Algorithmique Programmation Objet

Python

Andrea G. B. Tettamanzi
Université de Nice Sophia Antipolis
Département Informatique
andrea.tettamanzi@unice.fr

CM - Séance 4

Introduction à la programmation orientée objet

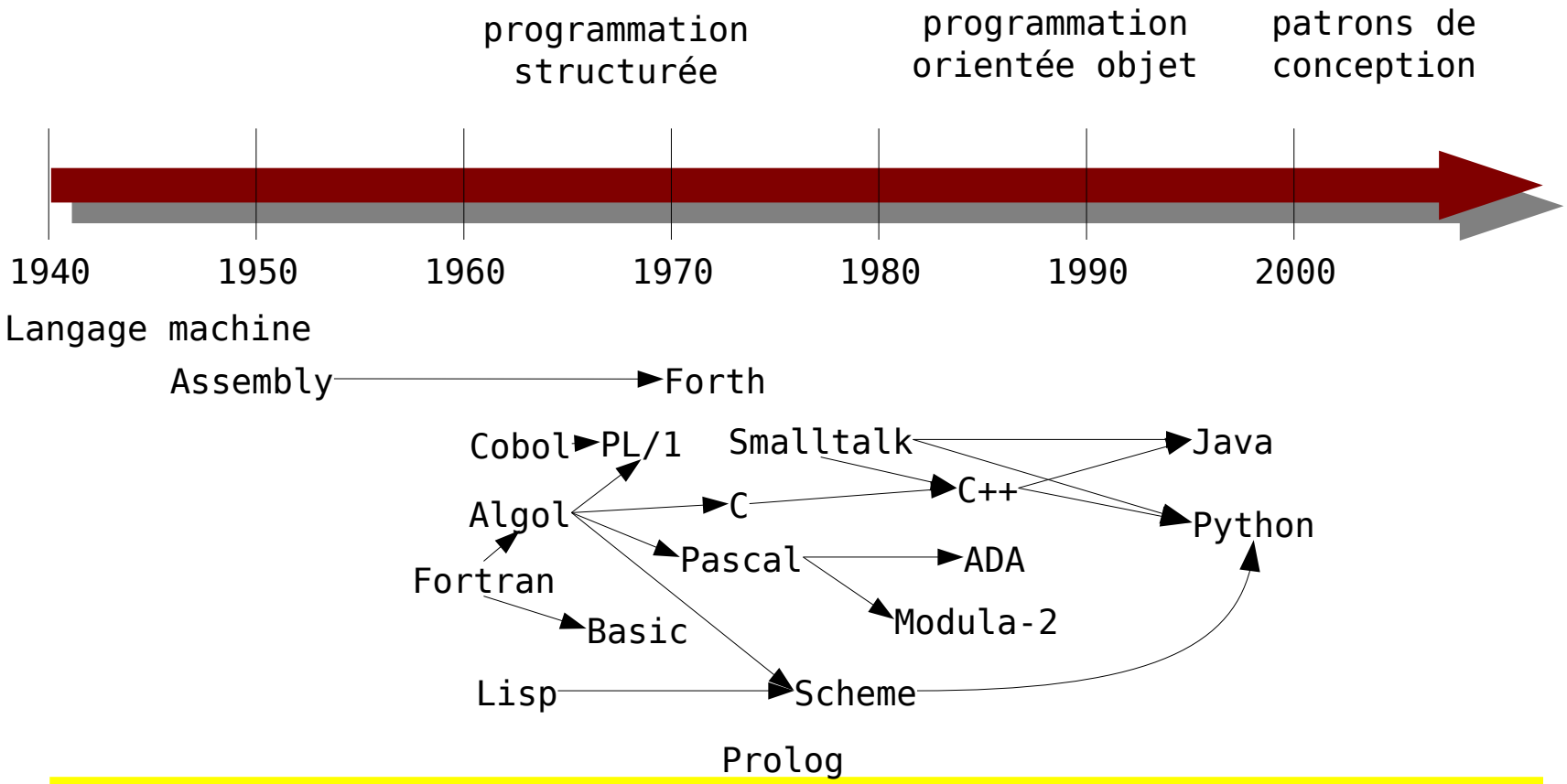
Plan

- Introduction
- Encapsulation
- Types de données abstraits
- Héritage et polymorphisme

Origines

- La programmation orientée objet surgit dans les années '80
- Logiciels de plus en plus complexes, travail en équipe
- Contrôler la complexité des logiciels
- Code réutilisable pour contenir les coûts du développement
- Contenir les coûts de la maintenance :
  - Ajout de nouvelles fonctionnalités
  - Modification de fonctionnalités existantes
  - *portage* sur d'autres plate-formes ou environnements
- Coordonner et répartir le travail de développement
- Modularité

Evolution de la programmation



Programmation orientée objet

Discipline de programmation dans laquelle le programmeur établit

- non seulement les structures de données,
- mais aussi les opérations qui peuvent leurs être appliquées.

