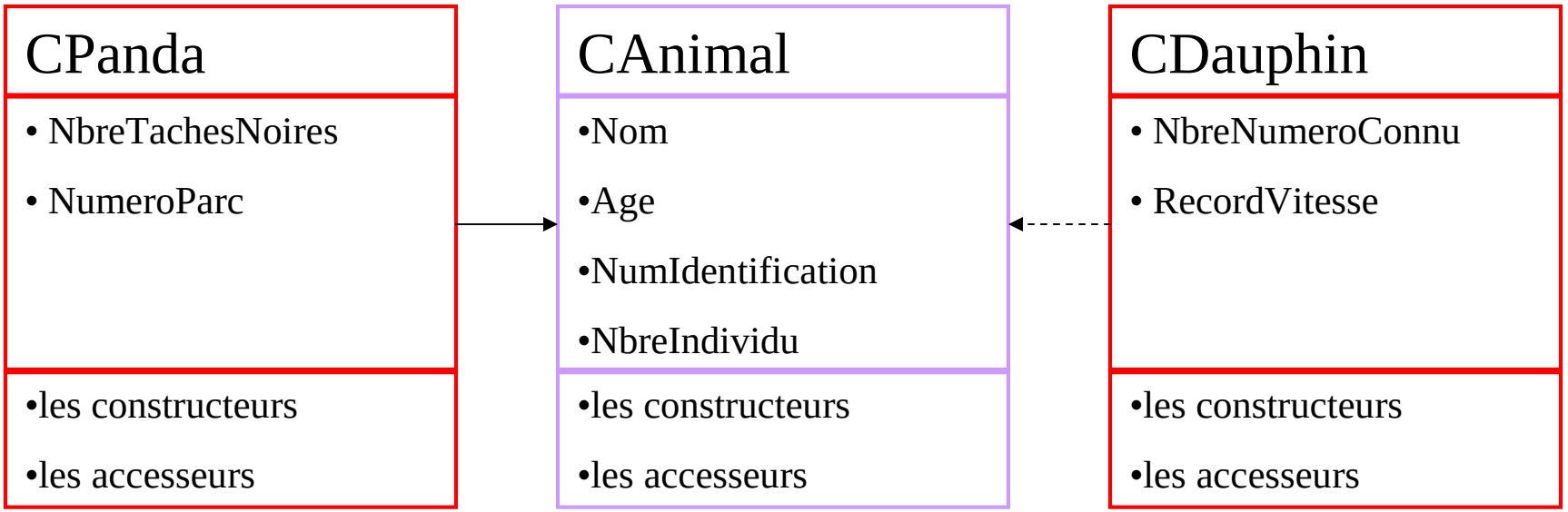
CAnimal

Un animal peut être représenté ainsi

CAnimal
•Nom •Age •NumIdentification •NbreIndividu
•les constructeurs •les accesseurs

CPanda/CDauphin

Nos classes CPanda et CDauphin n'ont plus qu'à hériter



Problème : un dauphin n'a pas de numéro d'espèce protégée

Meilleure Solution

On peut dire que

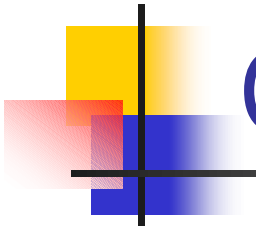
- Un Dauphin est un Animal
- Un Panda est un Animal - Protégé

⇒ On peut extraire de CAnimal les informations sur l'espèce protégée

⇒ On obtient alors 2 classes.

