ECE PARIS
ÉCOLE D'INGÉNIEURS

Robin FERCOQ
2018-2019

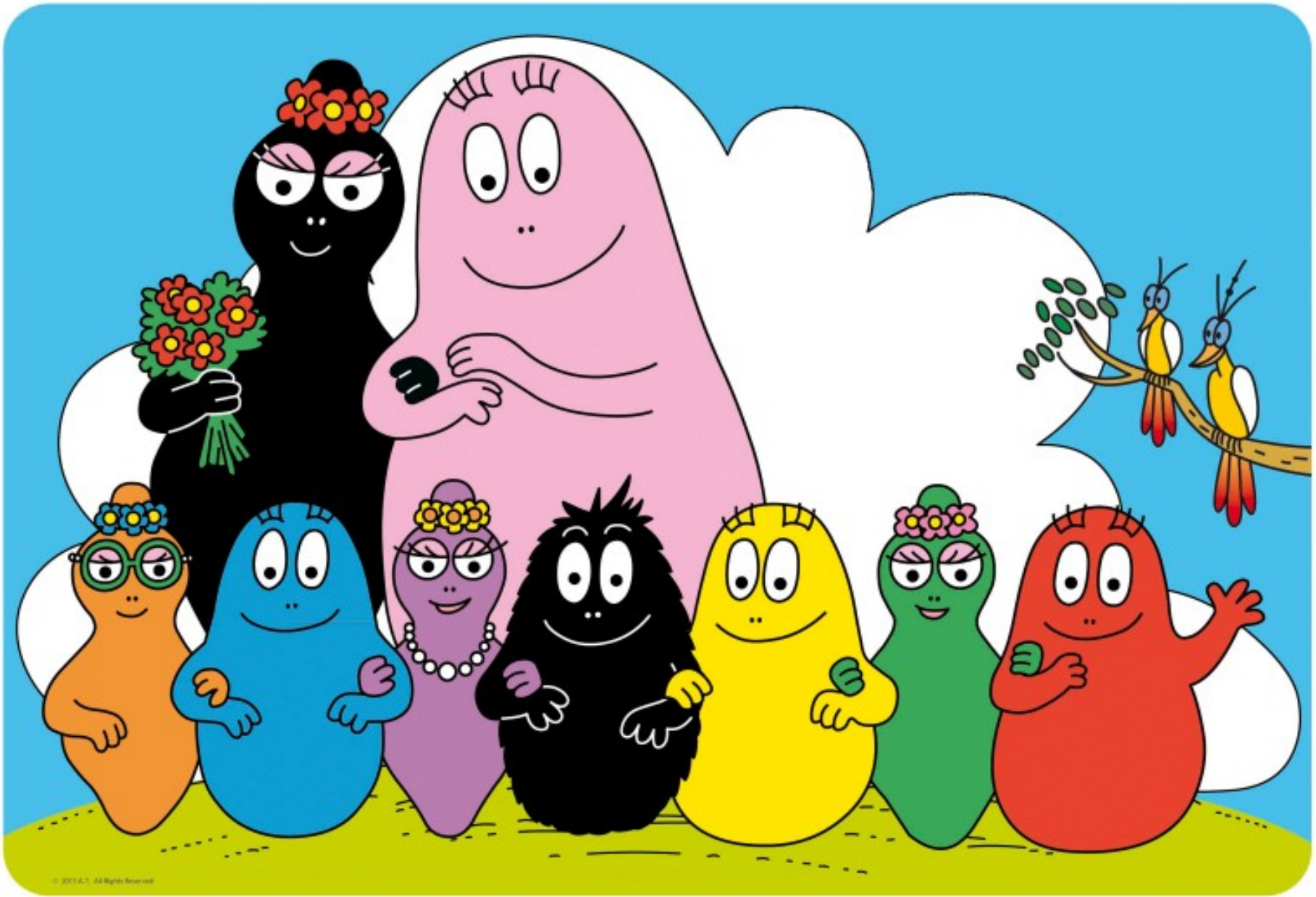
Conception et Programmation Orientée Objet C++

POO - C++

Sommaire général du semestre

COURS	Semaine suivante →	**TPs**
1. *Intro, concepts, 1 exemple*		1. *Organisation objet des données*
2. *Modélisation objet / UML*		2. *Diagrammes de classe UML*
3. *C++ pratique 1*		3. *C++ pratique, E/S, string, vector*
4. *C++ pratique 2*		4. *C++ pratique, type &, surcharge*
5. *Classes & C++ : bases*		5. *Date : une classe simple en C++*
6. *Classes & C++ : compléments*		6. *UML et C++, associations*
7. *Conteneurs & C++ : la STL*		7. *Gestion de collections complexes*
8. *Héritage / polymorphisme*		8. *Collections polymorphes*
9. *Modèles objets avancés*		9. *Modèle composite et graphismes*
10. *Exceptions, flots, fichiers ...*		10. *Persistance / fichiers / except.*
11. *Templates côté développeur*		11. *Développement de templates*
12. *Gestion mémoire / smarts ptrs*		12. *Soutenance de **projet** ...*

Héritage / polymorphisme



COURS 8

A) Héritage simple, présentation
B) Héritage simple en C++
C) Upcasting, slicing
D) Virtuel & polymorphisme

COURS 8

A) Héritage simple, présentation
B) Héritage simple en C++
C) Upcasting, slicing
D) Virtuel & polymorphisme

Héritage simple, présentation

Zebra

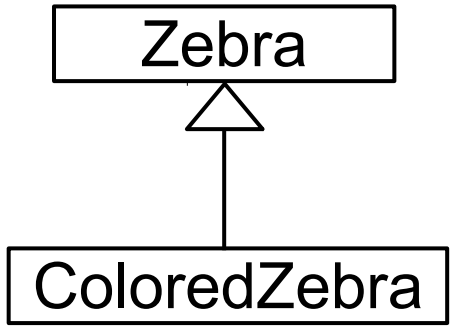
ColoredZebra

TalkingZebra

Héritage simple, présentation

- *L'**héritage** est un concept de programmation orientée objet : une **relation entre 2 classes***

- *Classe **de base** ou classe **mère***
- *Classe **dérivée** ou classe **fille***



- ***Très différent d'une association :***
  il n'y a pas 2 objets (instances) impliqués
- *Une classe fille **spécialise** la classe mère*
- *La classe mère **généralise** une classe fille*
  *(plutôt quand il y a plusieurs : généralise **des** classes filles)*

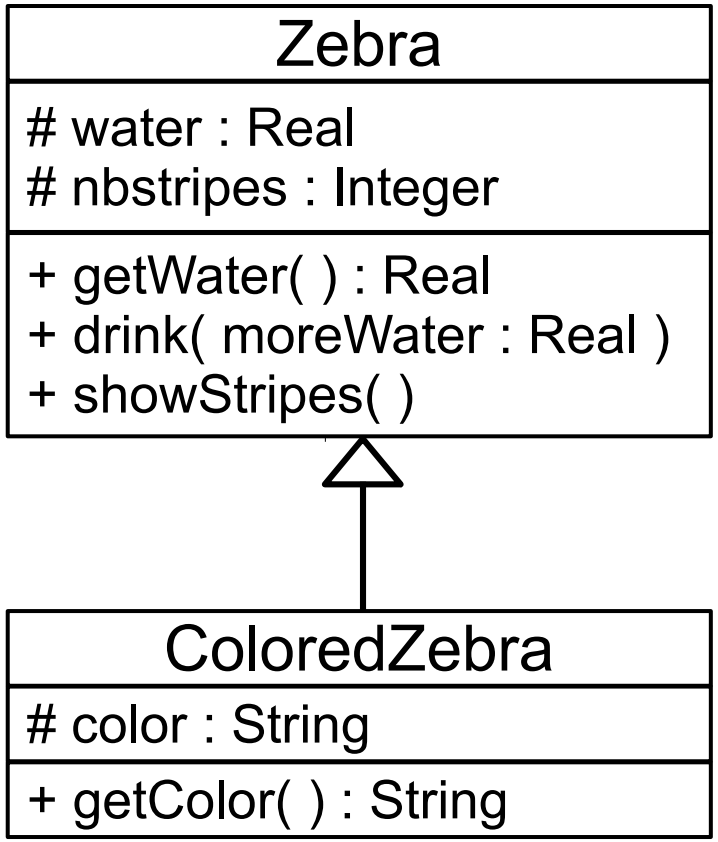
Héritage simple, présentation

- *Les **attributs** et **méthodes** de la classe mère sont automatiquement **hérités** par la classe fille*
- *On ne les répète pas, ni en UML, ni en code*

Chaque objet Zebra a
→ des attributs
→ des méthodes

Héritage des membres

Chaque objet ColoredZebra a
→ les attributs Zebra + les siens
→ les méthodes Zebra + les siennes



Héritage simple, présentation

- *Une même classe peut servir plusieurs fois de classe de base*
- *La classe fille n'ajoute pas forcément d'attribut*

Chaque objet Zebra a
→ des attributs
→ des méthodes

Héritage des membres

Chaque objet TalkingZebra a
→ les attributs Zebra **c'est tout !**
→ les méthodes Zebra + les siennes

Héritage simple, présentation

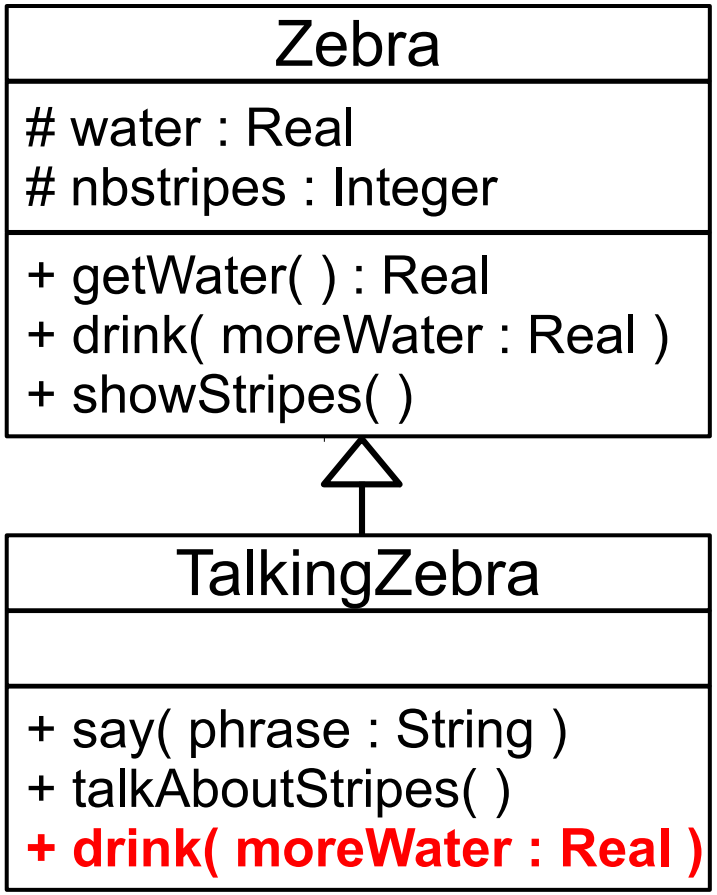
- *La classe fille **ne peut pas remplacer/enlever des attributs** par rapport à la classe mère*
- *Mais la classe fille peut **redéfinir** des méthodes*