Ainsi,

- la structure de données devient un **objet** qui inclut
  - Données, appelées **attributs**
  - Opérations, appelées **méthodes**
- Le programmeur peut définir des **relations** entre les objets

Encapsulation

- Seules les opérations définies à l'intérieur d'une structure de données peuvent manipuler les données de cette structure :
  - L'utilisation d'opérations certifiées sur les structures de données garantit leur consistence
  - Élimine une parmi les causes d'erreur les plus importantes

- Cacher les détails de réalisation :
  - Avère la modularité du code
  - facilite l'extension et la maintenance
  - facilite l'individuation des erreurs

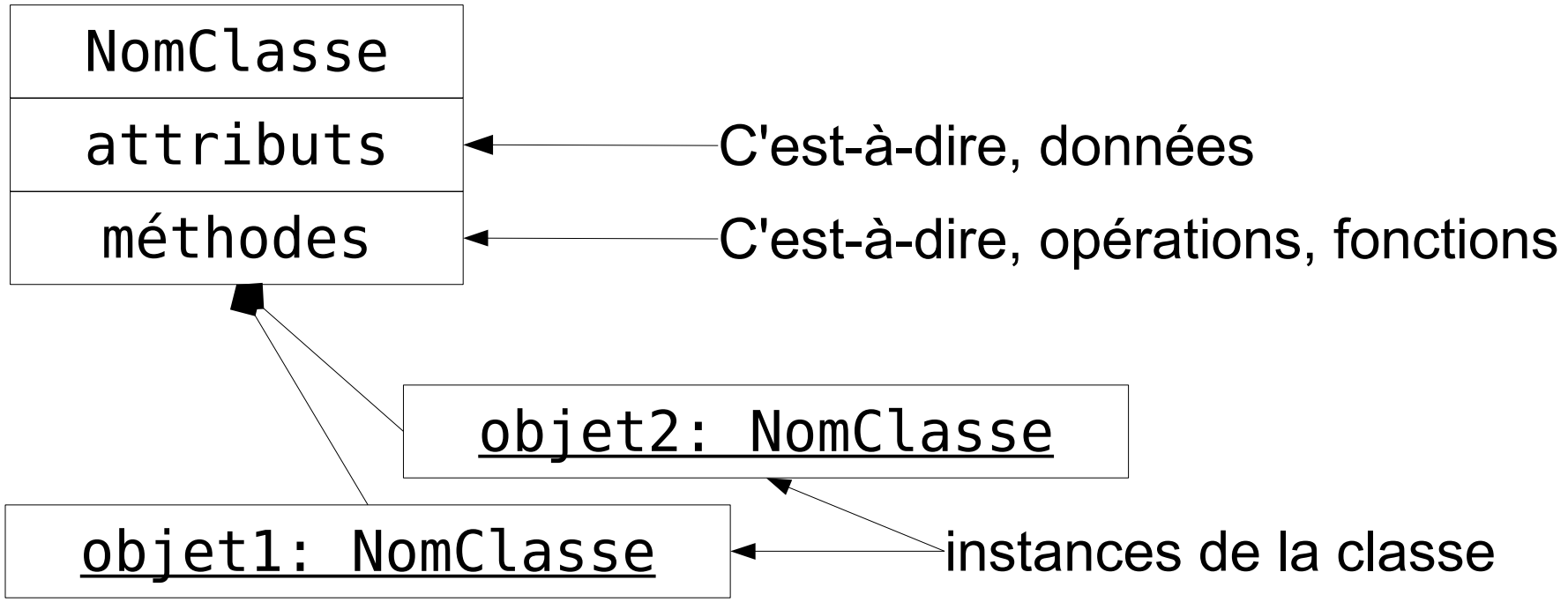
Type abstrait de données

- Possède un ***type***
- Définit un ensemble d'***opérations***, qui constituent son ***interface***
- Les opérations de l'interface sont *la seule manière d'accéder* au type abstrait de données
- Son domaine d'application est défini par des ***axiomes*** et des ***préconditions***

