Approches Objet de la Programmation

Surcharge, Masquage, Ré-écriture

Didier Verna

didier@didierverna.net

didierverna.net @didierverna didier.verna in/didierverna

Build 2024-11-11 11:28:40+01:00

Plan

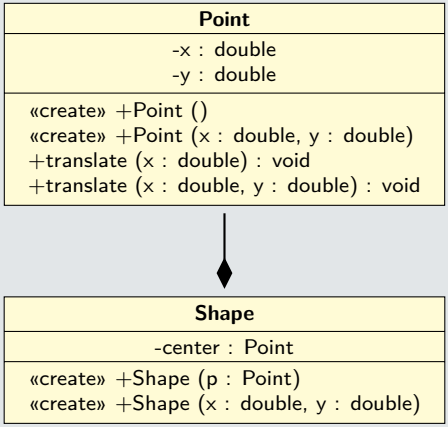
Plan

Plan

Surcharge

- Même nom, signature différente
  - Type / nombre de paramètres
  - Éventuellement, type de retour
- Constructeurs, méthodes
  - Y compris méthodes statiques
  - fonctions, opérateurs
- Utilité : faire...
  - ...des choses un peu différentes
  - ...la même chose différemment
- Polymorphisme « *ad hoc* » (intra-classe) [Strachey, 1967]
- Avantage : dispatch statique *compilation, performance*

Exemple



Surcharge

- Même nom, signature différente
  - Typ
  - Éve
- Construc
  - Y c
  - fonc
- Utilité :
  - ...de
  - ...la
- Polymor

  [Strachey,
- Avantag

  compilation, performance

Exemple

C++

```
class Point
{
public:
  // Constructor overloading
  Point ()                    : Point (0, 0)   {};
  Point (double x, double y) : x_ (x), y_ (y) {};

  // Method overloading
  void translate (double x)           { x_ += x; };
  void translate (double x, double y) { x_ += x; y_ += y; };

private:
  double x_, y_;
};
```

, y : double)
id
double) : void

«create» +Shape (x : double, y : double)

Surcharge

- Même nom, signature différente
  - Type / nombre de paramètres
  - Éve
- Construc
  - Y c
  - fonc
- Utilité :
  - ...de
  - ...la
- Polymor
  [Strachey,
- Avantage : dispatch statique
  compilation, performance

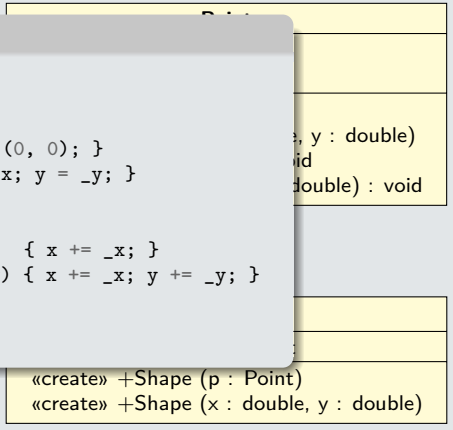
Java

```
public class Point
{
  // Constructor overloading
  public Point ()                          { this (0, 0); }
  public Point (double _x, double _y) { x = _x; y = _y; }

  // Method overloading
  public void translate (double _x)            { x += _x; }
  public void translate (double _x, double _y) { x += _x; y += _y; }

  private double x, y;
}
```

Exemple

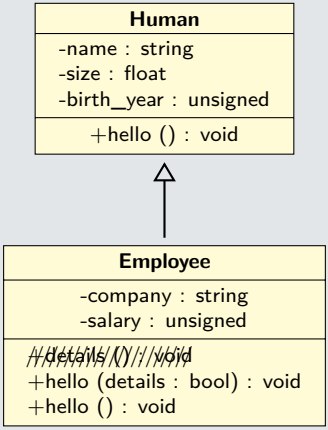


Plan

Masquage

- Polymorphisme inter-classe
- Utilité : identique à la surcharge
- Même nom, signature différente ou identique
  - Classe = facteur de distinction
- Y compris méthodes statiques
- Avantage : dispatch statique
  compilation, performance
- Inconvénient : dispatch statique... 😉

Exemple



Masquage

- Poly
- Util
- Mêr
- Y c
- Ava
  con
- Incc

C++

```
void Human::hello () const
{
  std::cout << "Hello! I'm " << name_ <<  ", "
            << size_ << "m, "
            << age () << "yo.\n";
}

void Employee::hello (bool details) const
{
  Human::hello ();
  if (details)
    std::cout << "Working at " << company_ <<  " for " << salary_ << "€, "
              << "started at the age of " << hiring_age () << ".\n";
}

void Employee::hello () const { hello (true); }
```