Ici un objet TalkingZebra a une **méthode drink spécifique** :
par rapport à la classe mère,
la méthode drink a été redéfinie

overriding

≠ overloading

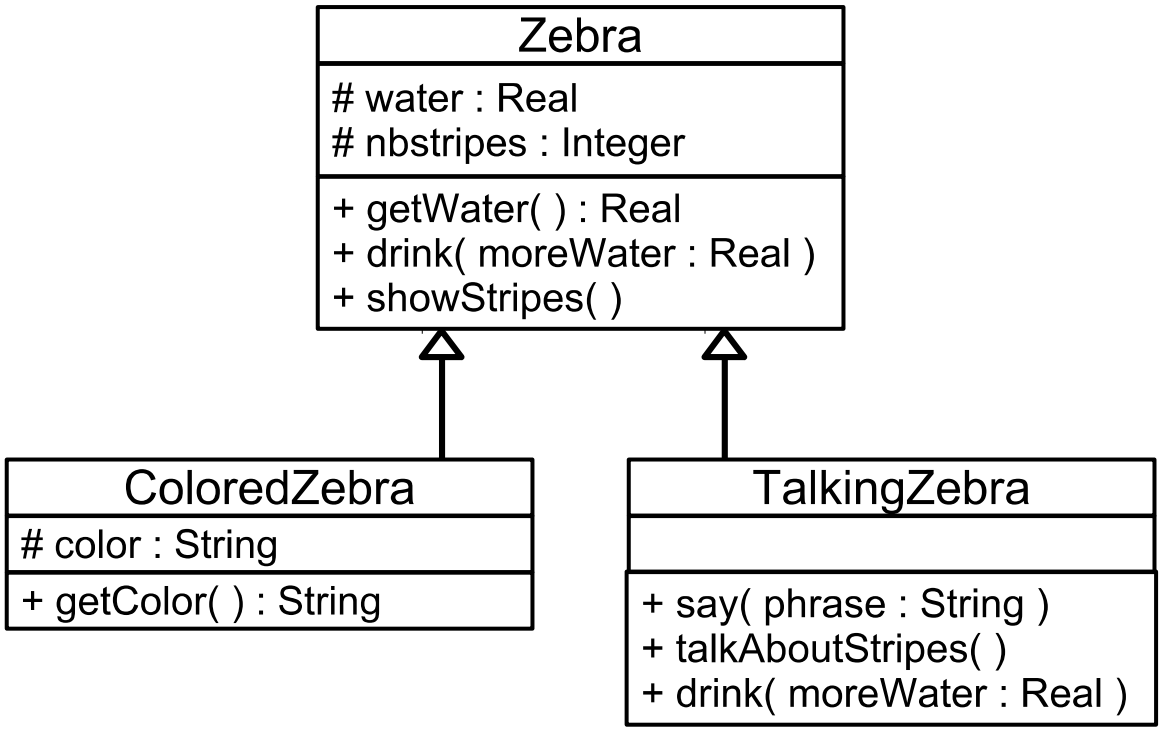
Concrètement l'appel à drink pour un TalkingZebra exécutera un autre code que drink de Zebra



Héritage simple, présentation

- *Une classe mère peut (ce n'est pas obligé !) avoir plusieurs classes filles (**≠héritage multiple**)*
- *Les classes sœurs n'ont **pas de relation spéciale***

Strictement équivalent au schéma slide suivant

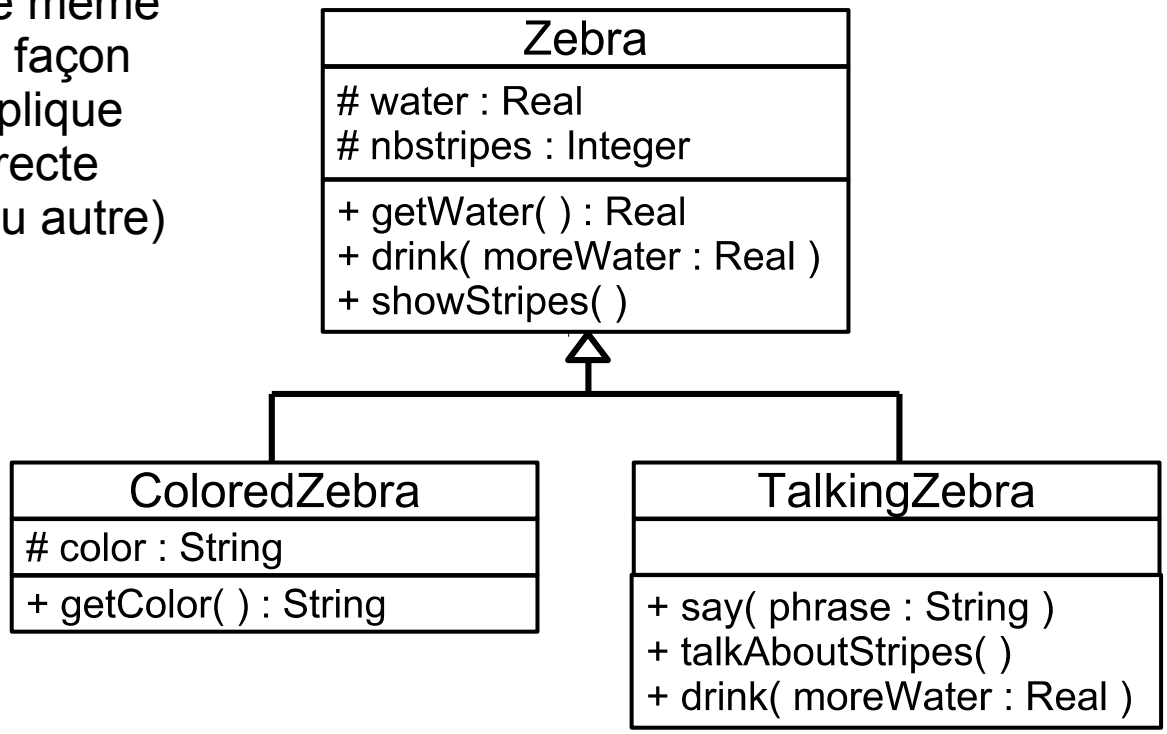


Héritage simple, présentation

- *Une classe mère peut (ce n'est pas obligé !) avoir plusieurs classes filles (**≠héritage multiple**)*
- *Les classes sœurs n'ont **pas de relation spéciale***

Cette **façon de dessiner** l'héritage de plusieurs classes filles d'une même classe mère est la façon **usuelle** : elle n'implique aucune relation directe (de dépendance ou autre) entre les 2 filles.

Strictement équivalent au schéma slide précédent

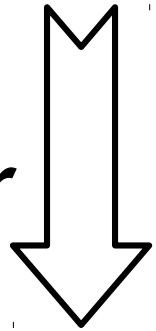


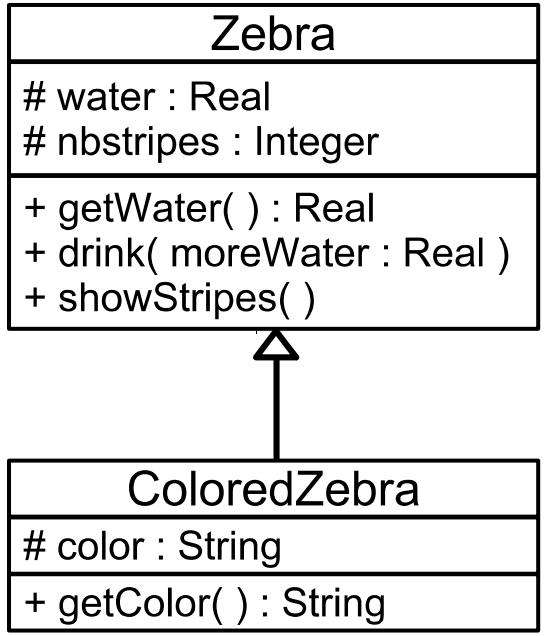
Héritage simple, présentation

- ***Usage 1*** *: réutiliser, modifier « à la carte »*
  - *J'ai déjà une classe Zebra, il me faut des zèbres de couleur, mais j'aurai encore besoin de zèbres sans couleur (je ne veux pas modifier Zebra)*

Sans modifier la classe mère
Sans recoder l'existant

Spécialiser
Customiser



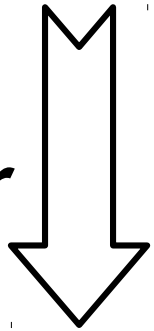


Héritage simple, présentation

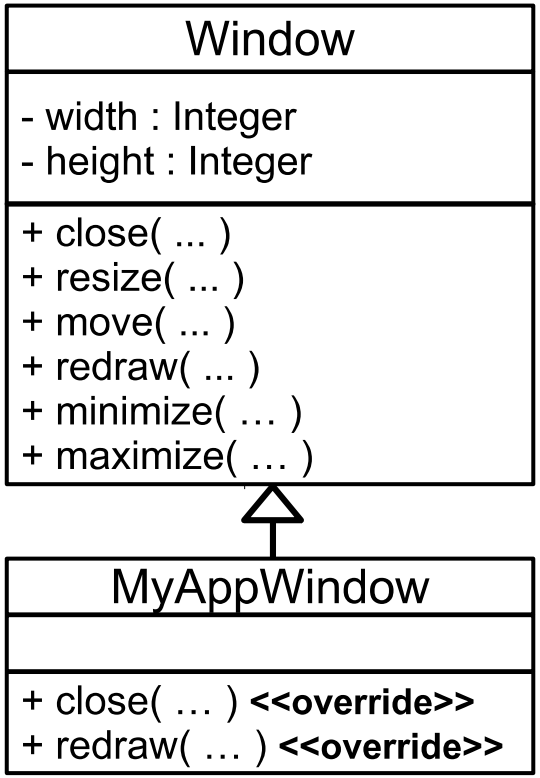
- ***Usage 1*** *: réutiliser, modifier « à la carte »*
- *C'est particulièrement utile quand on a des grosses classes de* ***bibliothèques*** *(on ne veut/peut pas accéder à la classe de base)*

Sans modifier la classe mère
Sans recoder l'existant
On utilise tout ce qui convient
On change juste ce qu'il faut !

Spécialiser
Customiser



Typiquement on redéfinit quelques méthodes : overriding

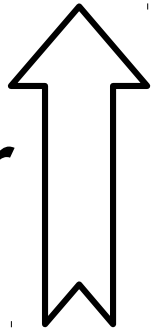


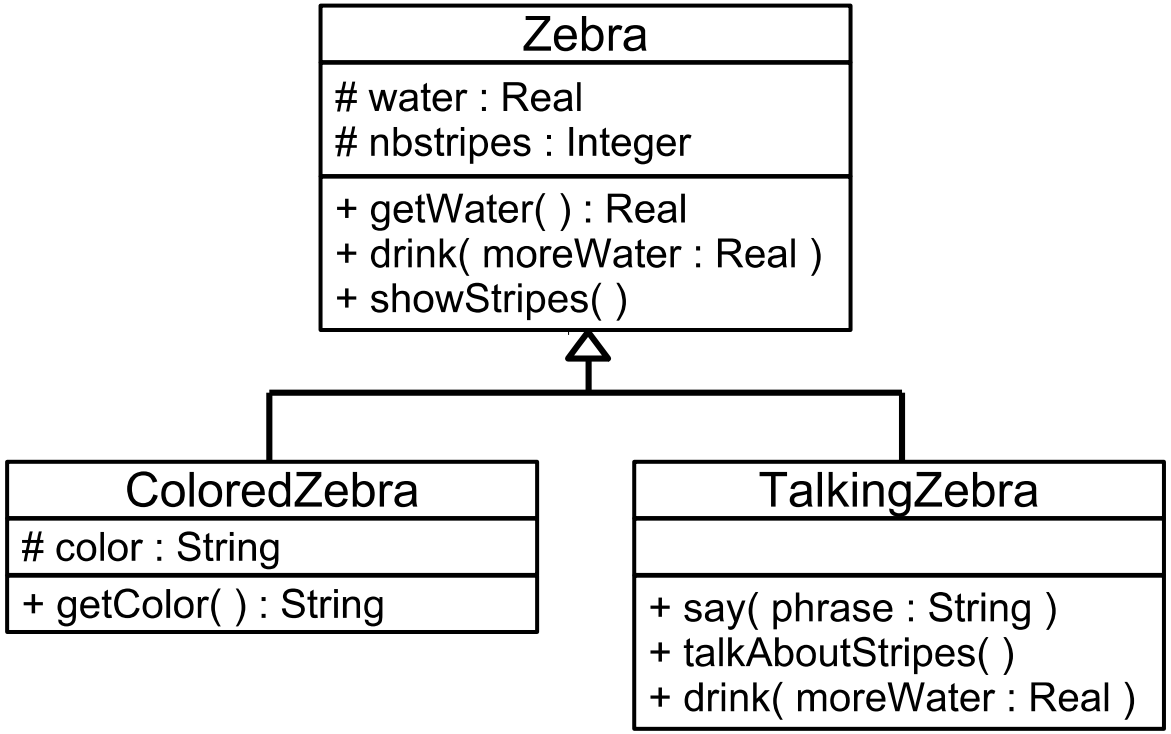
Héritage simple, présentation

- ***Usage 2*** *: factoriser, ne pas dupliquer du code*
  - *J'ai déjà une classe ColoredZebra, il me faut aussi des zèbres qui parlent, mais je ne veux pas recoder toute la « zèbritude » en commun*