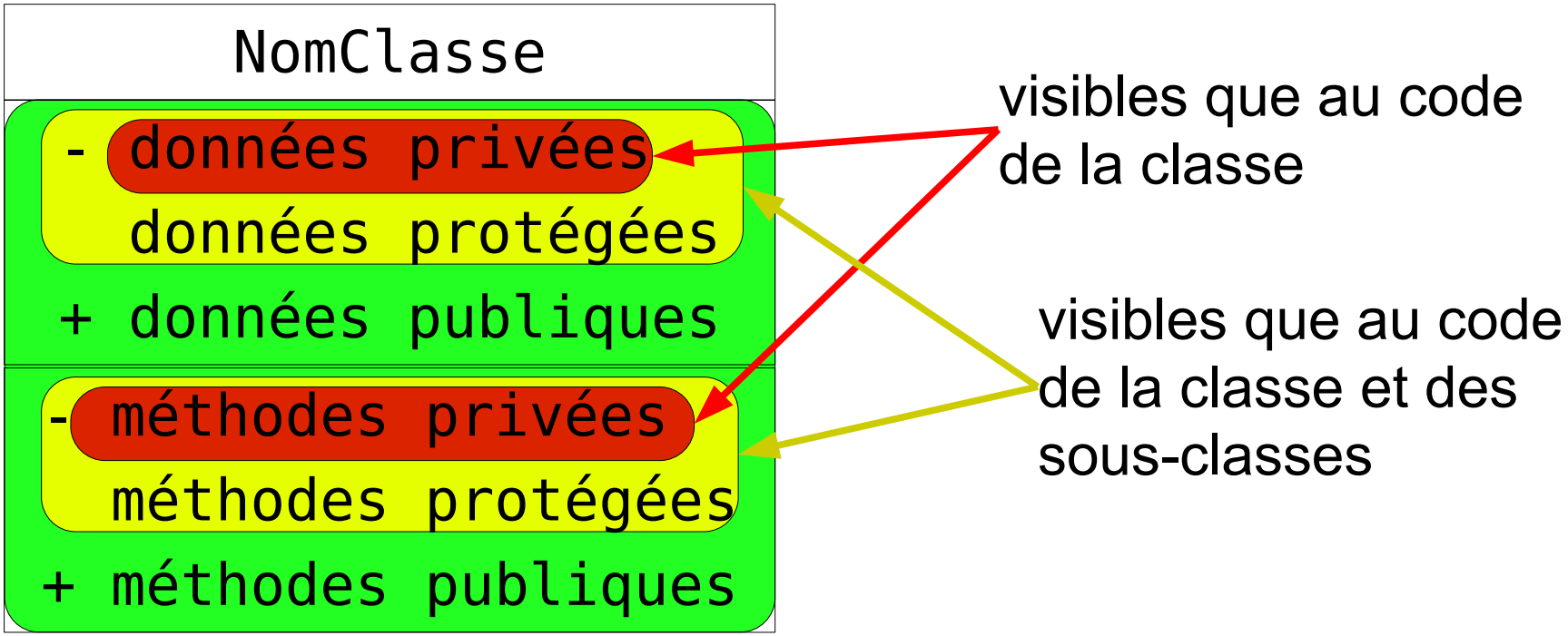
Exemple : Type abstrait de données "Nombre Naturel"

- **Type :** "Nombre Naturel"
- Opérations définies (**Interface**) :
  - S(*n*), *n* + *m*, *n* x *m*, etc.
  - *n* = *m*, *n* < *m*, *n* | *m*, premier(*n*), etc.
- **Axiomes et préconditions :**
  - *n* + 0 = 0 + *n* = *n*
  - *n* + S(*m*) = S(*n*) + *m*
  - *n* x 0 = 0 x *n* = 0
  - *n* x S(*m*) = (*n* x *m*) + *n*
  - *etc.*

Objets et Classes



Visibilité


NomClasse
- données privées
données protégées
+ données publiques
- méthodes privées
méthodes protégées
+ méthodes publiques
visibles que au code de la classe
visibles que au code de la classe et des sous-classes