CAnimal
•Nom •Age
•les constructeurs •les accesseurs

CProtege
•NumIdentification •NbreIndividu
•les constructeurs •les accesseurs

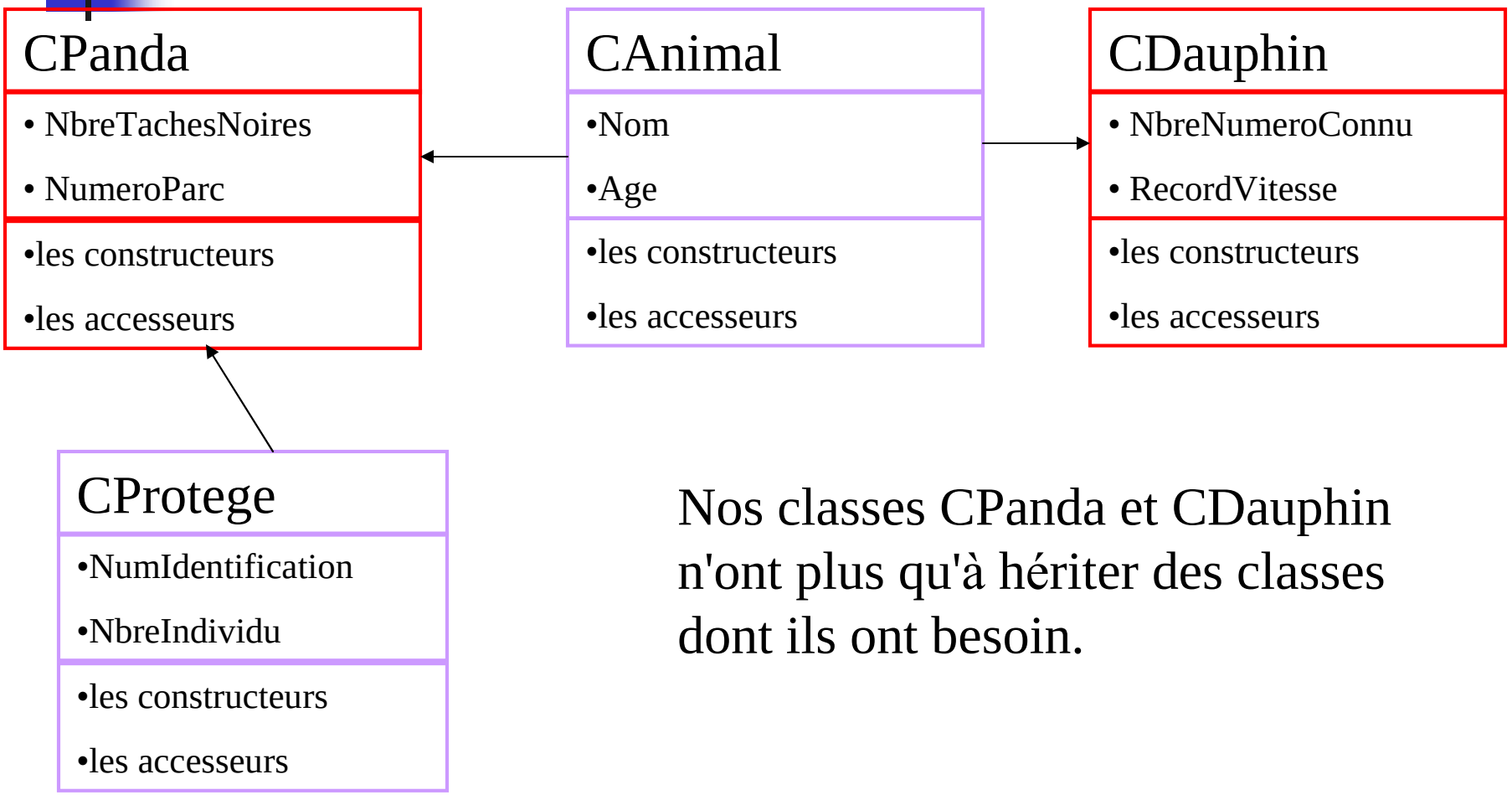


CAnimal/CProtege

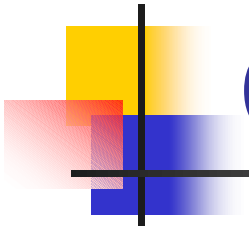
```
class CAnimal
{
protected:
        string Nom;
        int Age;
public :
        ...
};
```

```
class CProtege
{
protected:
        int NumIdentification;
        int NbreIndividu;
public :
        ...
};
```

CPanda/CDauphin



Nos classes CPanda et CDauphin n'ont plus qu'à hériter des classes dont ils ont besoin.



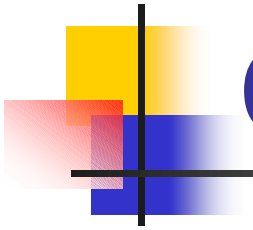
Class CPanda

```
class CDauphin : public CAnimal, public CProtege
{
   private:
        int NbreTachesNoires;
        int NumeroParc;
   public:
        ...
};
```

Héritage Multiple

La classe Fille hérite des attributs et des méthodes des classes $Mère_1$, $Mère_2$, $Mère_3$,…$Mère_n$

```
class Fille : public Mère1, public Mère2..., public
   Mèren
{
   ...

};
```

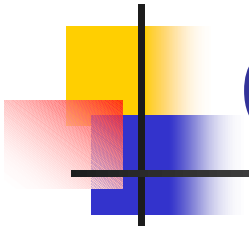


Constructeurs

Supposons les constructeurs suivant pour CAnimal et CProtege

```
CAnimal(string nom, int age)
{
    SetNom(nom);
    SetAge(age);
}
```

```
CProtege(int numId, int nbreIndividu)
{
   SetNumIdentification(numId);
   SetNbreIndividu(nbreIndividu);
}
```



Constructeur de CPanda

Ainsi, pour CPanda, on a besoin d'initialiser que ses attributs

```
CPanda(string nom, int age, int numId, int nbreVivant, nbreTache,
            int numParc)
    :CAnimal(nom,age),CProtege(numId,nbreVivant)
{
    SetNbreTachesNoires(nbreTache);
    SetNumeroParc(numParc);
}
```

Exemple d'usage :

```
CPanda panda ("popo",7,147896,64,11,4);
```

Constructeur multiple

La classe Fille hérite des classes $Mère_1$, $Mère_2$, $Mère_3$, … $Mère_n$

```
class Fille : public Mère
{
    Fille(type_a par_11,..., type_b par_1m, ..., type_c par_n1,... , type_d par_nl,
    ...)
            : Mère_1(par_11,...,par_1m) ,..., Mère_n(par_n1,...,par_nl)
    {
            ...
    }
};
```