Sans dépendre d'une sœur
Sans recoder l'existant
Identifier une classe mère

Généraliser



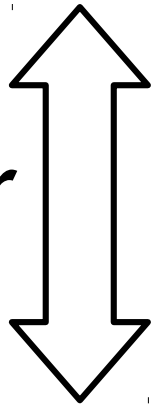


Héritage simple, présentation

- ***Usage 3*** *: modulariser, séparer les niveaux*
- ➔ *Au final je veux des zèbres colorés télépathes, je peux développer séparément et garder séparés zèbritude / coloration / télépathie*

Séparer en modules
3 fois 100 lignes de code

Organiser

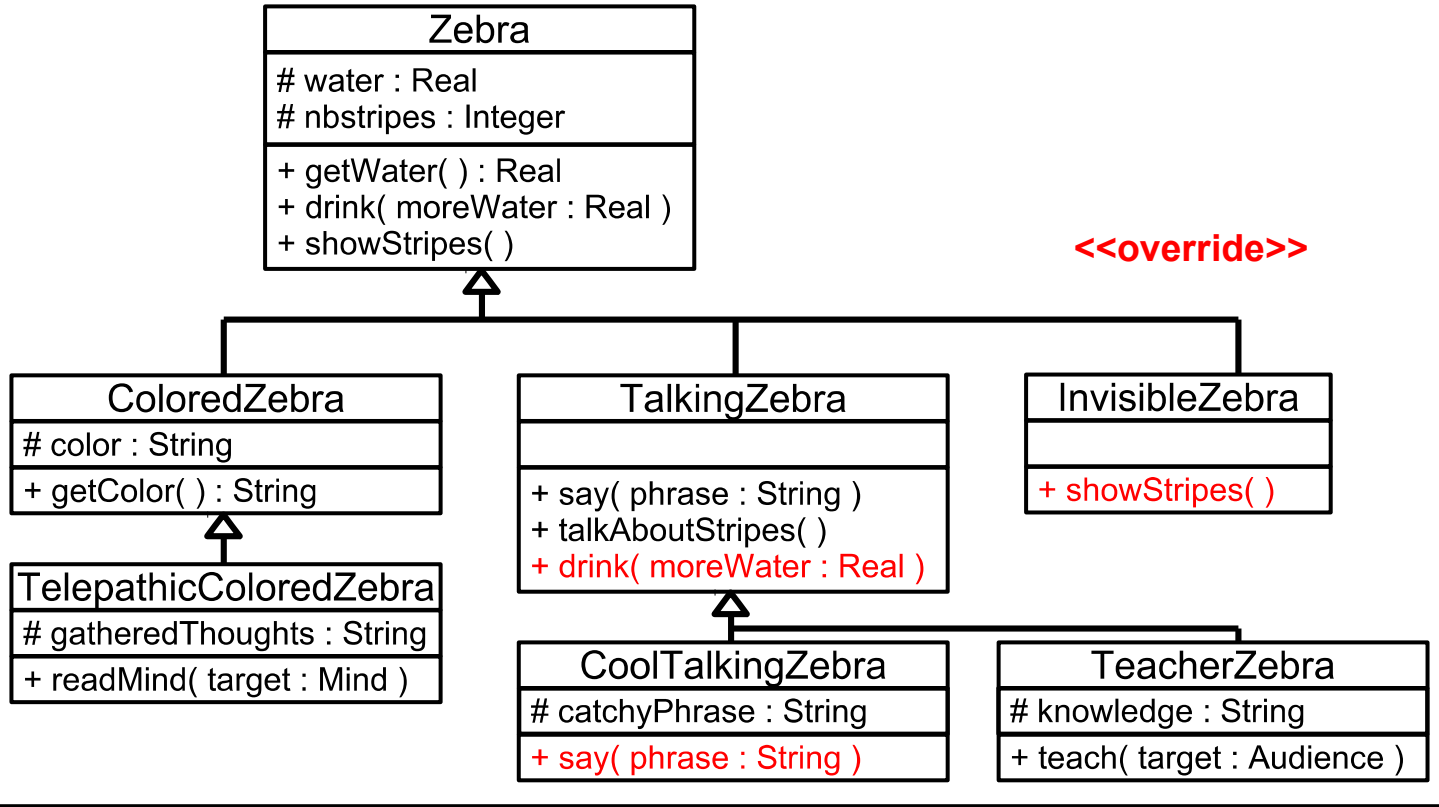


Héritage simple, présentation

- ***Usage 3*** *: modulariser,* ***hiérarchiser***
- ➔ *Au final je veux différentes variantes de zèbres plus ou moins spécialisées : on arrive à une* ***hiérarchie*** *ou* ***arbre d'héritage***

Hiérarchiser

Organiser

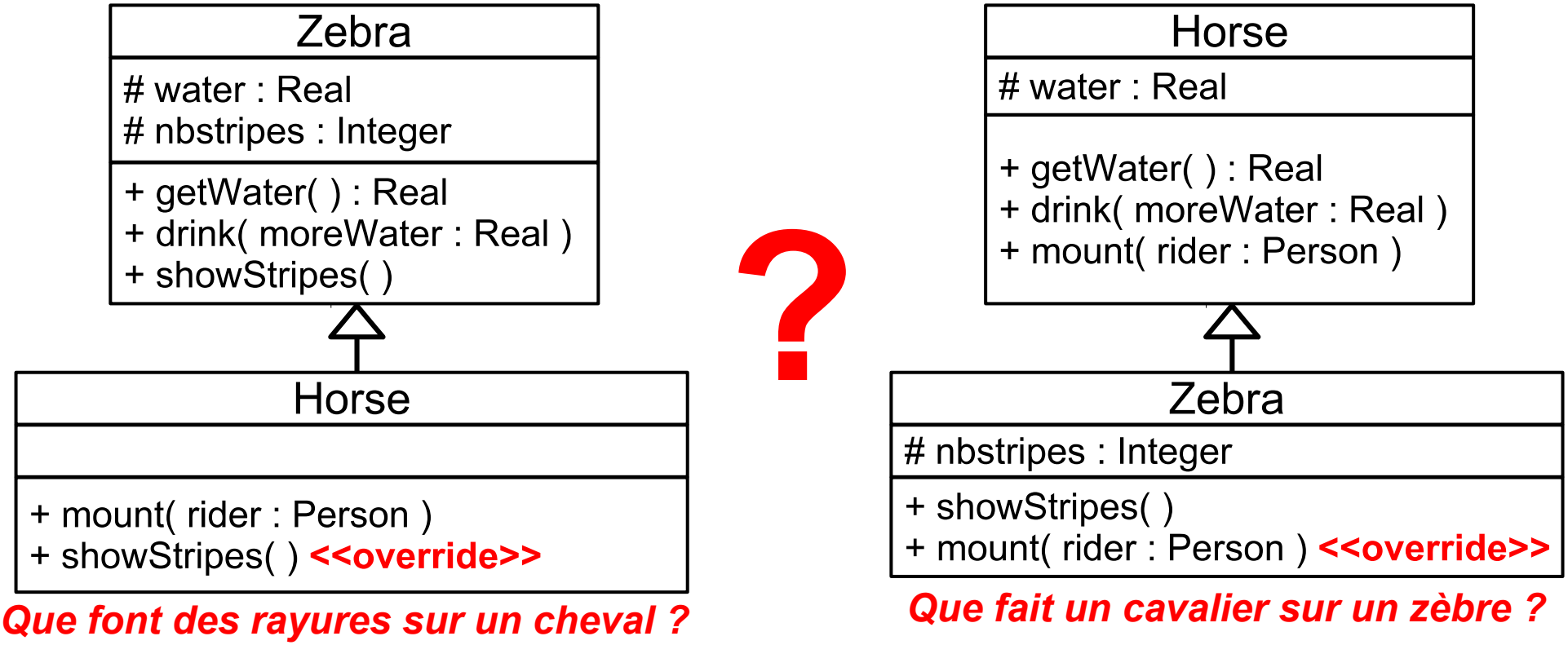


Héritage simple, présentation

- ***Usages mauvais***
- ➔ *Attention, si on ne retient que l'aspect « réutilisation de code existant » il est* ***facile de mal utiliser l'héritage***
- *Exemple : il se trouve que j'ai déjà développé une classe TalkingZebra, rien d'autre. D'un coup j'ai besoin d'une classe cheval (une urgence, le patron met la pression) …*
- *J'hérite Horse de Zebra ? Que faire des rayures ?*
- *Je renomme Zebra en Horse, et j'hérite Zebra de Horse en ajoutant des rayures ? Que faire du cavalier que Zebra accepte maintenant ?*

Héritage simple, présentation

- ***Usages mauvais***
- ➔ *Attention, si on ne retient que l'aspect « réutilisation de code existant » il est* ***facile de mal utiliser l'héritage***