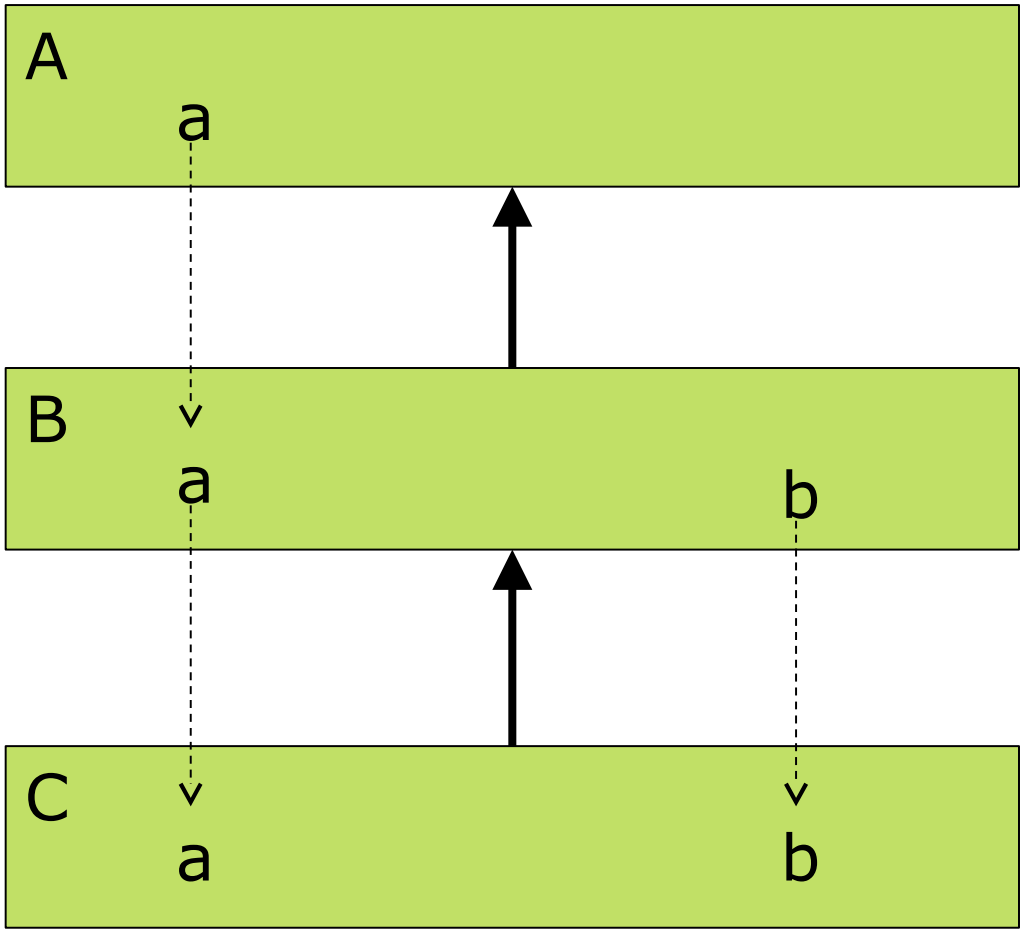
Héritage : Définition

L'**héritage** est la caractéristique d'un langage orienté objet qui fait que les objets d'une classe “héritent” toutes les propriétés définies pour les classes de niveau supérieur :

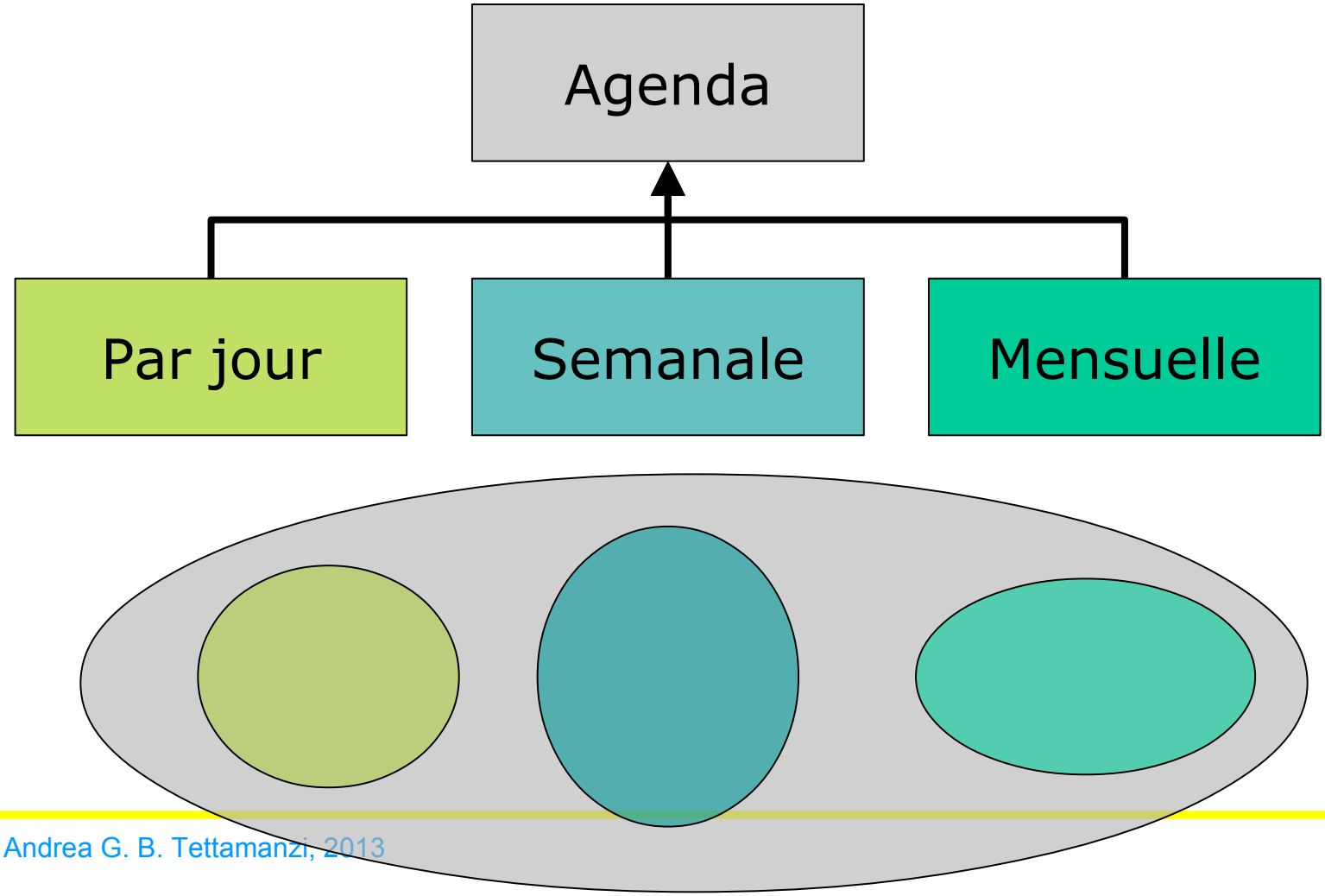
- attributs;
- méthodes;
- etc.