Masquage

- Polymorphisme inter-classe
- Utilité : identique à la surcharge
- Mêm
  - ...
- Y c
- Ava
  con
- Incc

C++

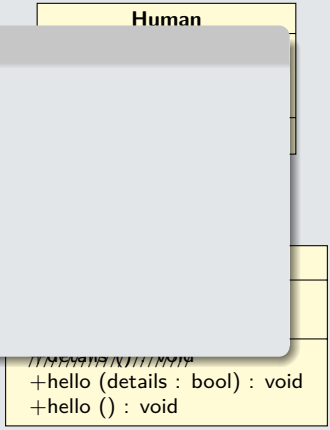
```cpp
auto h = Human { ... };
h.hello (); // Human::hello

auto e = Employee { ... };
e.hello (); // Employee::hello

Human* incognito = new Employee (...);
incognito->hello (); // Human::hello !

const Human& dont_know = unpredictable ();
dont_know.hello (); // Human::hello !
```

Exemple



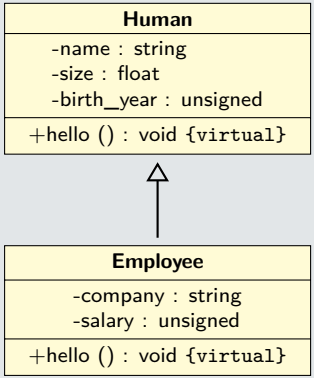
Plan

Plan

Ré-écriture

- *« We also needed to group together common process properties in such a way that they could be applied later, in a variety of different situations not necessarily known in advance. »* [Dahl, 1978]
- Polymorphisme « d'inclusion » (intra-hiérarchie)
- Utilité : *cf.* surcharge / masquage
- Même nom, même signature
- Avantage : dispatch dynamique
- Inconvénient : coût en performance
- Méthodes « virtuelles »

Exemple



Ré-écriture

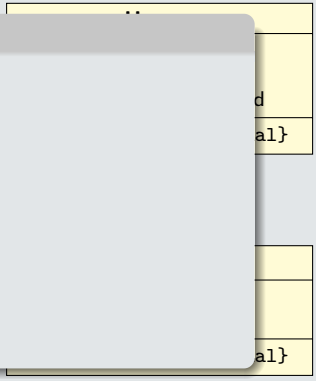
- *« We also needed to group together common process properties in such a way that they could be applied later, in a va*
- Poly
- Util
- Mêr
- Ava
- Incc
- Mét

C++

```
class Human
{
public:
  virtual void hello () const { ... };
};

class Employee : public Human
{
public:
  void hello (bool details) const { ... };
  void hello () const override { ... };
};
```

Exemple



Ré-écriture

Java

```java
public class Human
{
  public void hello ()
  {
    System.out.println ("Hello! I'm " + name + ", " + size + "m, "
                        + age () + "yo.");
  }
}

public class Employee extends Human
{
  public void hello (boolean details)
  {
    super.hello ();
    if (details)
      System.out.println ("Working at " + company + " for " + salary + "€, "
                          + "started at the age of " + hiringAge () + ".");
  }

  @Override public void hello () { hello (true); }
}
```

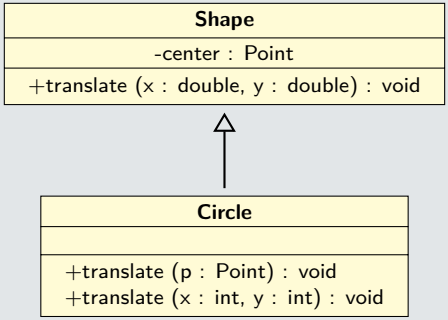
Plan

Build 2024-11-11 11:28:40+01:00

Surcharge, Masquage, Ré-écriture

- Propagation de la surcharge : langage-dépendant
  - Java : oui
  - C++ : non par défaut
    masquage intégral, cf. `using`
  - Indépendant du caractère virtuel ou statique
- Attention aux conversions de type !
- Masquage $\neq$ ré-écriture
  - Statique *vs.* dynamique
  - Java : méthodes statiques uniquement même signature
  - C++ : méthodes statiques *et* non virtuelles signatures identiques ou différentes