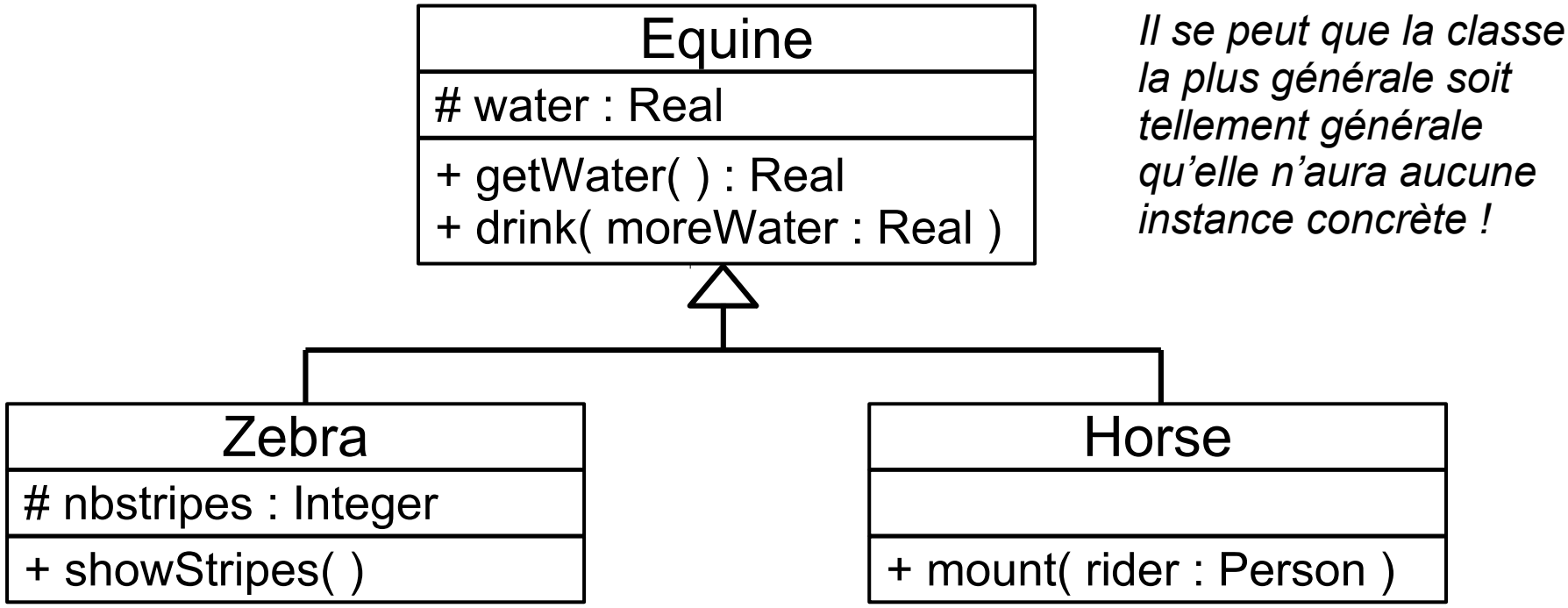
Héritage simple, présentation

- ***Usages mauvais***

Héritage simple, présentation

- ***Usage correct***

Equine = *équidé* = *famille des chevaux, des ânes, des zèbres*



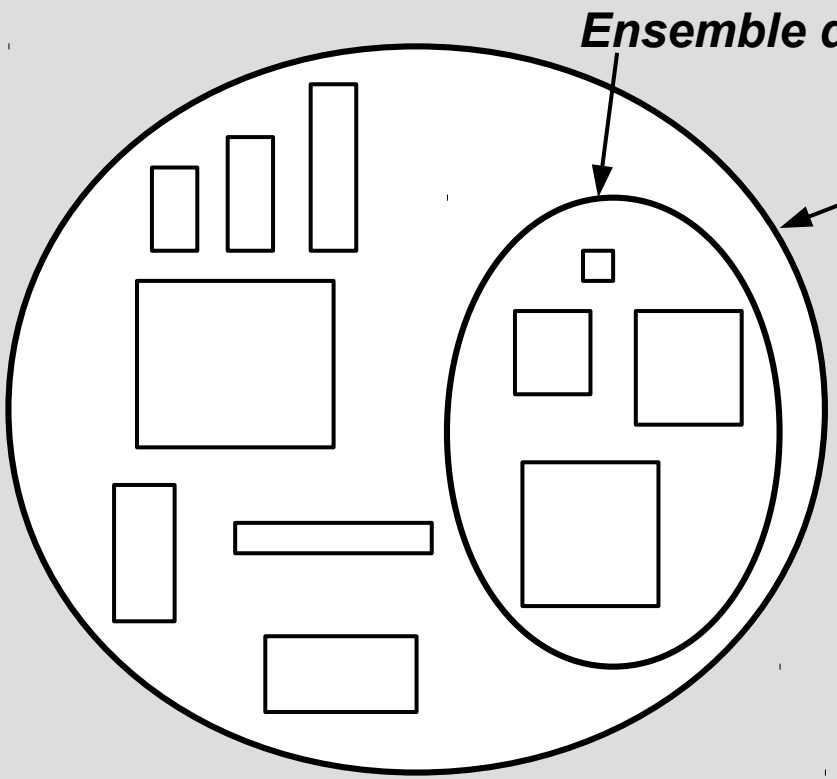
Il se peut que la classe la plus générale soit tellement générale qu'elle n'aura aucune instance concrète !

Héritage simple, présentation

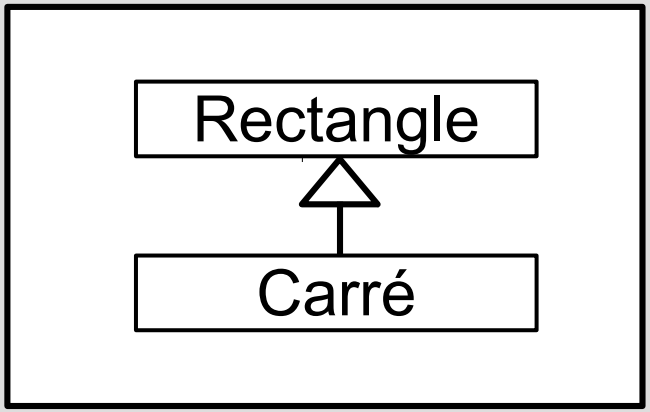
- ***Usages mauvais / usages corrects***
- *Au delà de l'**aspect technique** « héritage des attributs et méthodes de la classe parente », l'héritage a une **sémantique** de conception : **spécialisation / généralisation***
- *Dire **«** classe-dérivée **est une** classe-parente **»** ou « classe-dérivée **est une sorte de** classe-parente » doit faire sens sinon on ne comprend plus rien*
- *Dire « un cheval **est un** équidé » fait sens*
- *Dire « un cheval est un zèbre » ne fait pas sens*
- *Dire « un zèbre est un cheval » ne fait pas sens*

Héritage simple, présentation

- ***Usages mauvais d'un autre genre !***
  - *Attention, à l'inverse si on ne retient que l'aspect « sémantique sous-ensemble⊂ensemble » il est* ***encore facile de mal utiliser l'héritage***



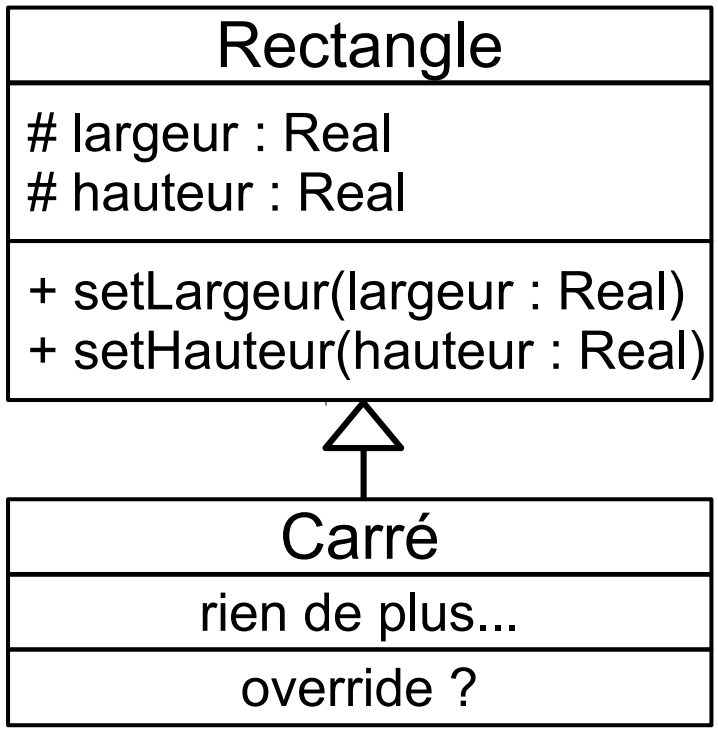
Un carré **est un** rectangle :
automatiquement l'héritage est bon ?



Informatique ≠ Maths, réfléchissons !

Héritage simple, présentation

- ***Usages mauvais d'un autre genre !***
- ➔ *Attention, à l'inverse si on ne retient que l'aspect « sémantique sous-ensemble⊂ensemble » il est* ***encore facile de mal utiliser l'héritage***



Que faire de 2 attributs hérités quand l'objet spécialisé n'en demande qu'un ?

Attention l'héritage informatique n'est pas juste de la théorie des ensembles, c'est un procédé essentiellement additif éventuellement transformatif (override) mais jamais soustractif

Ici on est embêtés, l'héritage marche mal, Pas de solution toute faite, voir le CDC : à quoi vont servir ces rectangles et carrés ? …

Héritage simple, présentation

Composition vs. Inheritance: How to Choose ?

There is no substitute for object modeling and critical design thinking. But if you must have some guidelines, consider these -

Inheritance should only be used when:

1. Both classes are in the same logical domain
2. The subclass is a proper subtype of the superclass
3. The superclass's implementation is necessary or appropriate for the subclass
4. The enhancements made by the subclass are primarily additive.

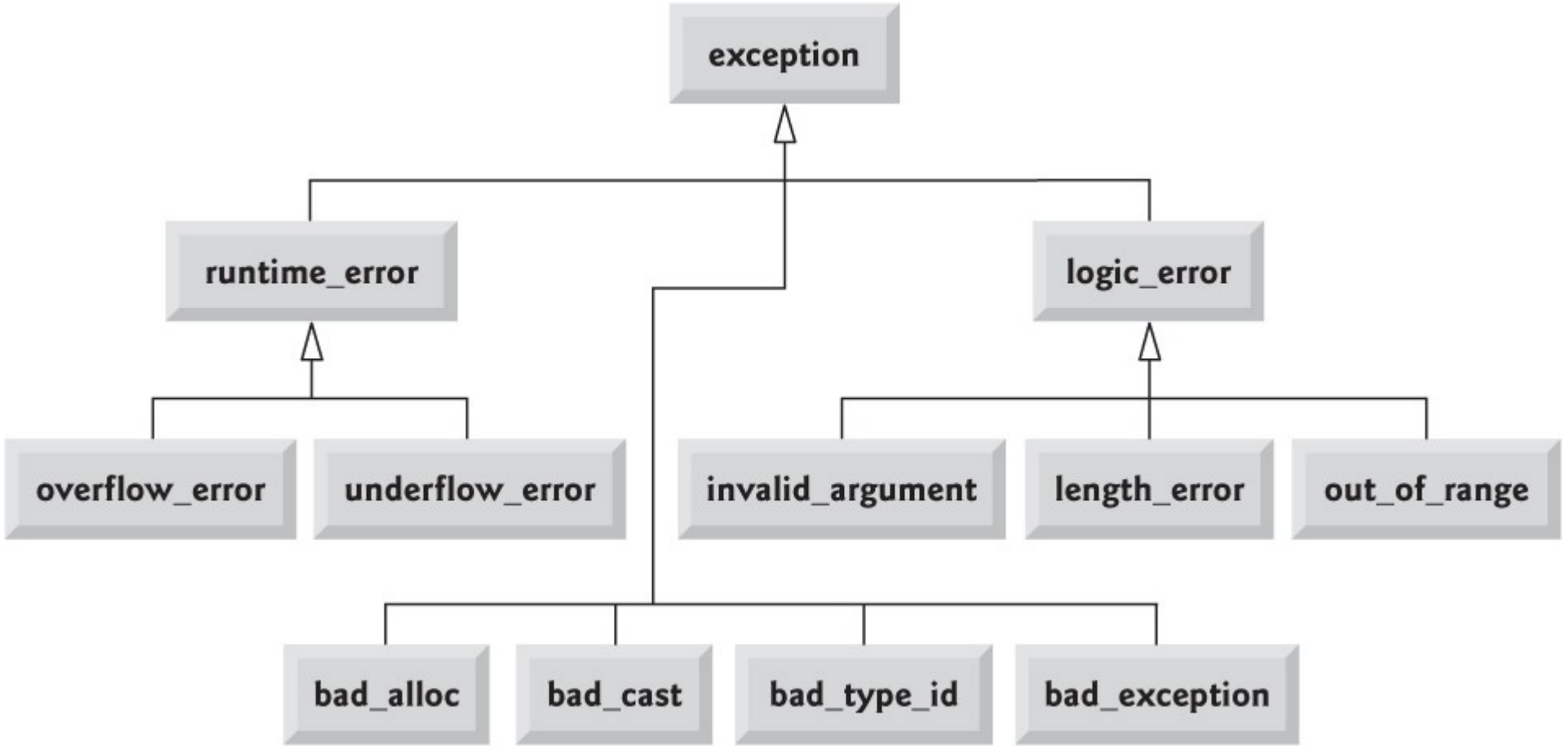
There are times when all of these things converge:

- Higher-level domain modeling
- Frameworks and framework extensions
- Differential programming

If you're not doing any of these things, you probably won't need class inheritance very often. The "preference" for composition is not a matter of "better", it's a question of "most appropriate" for your needs, in a specific context.

Héritage simple, présentation

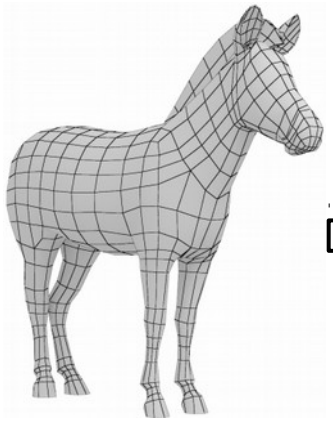
- ***Un exemple de hiérarchie : exceptions STL***
  Les exceptions sont un mécanisme de gestion des situations anormales (échec ouverture fichier...)