Historiquement, c'est une des caractéristiques les plus controversées.

Héritage



Vision naïve-intuitive



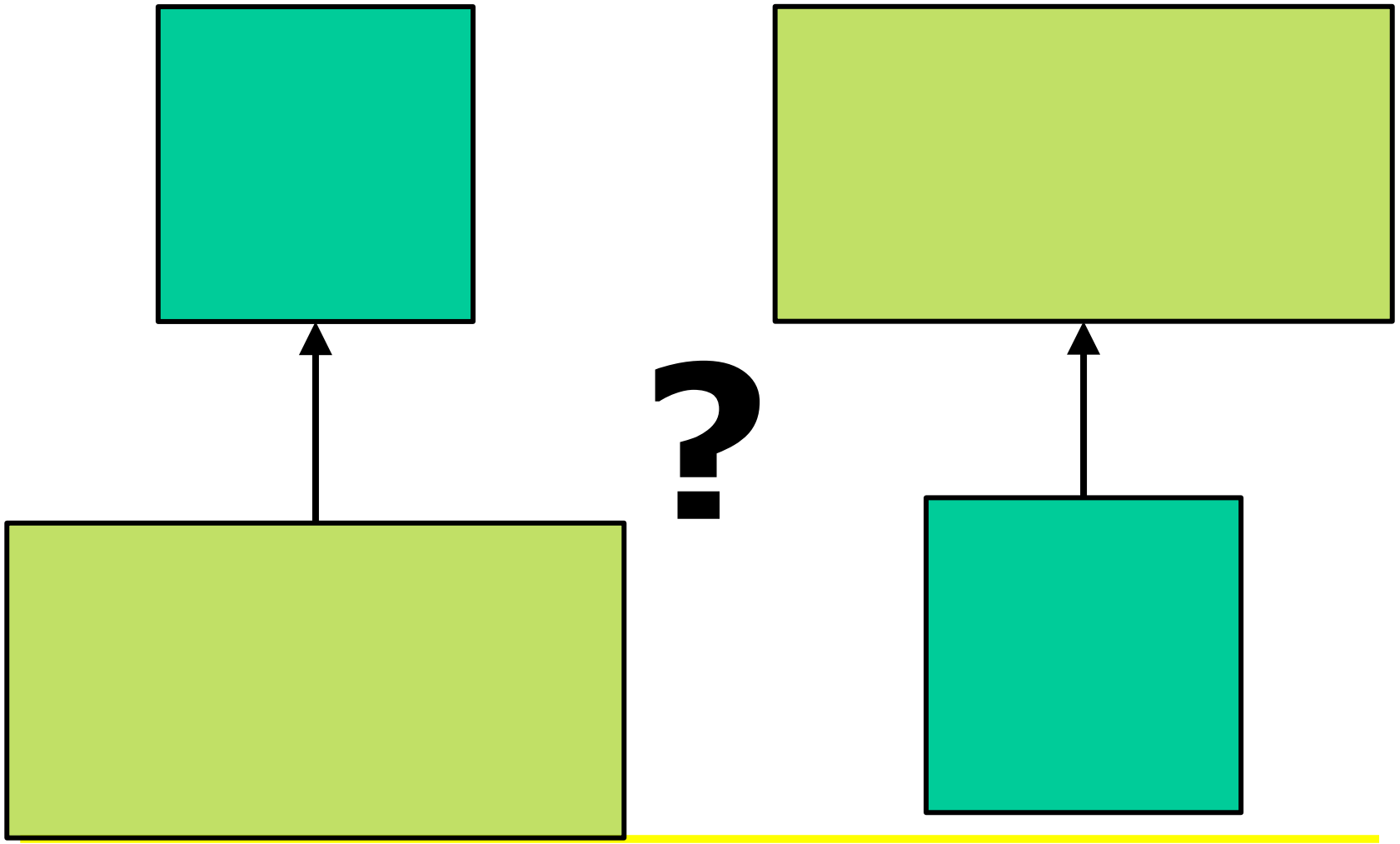
Principe de substitution de Liskov

Sous-classe = sous-type

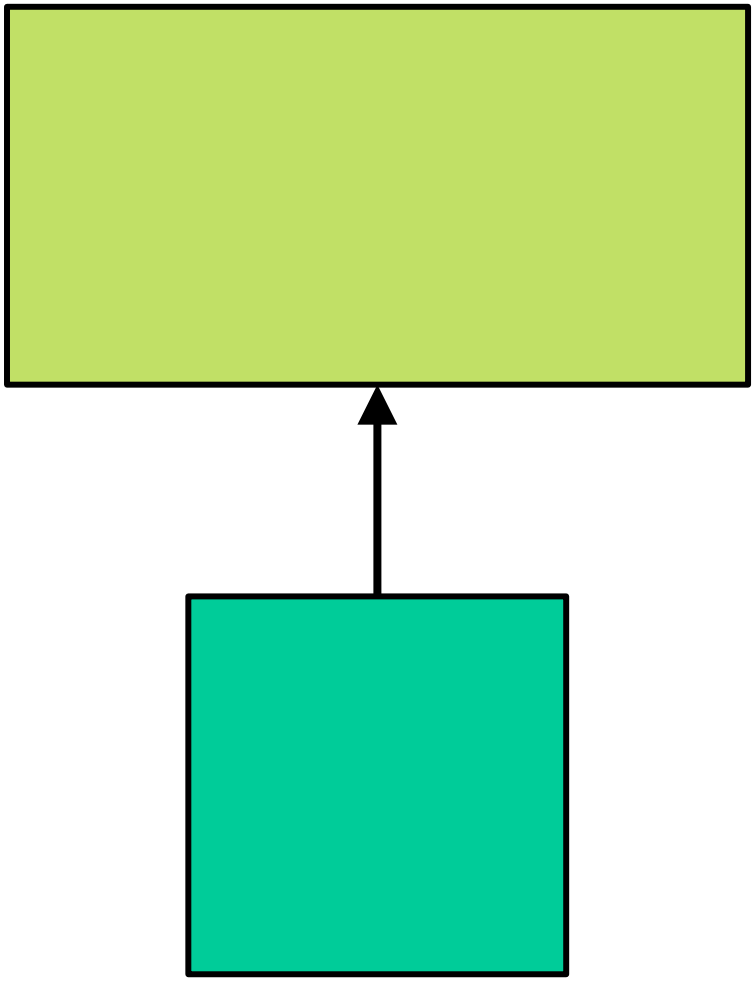
"B sous-type de A ⇔ pour chaque programme qui utilise des objets de classe A, on peut utiliser des objets de classe B à leur place sans que le comportement logique du programme change".

Une sous-classe ne peut pas contraindre le comportement des super-classes.

Exemple



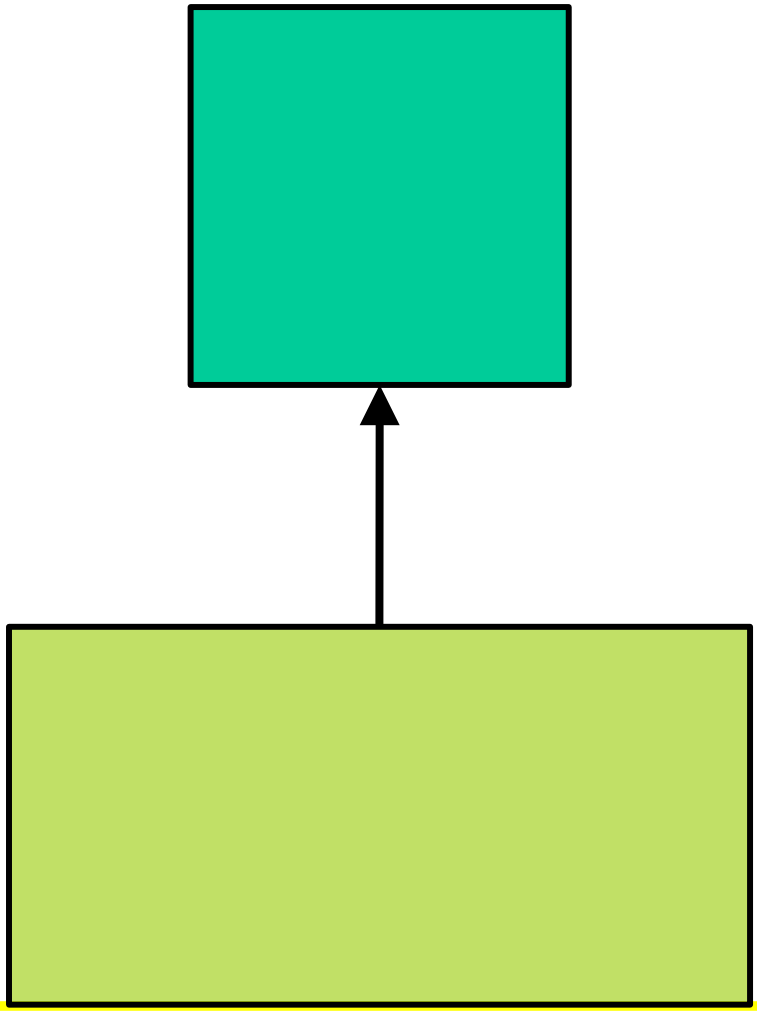
Exemple



protected double b, h;

setBase(double);
setHeight(double);

Exemple



protected double b;

setBase(double);

protected double h;

setHeight(double);

Types d'héritage

Deux notions cohabitent dans l'héritage :

- Héritage d'interface ou *subtyping*;
- Héritage de réalisation ou *subclassing*.

Héritage de réalisation

Il s'agit d'un mécanisme de réutilisation du code.

La sous-classe réutilise les méthodes des super-classes.