COURS 8

A) Héritage simple, présentation
B) Héritage simple en C++
C) Upcasting, slicing
D) Virtuel & polymorphisme

Héritage simple en C++



Attribut(s)

Classe de base

Méthode(s)

Héritage

Attribut(s) en plus

Classe dérivée

Méthode(s) en plus

Héritage simple en C++

- *Avant d'entrer dans le vif du code, un mot sur les spécificateurs d'accès* ***public*** */* ***protected*** */* ***private***

Symboles UML correspondants

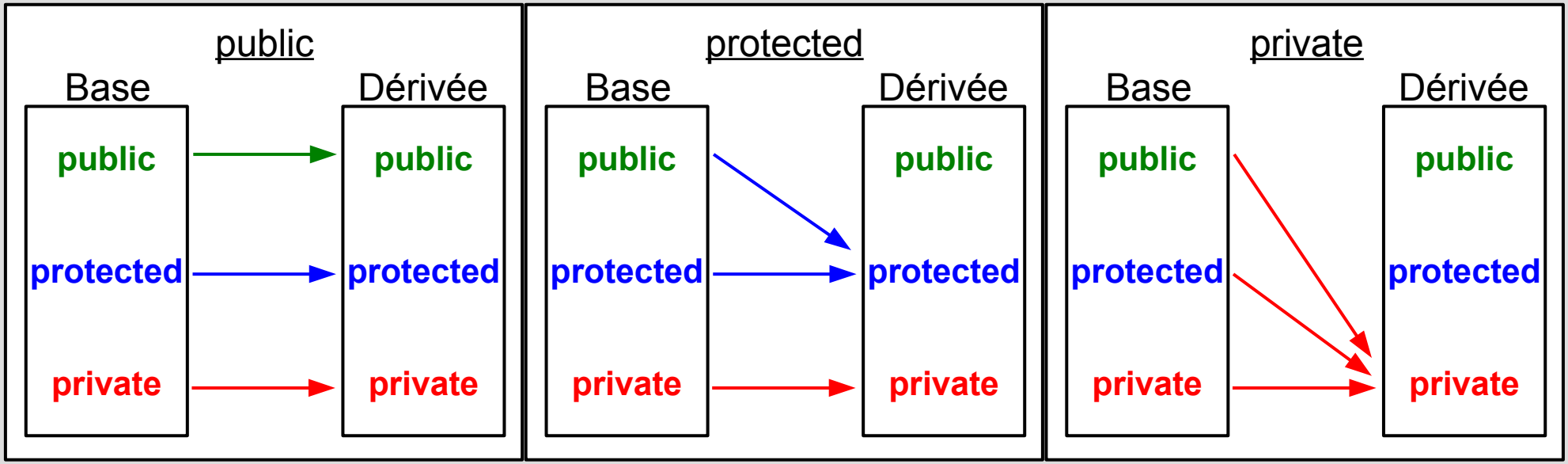
- **+** ***public*** *: membre accessible par n'importe quel code qui « connaît » l'objet (valeur ou référence...)*
- **#** ***protected*** *: membre accessible uniquement par les méthodes de la classe elle même et les méthodes des classes filles*
- **–** ***private*** *: membre accessible uniquement par les méthodes de la classe elle même pas d'accès pour les classes filles !*

Héritage simple en C++

- *Si on considère que les classes filles ont vocation à être intimes (couplées) avec les attributs de la classe mère alors on les met en **protected***
- *Vous comprenez ce que ça implique : si un aspect **implémentation** de la classe mère change, par exemple ses besoins en attributs évoluent, alors le code des classes filles qui accèdent directement à ces attributs « protected » est cassé*
- *C'est un compromis qui dépend de l'application :*
  - la classe mère a-t-elle des attributs + ou – stables ?
  - les classes filles doivent-elles entrer dans les détails ?
- *Si on préfère encapsuler la mécanique de la classe mère sans pour autant en bloquer l'usage pour ses filles ni la truffer d'accesseurs publiques :*
  attributs en private, getters et setters en protected

Héritage simple en C++

- *L'héritage lui même peut être public / protected / private*
- *Un héritage public maintient les droits au même niveau l'héritage protected ou private les restreint*



- *Dans les exemples de code qui suivent j'ai choisi de mettre les attributs en protected (ouvrir leur accès direct au classes filles) et l'héritage en public pour maintenir les droits d'accès au même niveau dans toute la hiérarchie et ne pas alourdir avec des getters...*

Héritage simple en C++

zebra.h

```
class Zebra
{
    public :
        Zebra(double water, int nbstripes = 7);
        double getWater();
        void drink(double moreWater);
        void showStripes();

    protected :
        double m_water;
        int m_nbstripes;
};
```

zebra.cpp

```
Zebra::Zebra(double water, int nbstripes)
    : m_water{water}, m_nbstripes{nbstripes}
{ }

double Zebra::getWater()
{
    return m_water;
}

void Zebra::drink(double moreWater)
{
    m_water = std::min(m_water+moreWater, 20.0);
}

void Zebra::showStripes()
{
    std::cout << std::string(m_nbstripes,'|')
              << std::endl;
}
```

Héritage simple en C++

main.cpp

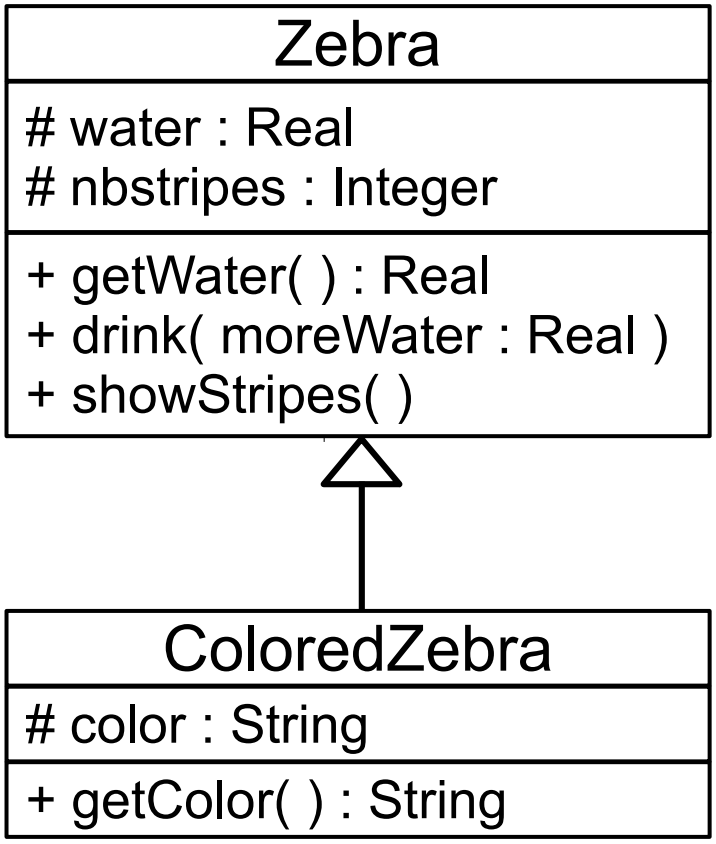
```cpp
int main()
{

    /// Arnold Zebra
    Zebra arnold{5};

    std::cout << "Arnold Zebra" << std::endl;
    arnold.showStripes();
    arnold.drink(5);
    std::cout << arnold.getWater() << std::endl;
    std::cout << std::endl;
```

```
Arnold Zebra
|||||||
10
```

zebra.cpp

```cpp
double Zebra::getWater()
{
    return m_water;
}

void Zebra::drink(double moreWater)
{
    m_water = std::min(m_water+moreWater, 20.0);
}

void Zebra::showStripes()
{
    std::cout << std::string(m_nbstripes,'|')
              << std::endl;
}
```

Héritage simple en C++



Héritage simple en C++

coloredzebra.h

```
class ColoredZebra : public Zebra
{
    public :
        ColoredZebra(double water, std::string color);
        std::string getColor();

    protected :
        std::string m_color;
};
```

coloredzebra.cpp

```
ColoredZebra::ColoredZebra(double water, std::string color)
    : Zebra{water}, m_color{color}
{ }

std::string ColoredZebra::getColor()
{
    return m_color;
}
```

main.cpp

```
    /// Barbara ColoredZebra
    ColoredZebra Barbara{19, "blue"};

    std::cout << "Barbara ColoredZebra" << std::endl;
    barbara.showStripes();
    std::cout << "^" << barbara.getColor() << "^\n";
    barbara.drink(5);
    std::cout << barbara.getWater() << std::endl;
    std::cout << std::endl;
```

```
Barbara ColoredZebra
|||||||
^blue^
20
```

Héritage simple en C++

```
class ColoredZebra : public Zebra
{
    public :
        ColoredZebra(double water, std::string color);
        std::string getColor();

    protected :
        std::string m_color;
};
```

coloredzebra.h

- *La syntaxe de l'héritage publique de la classe Derivee depuis la classe Base est*

```
class Derivee : public Base
{
    … attributs et méthodes ajoutées …
};
```

Héritage simple en C++

zebra.h

```
class Zebra
{
    public :
        Zebra(double water, int nbstripes = 7);
    ...
```

zebra.cpp

```
Zebra::Zebra(double water, int nbstripes)
    : m_water{water}, m_nbstripes{nbstripes}
{ }
```

coloredzebra.h

```
class ColoredZebra : public Zebra
{
    public :
        ColoredZebra(double water, std::string color);
        std::string getColor();

    protected :
        std::string m_color;
};
```

coloredzebra.cpp

```
ColoredZebra::ColoredZebra(double water, std::string color)
    : Zebra{water}, m_color{color}
{ }
```

main.cpp

```
    ColoredZebra barbara{19, "blue"};
```

1) Demande construction dérivée

Héritage simple en C++

zebra.h

```
class Zebra
{
    public :
        Zebra(double water, int nbstripes = 7);
    ...
```

Deuxième paramètre defaulted !

zebra.cpp

```
Zebra::Zebra(double water, int nbstripes)
    : m_water{water}, m_nbstripes{nbstripes}
{ }
```

coloredzebra.h

```
class ColoredZebra : public Zebra
{
    public :
        ColoredZebra(double water, std::string color);
        std::string getColor();

    protected :
        std::string m_color;
};
```

coloredzebra.cpp

```
ColoredZebra::ColoredZebra(double water, std::string color)
    : Zebra{water}, m_color{color}
{ }
```

main.cpp

```
    ColoredZebra barbara{19, "blue"};
```

2) Relais liste init. constr. dérivée

Héritage simple en C++

zebra.h

```
class Zebra
{
    public :
        Zebra(double water, int nbstripes = 7);
    ...  Attributs de base initialisés
```

zebra.cpp

```
Zebra::Zebra(double water, int nbstripes)
    : m_water{water}, m_nbstripes{nbstripes}
{ }
```

coloredzebra.h

```
class ColoredZebra : public Zebra
{
    public :
        ColoredZebra(double water, std::string color);
        std::string getColor();

    protected :
        std::string m_color;
};
```

coloredzebra.cpp

```
ColoredZebra::ColoredZebra(double water, std::string color)
    : Zebra{water}, m_color{color}
{ }
```

main.cpp

```
    ColoredZebra barbara{19, "blue"};
```

3) liste init. constr. base, l'objet de base est construit !

Héritage simple en C++

zebra.h

```
class Zebra
{
    public :
        Zebra(double water, int nbstripes = 7);
    ...
```

zebra.cpp

```
Zebra::Zebra(double water, int nbstripes)
    : m_water{water}, m_nbstripes{nbstripes}
{ }
```

coloredzebra.h

```
class ColoredZebra : public Zebra
{
    public :
        ColoredZebra(double water, std::string color);
        std::string getColor();

    protected :
        std::string m_color;
};
```

Attributs de dérivée initialisés

coloredzebra.cpp

```
ColoredZebra::ColoredZebra(double water, std::string color)
    : Zebra{water}, m_color{color}
{ }
```

main.cpp

```
    ColoredZebra barbara{19, "blue"};
```

4) liste init. constr. dérivée, suite et fin dérivée est construit !