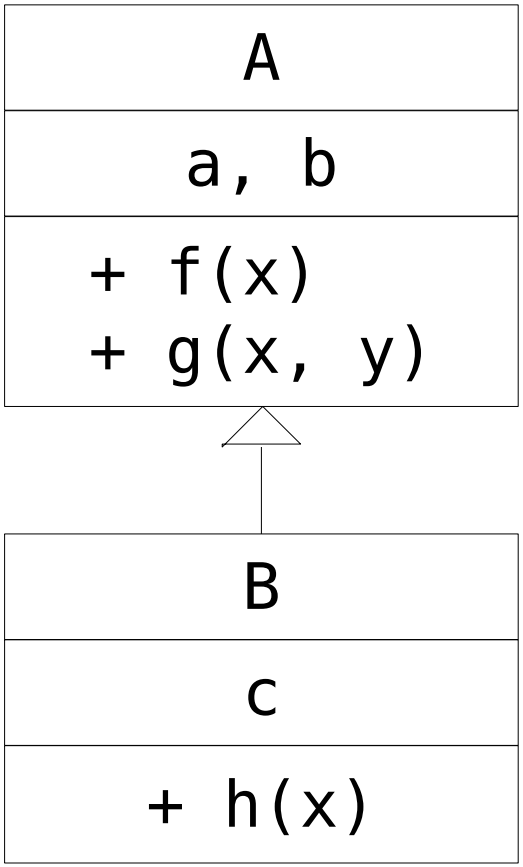
Héritage d'interface

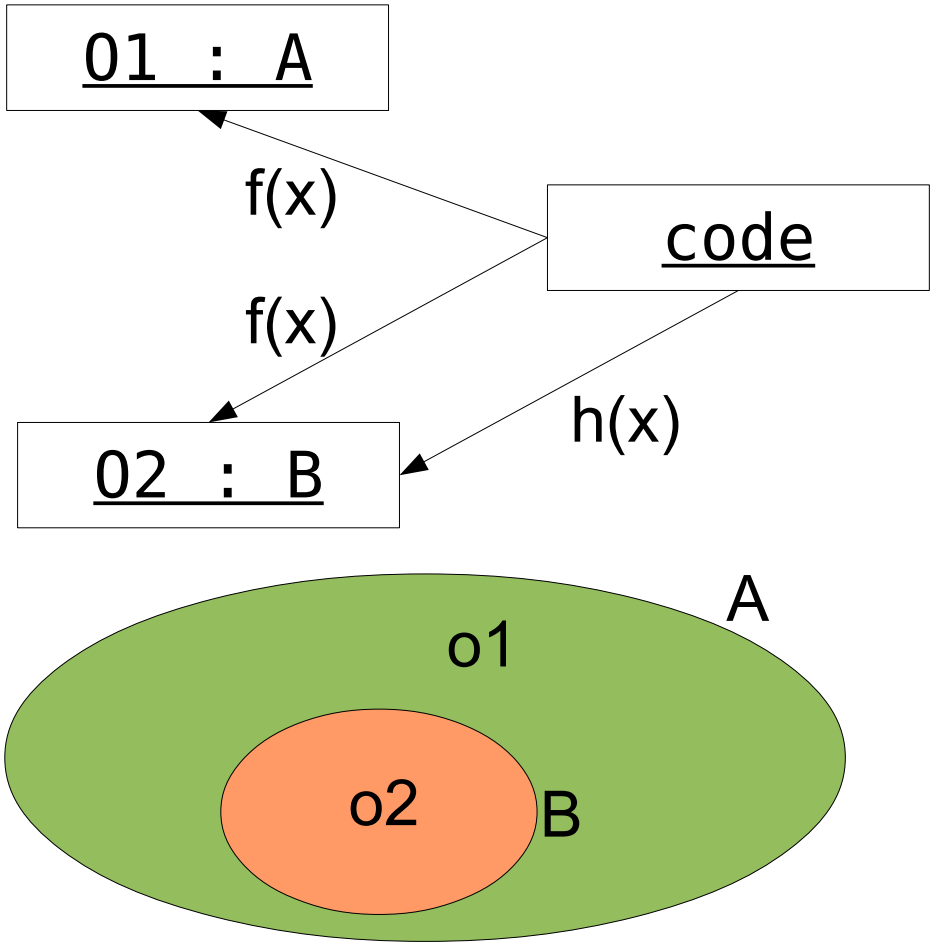
Il s'agit d'un mécanisme de compatibilité entre des types.

Il permet le polymorphisme par sous-typage.

Une sous-classe est un sous-type compatible vers le haut avec tous les types définis tout au long de sa chaîne d'héritage.

Héritage





Héritage multiple

Deux ordres d'avantages :

- fusion d'interfaces provenant de sources différentes ;
- réutilisation de code provenant de sources différentes.

Héritage d'interface :

- La correction peut être vérifiée statiquement (= au moment de la compilation);

Héritage de réalisation :

- Plus problématique : ex., la même méthode réalisée (comment ?) en deux super-classes distinguées.

Polymorphisme

Proprieté d'une même entité qui se manifeste dans des formes différentes dans de contextes différents…

In Informatique :

Idée d'autoriser le même code à être utilisé avec différents types, ce qui permet des implémentations plus abstraites et générales.

Exemple : monomorphisme

int **max**(int *a,* int *b*) :

    if(*a* > *b*) : return *a*

    else : return *b*

double **maxd**(double *a,* double *b*) :

    if(*a* > *b*) : return *a*

    else : return *b*

Exemple : monomorphisme

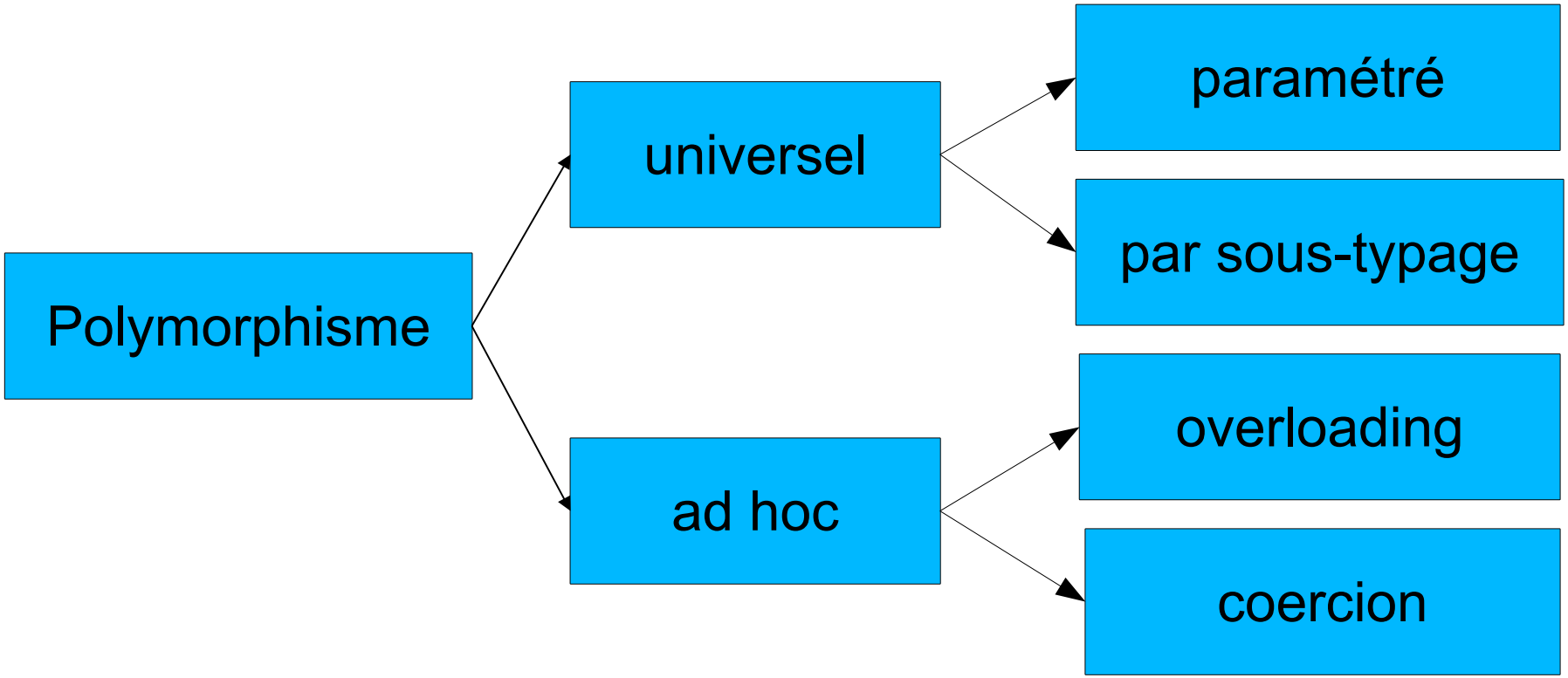
int **max**(int *a,* int *b*) :

```
if(a > b) : return a
else : return b
```

double **maxd**(double *a,* double *b*) :

```
if(a > b) : return a
else : return b
```

Classification selon Cardelli-Wegner



Overloading

La même fonction ou les mêmes opérateurs peuvent être appliqués à des types différents.

Exemple (en C):

```
double x, y, z;
int i, j, k;

z = x + y;
k = i + j;
```

Coercion

Les arguments d'une fonction ou opérateur sont transformés implicitement en le type applicable.

Exemple (en C):

```
double pow(double, double);
…
double x, y;
int n;
y = pow(x, n);
```

Polymorphisme paramétré

Fonctions et opérateurs paramétrés selon le type auquel il peuvent être appliqués.

Exemple (en C++):

```
template <typename T> T max(T a, T b)
{
    if(a > b) return a;
    else return b;
}
```

Typique de la "programmation générique".

Polymorphisme par sous-typage

- Une méthode (c'est-à-dire une opération) peut être appliquée à tous les objets qui appartiennent à la classe qui la prévoit dans son interface.
- Le principe de substitution de Liskov s'applique
- Typique de la programmation orientée objet

Exemple : max

```
interface Comparable
{
    int compareTo(Comparable c);
}
```

```
static Comparable max(Comparable a, Comparable b)
{
    if(a.compareTo(b) > 0) return a;
    else return b;
}
```

Merci de votre attention