Héritage simple en C++

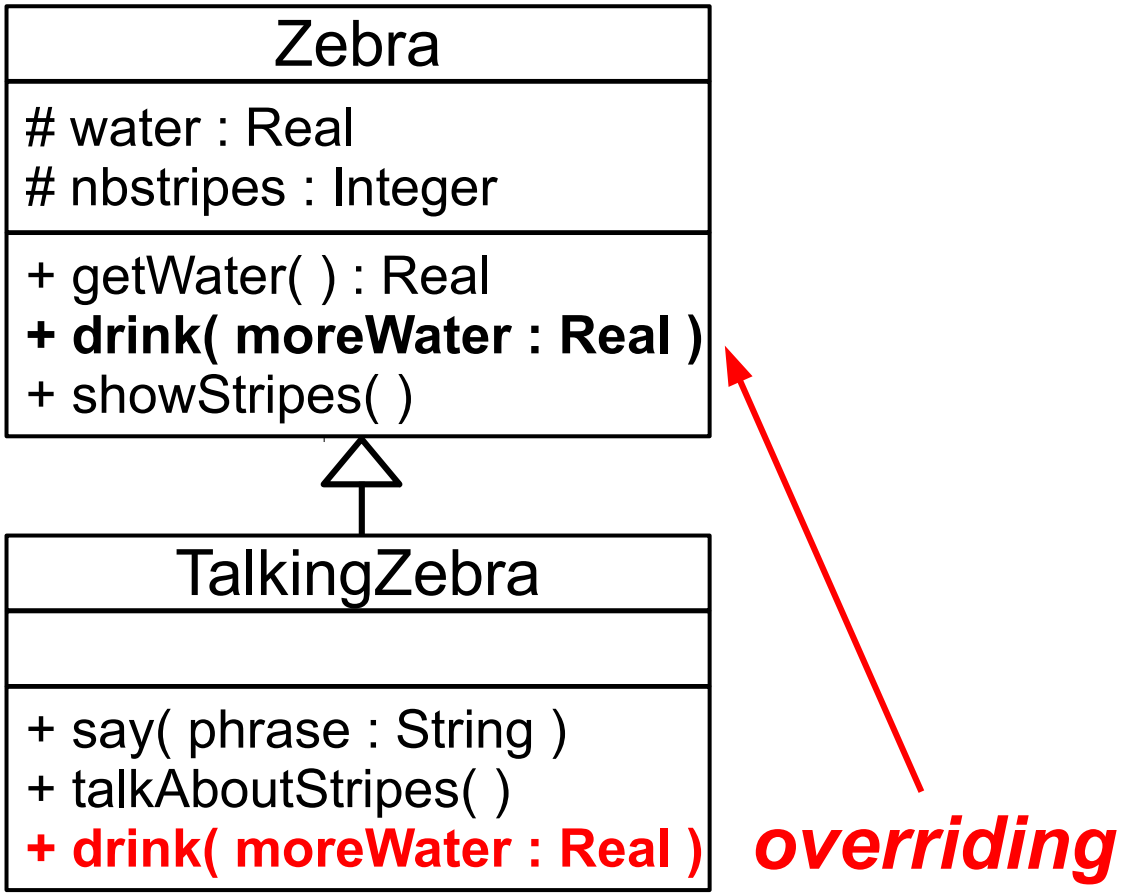
- *La demande de construction de l'objet dérivé par le code client **passe par le constructeur de dérivée** qui fait **relais vers le constructeur de base***
- *La partie **base est construite avant** la partie dérivée*
- *On revient **après au constructeur dérivé** pour finir !*
- *Pour une hiérarchie à plusieurs niveaux les relais s'enchaînent du plus dérivé au plus haut... La construction/finalisation part enfin du plus haut et redescend vers les plus spécialisés*
- *Sans oublier qu'il faut qu'il y ait **relais explicite 2)** on peut dire que concrètement la construction se fait de la classe base vers la classe dérivée*
- *La destruction (plus simple) se fait en sens inverse !*

Héritage simple en C++

Construction
classe dérivée

Destruction
classe dérivée

Héritage simple en C++



Héritage simple en C++

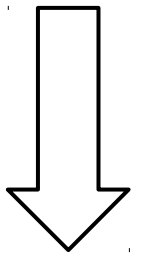
talkingzebra.h

```
class TalkingZebra : public Zebra
{
    public :
        TalkingZebra();
        TalkingZebra(int nbstripes);
        void say(std::string phrase);
        void talkAboutStripes();
        /// Overriding
        void drink(double moreWater);
};
```

talkingzebra.cpp

```
/// Les zèbres bavards ont toujours soif au départ...
TalkingZebra::TalkingZebra()
    : Zebra{0}
{ }

TalkingZebra::TalkingZebra(int nbstripes)
    : Zebra{0, nbstripes}
{ }

void TalkingZebra::say(std::string phrase)
{
    size_t ps = phrase.size();
    std::cout << "╔" << std::string(ps,'=') << "╗\n";
    std::cout << "╠" <<          phrase      << "╣\n";
    std::cout << "╚" << std::string(ps,'=') << "╝\n";
}
```

Suite slide suivant



Héritage simple en C++

talkingzebra.h

```
class TalkingZebra : public Zebra
{
    public :
        TalkingZebra();
        TalkingZebra(int nbstripes);
        void say(std::string phrase);
        void talkAboutStripes();
        /// Overriding
        void drink(double moreWater);
};
```

talkingzebra.cpp

```
void TalkingZebra::talkAboutStripes()
{
    say("Hey, see my "
        + std::to_string(m_nbstripes)
        + " stripes ?");

    showStripes();   Appel à la méthode showStripes de la classe de base
}

void TalkingZebra::drink(double moreWater)
{
    if ( m_water<1.0 )
        say("J'ai trop soif !"); Appel à la méthode say de la classe dérivée
    else
        say("Slurp...");
    Zebra::drink(moreWater);    Appel à la méthode drink de la classe de base
}
```

Héritage simple en C++

main.cpp

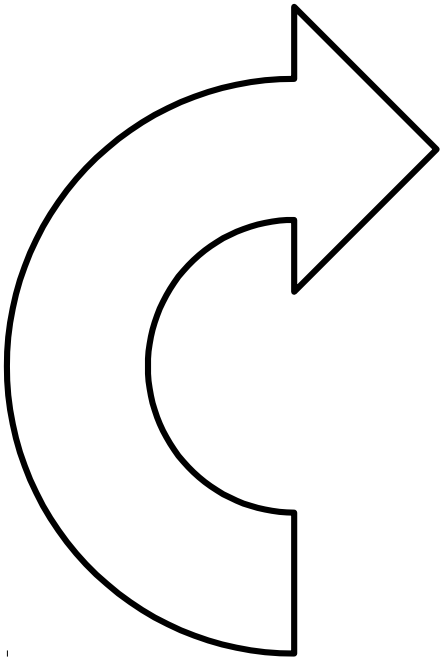
```
/// Cathy TalkingZebra
TalkingZebra cathy;

cathy.say("Hello, I'm Cathy"
          " the TalkingZebra");
cathy.drink(5);
cathy.drink(5);
cathy.talkAboutStripes();
std::cout << std::endl;

/// David TalkingZebra with less stripes
TalkingZebra david{5};

david.say("I'm David the TalkingZebra");
david.talkAboutStripes();
david.say("I'm envious of Cathy !");
std::cout << std::endl;
```

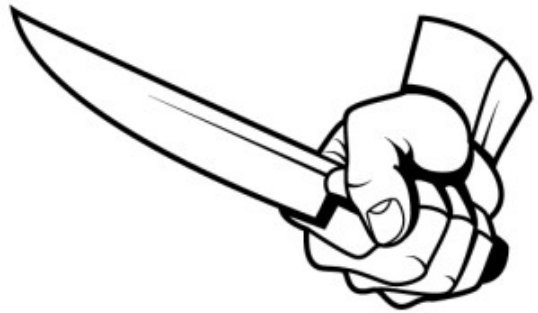
Appels à la méthode redéfinie (overriden)

COURS 8

A) Héritage simple, présentation
B) Héritage simple en C++
C) Upcasting, slicing
D) Virtuel & polymorphisme

Upcasting, slicing



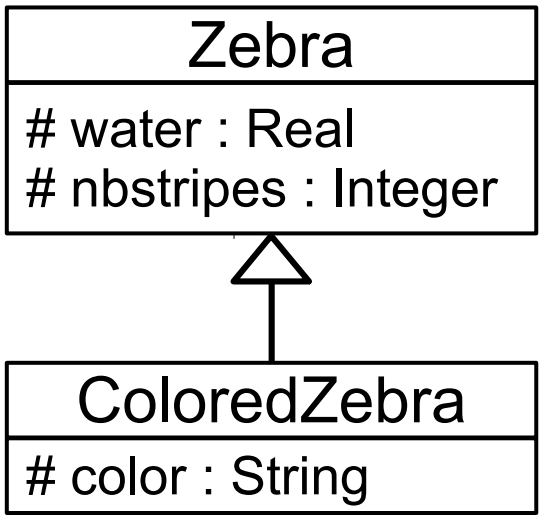


Upcasting, slicing

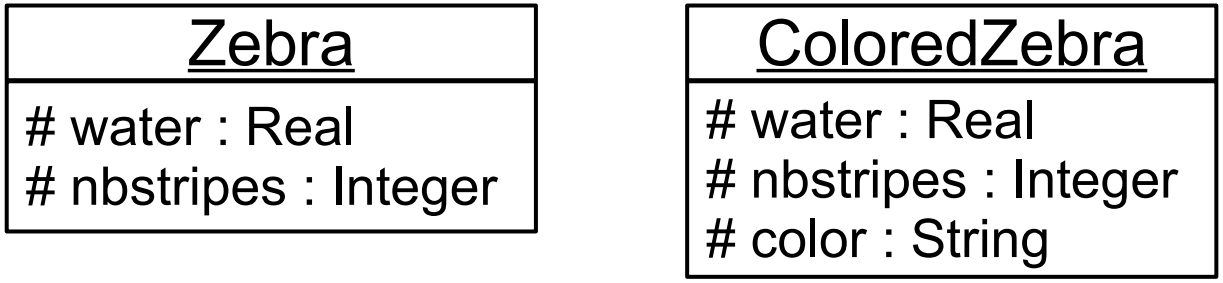
- *Il est possible de convertir un type dérivé dans un type de base, cette conversion de type est un **cast***
- *Cast « vers le haut » (vers une classe de base) => **upcasting***
- *L'upcasting se fait automatiquement, sans erreur, sans warning...*
- *La version castée vers le type de base est une version « raccourcie » de la classe dérivée : on y a perdu les attributs spécifiques de la dérivée !*
- *Les données spécifiques sont toujours là, rien n'est perdu tant qu'on garde l'objet dans son type dérivé quelque part : c'est **la copie ou la référence** au type de base qui n'a pas tout. C'est le **slicing**.*

Upcasting, slicing

Le modèle théorique
de l'héritage : on ne répète pas
les données membres dans la classe dérivée

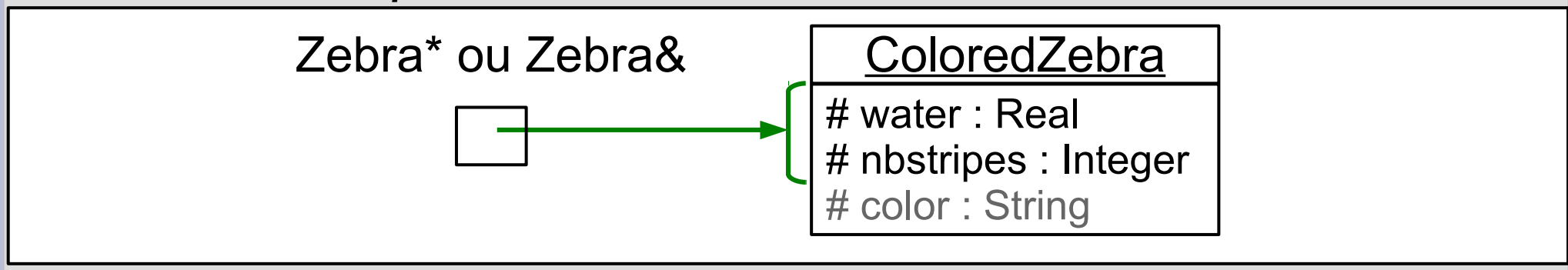


Le modèle réel de l'héritage : en mémoire un objet
de la classe dérivée a bien les données de la classe parente !

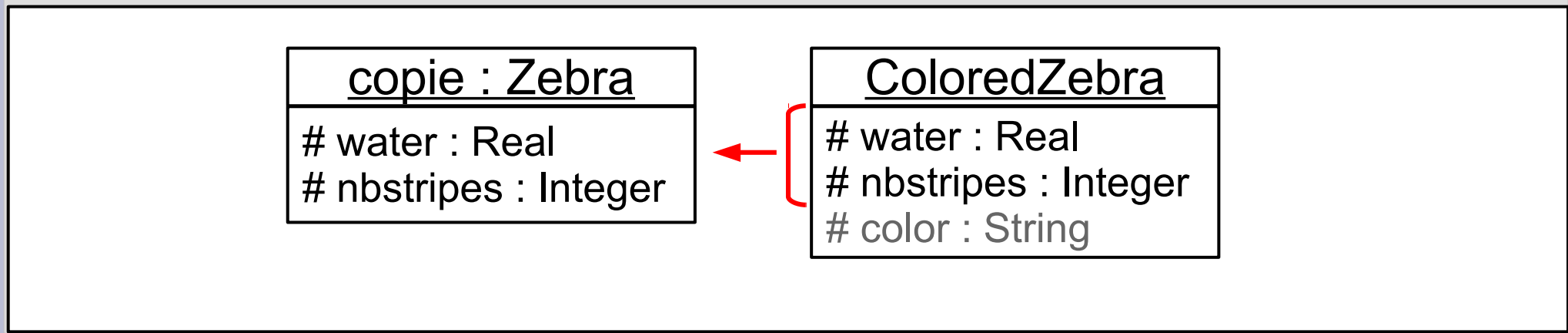


Upcasting, slicing

- *Lors d'un **upcast** par référence ou par adresse le système ne considère que la « partie haute » et n'autorise pas l'accès aux attributs dérivés*



- *Lors d'un **upcast** par valeur le système copie la partie commune à la classe de base dans un nouvel objet de la classe de base*



Upcasting, slicing

```
Zebra arnold{0};
ColoredZebra barbara{0, "blue"};
TalkingZebra cathy;

Zebra a = arnold;
Zebra b = barbara;
Zebra c = cathy;

std::cout << sizeof(a) << " "
          << sizeof(arnold) << std::endl;

std::cout << sizeof(b) << " "
          << sizeof(barbara) << std::endl;

std::cout << sizeof(c) << " "
          << sizeof(cathy) << std::endl;
```

Sliced !

```
16 16
16 40
16 16
```

Ce qui n'implique aucune perte d'information,
l'objet barbara est toujours là avec ses 40 octets
c'est la copie b qui est slicée …
Ce qu'il ne faut pas c'est jeter barbara et garder b !

Upcasting, slicing

```
void giveWaterByReference(Zebra& z)
{
    std::cout << sizeof(z) << std::endl;
    z.drink(5);
}

int main()
{
    Zebra arnold{0};
    ColoredZebra barbara{0, "blue"};
    TalkingZebra cathy;

    std::cout << arnold.getWater() << std::endl;
    std::cout << barbara.getWater() << std::endl;
    std::cout << cathy.getWater() << std::endl;

    giveWaterByReference(arnold);
    giveWaterByReference(barbara);
    giveWaterByReference(cathy);

    std::cout << arnold.getWater() << std::endl;
    std::cout << barbara.getWater() << std::endl;
    std::cout << cathy.getWater() << std::endl;
```

On donne à boire à barbara par l'intermédiaire d'une référence slicée !

Et tout va bien !

```
0
0
0
16
16
16
5
5
5
```

Upcasting, slicing

```
void giveWaterByReference(Zebra& z)
{
    std::cout << sizeof(z) << std::endl;
    z.drink(5);
}

int main()
{
    Zebra arnold{0};
    ColoredZebra barbara{0, "blue"};
    TalkingZebra cathy;

    std::cout << arnold.getWater() << std::endl;
    std::cout << barbara.getWater() << std::endl;
    std::cout << cathy.getWater() << std::endl;

    giveWaterByReference(arnold);
    giveWaterByReference(barbara);
    giveWaterByReference(cathy);

    std::cout << arnold.getWater() << std::endl;
    std::cout << barbara.getWater() << std::endl;
    std::cout << cathy.getWater() << std::endl;
```

Par contre pour Cathy elle a perdu sa qualité talkingZebra : la méthode redéfinie drink est slicée !

« J'ai trop soif ! » ?

```
0
0
0
16
16
16
5
5
5
```

Upcasting, slicing

- *Attention à la situation suivante, qui compile sans même un Warning et qui conduit à des anomalies (fuites mémoires et/ou plantages)*
- *On aura une solution radicale à ces 2 problèmes...*

```
void discardZebra(Zebra* pz)
{
    std::cout << "Good bye zebra of size "
              << sizeof(*pz) << std::endl;
    delete pz;
}

int main()
{

    ColoredZebra *dynamic = new ColoredZebra{0, "red"};

    std::cout << "Hello to zebra of size "
              << sizeof(*dynamic) << std::endl;

    /// Use dynamic...

    discardZebra(dynamic);
```

NOT GOOD !

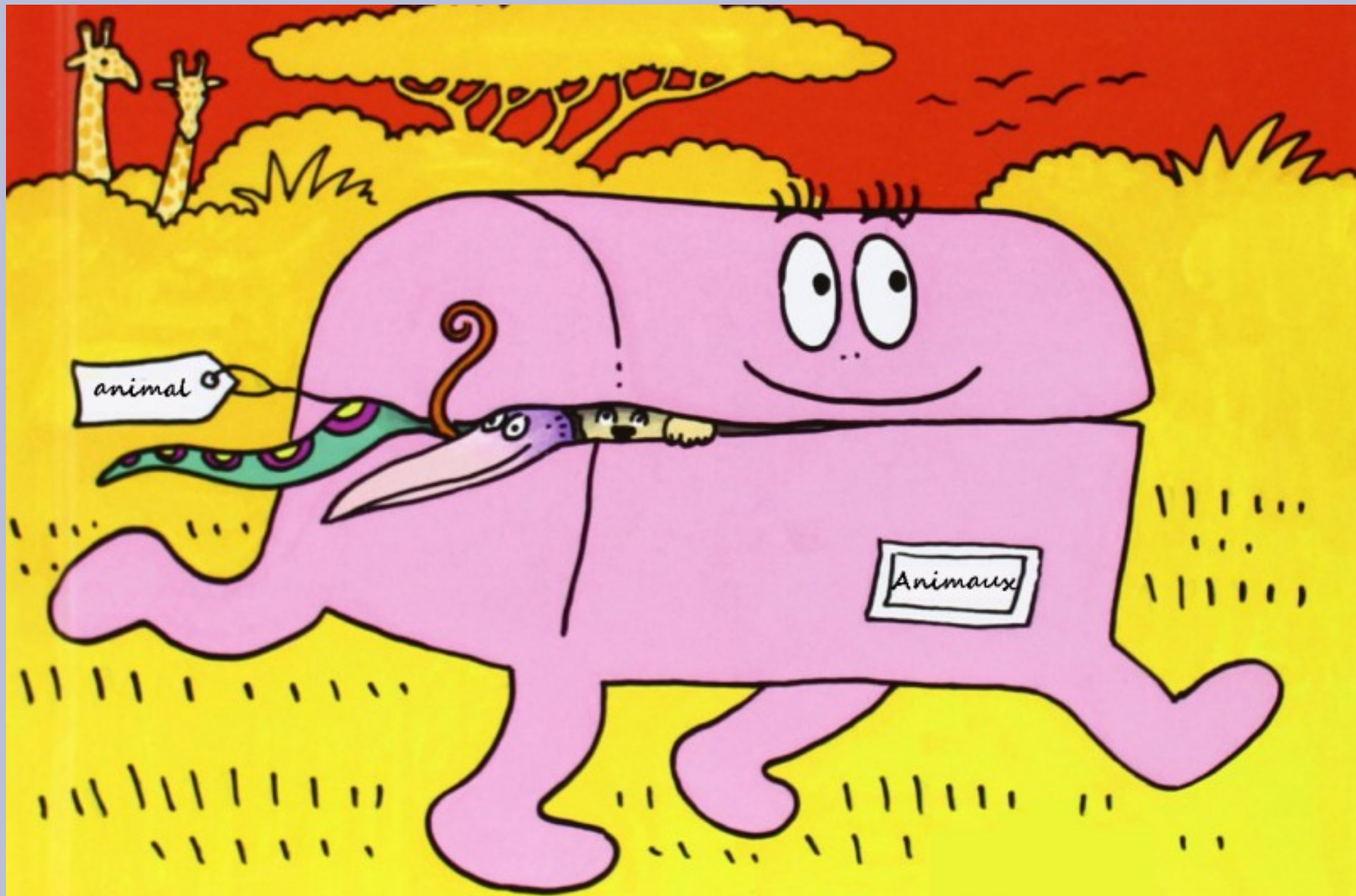
```
Hello to zebra of size 40
Good bye zebra of size 16
```

Note :
En toute rigueur sizeof est une indication "statique" qui ne pourrait pas retourner une info sur un aspect de type polymorphe au run-time. Donc ça n'est pas décisif...

COURS 8

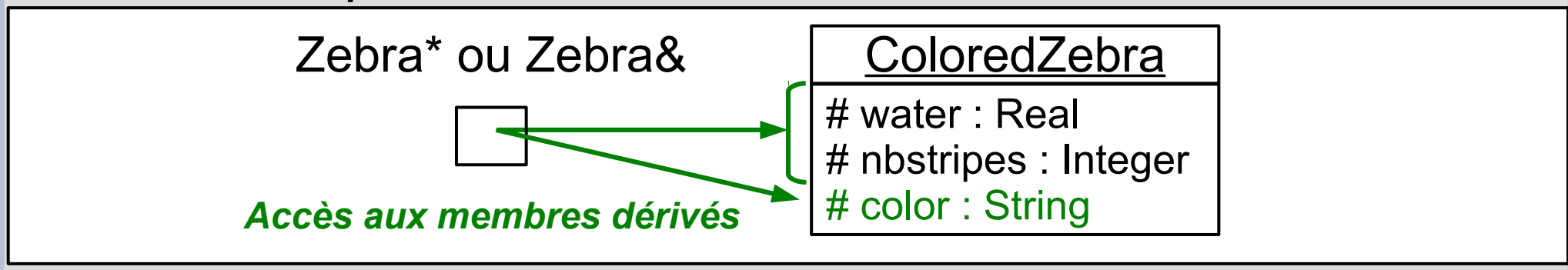
A) Héritage simple, présentation
B) Héritage simple en C++
C) Upcasting, slicing
D) Virtuel & polymorphisme

Virtuel & polymorphisme

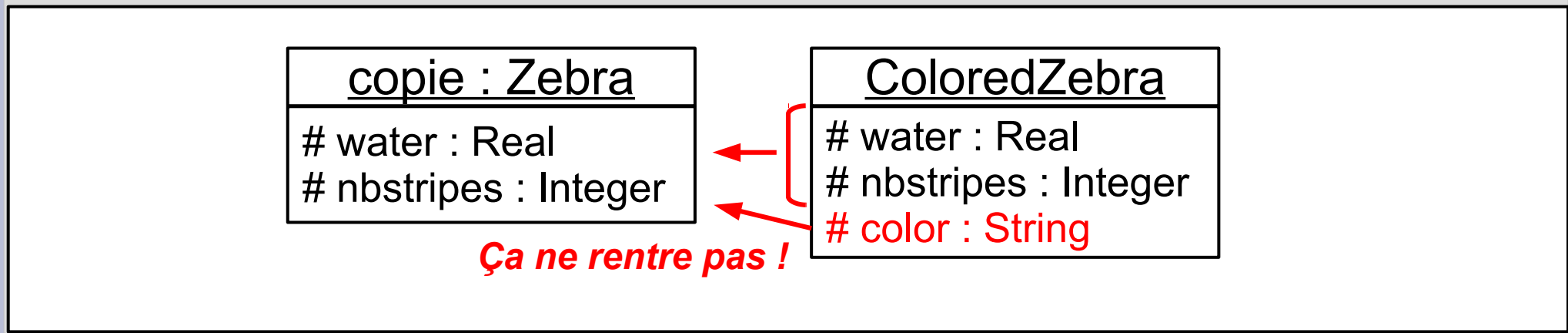


Virtuel & polymorphisme

- *On peut demander au système de compilation de conserver un accès aux méthodes et attributs dérivés quand on a un accès adresse ou référence*

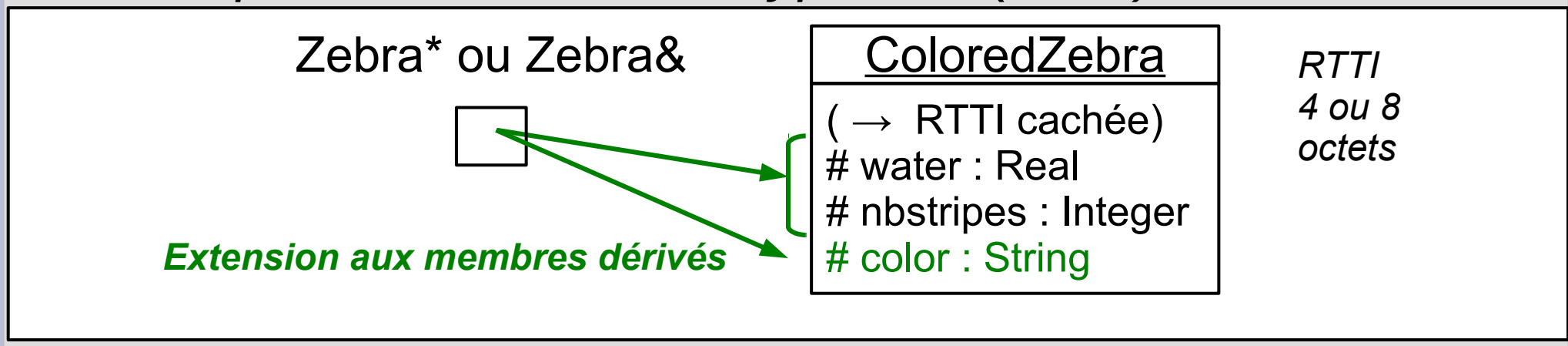


- *Ce mécanisme ne peut pas fonctionner par valeur : on ne peut pas « caser » les attributs supplémentaires dans l'espace de stockage du type de base...*



Virtuel & polymorphisme

- *Mais ça implique que chaque objet soit porteur d'une information de type qui n'est pas connue à la compilation : Run-Time Type Info (RTTI)*



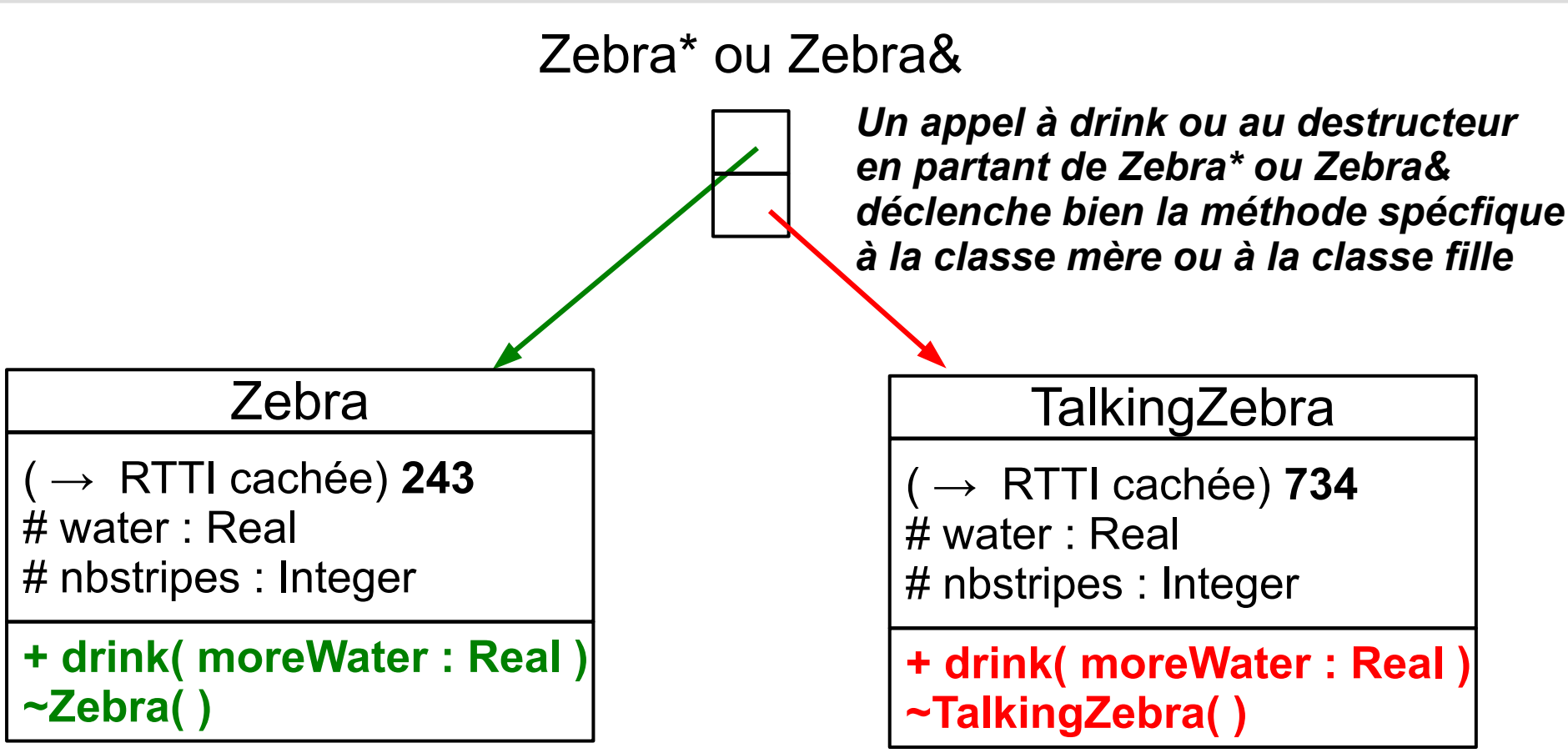
- *Cette information a un poids. La philosophie du C++ est qu'on ne paye que pour ce qu'on utilise...*
  Ce qui est pratique mais qui coûte doit être demandé
- *La façon de demander un comportement polymorphe est de déclarer une/des méthodes **virtuelles***
  en faisant précéder leur déclaration de **`virtual`**

Virtuel & polymorphisme

- *En fait les attributs sont généralement cachés (private ou protected) et n'existent pas vu depuis le type de base...*
- *Ce qui nous intéresse ce sont les* ***méthodes redéfinies dans les classes dérivées*** *(lesquelles peuvent effectivement utiliser des attributs que la classe de base n'a pas)*
- *Ce sont ces méthodes qui seront déclarées* `virtual` *dans la classe de base pour qu'un pointeur ou une référence de type base puisse appeler la version redéfinie de la classe dérivée effectivement référencée*
- *C'est aussi valable et important pour le destructeur, (y compris le destructeur automatique) pour pouvoir détruire des objets dynamiques dérivés, par un Base**

Virtuel & polymorphisme

- *En fait les attributs sont généralement cachés (private ou protected) et n'existent pas vu depuis le type de base...*



Virtuel & polymorphisme

- *On dit alors qu'on a un **comportement polymorphe** et que la classe est une classe polymorphe (on peut redéfinir ses méthodes sans se faire slicer!)*
- *A retenir : **4 "conditions" pour le polymorphisme***

➔ ***Ça concerne l'héritage*** *(pas d'héritage, pas de polymorphisme)*

➔ ***Une classe fille redéfinit des méthodes de la mère***

➔ ***Les objets dérivés (ou pas) sont manipulés par pointeurs ou références de type classe mère***

➔ ***Les méthodes polymorphes sont déclarées*** `virtual` ***dans la classe mère***

- ***Quelle usine à gaz ! Ça sert à quoi ? C'est génial ?***

Virtuel & polymorphisme

zebra.h

```
class Zebra
{
    public :
        Zebra(double water, int nbstripes = 7);
        virtual ~Zebra() = default;
        double getWater();
        virtual void drink(double moreWater);
        void showStripes();

    protected :
        double m_water;
        int m_nbstripes;
};
```

Obligatoire pour le polymorphisme des méthodes destructeur et drink virtual dans la classe mère

talkingzebra.h

```
class TalkingZebra : public Zebra
{
    public :
        TalkingZebra();
        TalkingZebra(int nbstripes);
        virtual ~TalkingZebra() = default;
        virtual void say(std::string phrase);
        void talkAboutStripes();
        /// Overriding
        virtual void drink(double moreWater);
};
```

Facultatif on peut remettre virtual dans la version spécialisée de la classe fille

Obligatoire pour le polymorphisme de la méthode say si on veut la redéfinir pour CoolTalkingZebra

Virtuel & polymorphisme

```
void giveWaterByReference(Zebra& z)
{
    std::cout << sizeof(z) << std::endl;
    z.drink(5);
}

int main()
{
    Zebra arnold{0};
    ColoredZebra barbara{0, "blue"};
    TalkingZebra cathy;

    std::cout << arnold.getWater() << std::endl;
    std::cout << barbara.getWater() << std::endl;
    std::cout << cathy.getWater() << std::endl;

    giveWaterByReference(arnold);
    giveWaterByReference(barbara);
    giveWaterByReference(cathy);

    std::cout << arnold.getWater() << std::endl;
    std::cout << barbara.getWater() << std::endl;
    std::cout << cathy.getWater() << std::endl;
```

Cathy garde sa qualité talkingZebra à travers une référence Zebra: la méthode redéfinie drink est polymorphe !

```
0
0
0
24
24
24
J'ai trop soif !
5
5
5
```

8 octets en plus pour tous les objets (RTTI)

Virtuel & polymorphisme

- Pas impressionnés ?
- Attendez de voir un **conteneur polymorphe**
- **On peut traiter de manière uniforme un ensemble de types hétérogènes (avec un parent commun)**
- Utilisation typique : dans une interface graphique vous avez des boutons, des cadres, des menus, des zones de textes, des zones d'images etc... Tous ces éléments partagent le fait d'être dessinables et cliquables. Une classe parente à tous déclare virtual sur une méthode dessiner et sur une méthode cliquer et hop : on peut mettre tout ces éléments dans un même conteneur. On dira qu'ils implémentent une même interface. Prochain cours !

Virtuel & polymorphisme

```
int main()
{
    std::vector<Zebra*> zoo;

    /// Tout le monde dans le même zoo !
    zoo.push_back(new Zebra{5});
    zoo.push_back(new ColoredZebra{10, "orange"});
    zoo.push_back(new TalkingZebra);

    /// pz est un pointeur sur chacun successivement
    for (const auto& pz: zoo)
    {
        /// Pas de test à faire, tout le monde sait boire
        pz->drink(5);
        std::cout << pz->getWater() << std::endl;

        /// Est-ce que j'ai un zèbre de couleur ?
        ColoredZebra* cz = dynamic_cast<ColoredZebra*>(pz);
        if (cz)
            std::cout << cz->getColor() << std::endl;

        std::cout << std::endl;
    }

    /// On efface tout le monde sans fuite !
    for (const auto& pz: zoo)
        delete pz;
```

```
10

15
orange

J'ai trop soif !

5
```

Bonnes vacances !