Exemple



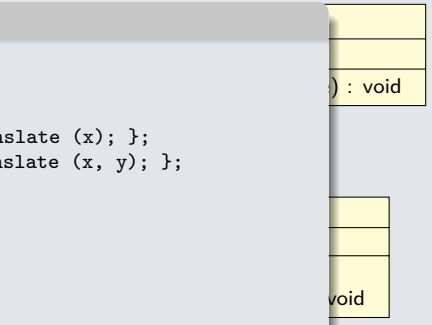
Surcharge, Masquage, Ré-écriture

- Propagation de la surcharge : langage-dépendant
  - C++
  -
  -
- Att
- Mas
  -
  -

```
class Shape
{
public:
  void translate (double x)           { center_.translate (x); };
  void translate (double x, double y) { center_.translate (x, y); };
};

class Circle : public Shape
{
public:
  using Shape::translate;
  void translate (Point p) { center_ = p; }
};
```

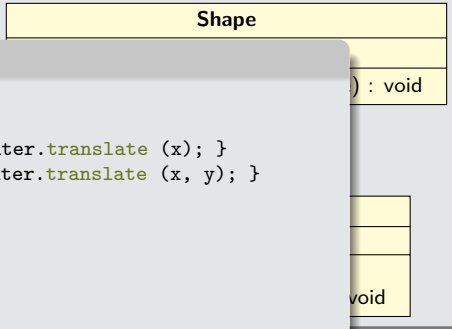
Exemple



Surcharge, Masquage, Ré-écriture

- Propagation de la surcharge : langage-dépendant
  - Java : oui
  - [illegible]
  - [illegible]
- Att[illegible]
- Mas[illegible]
  - [illegible]
  - [illegible]
  - C++ : méthodes statiques et non virtuelles
    signatures identiques ou différentes

Exemple



Java

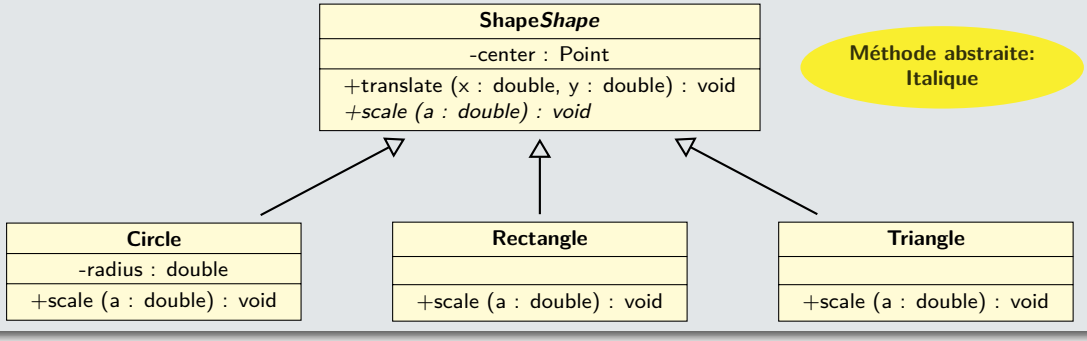
```
public class Shape
{
  public void translate (double x)           { center.translate (x); }
  public void translate (double x, double y) { center.translate (x, y); }
}

public class Circle extends Shape
{
  public void translate (Point p) { center = p; }
}
```

Plan

Méthodes abstraites

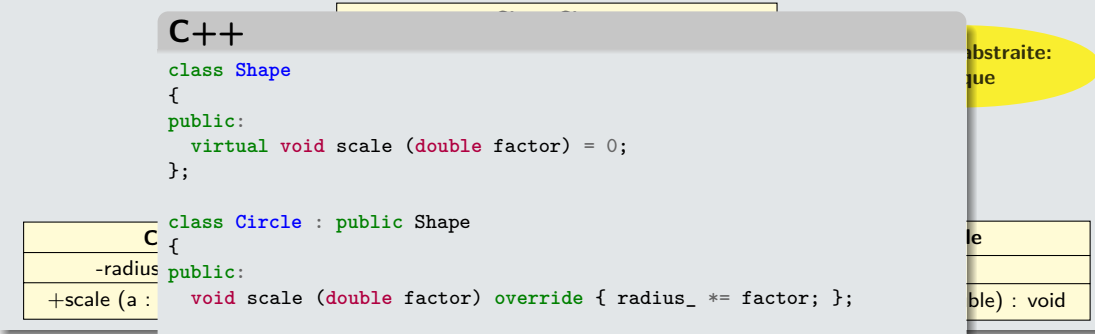
Exemple



- Méthodes *déclarées*, mais sans implémentation initiale (*a.k.a. « virtuelles pure »*)
- Implémentation nécessaire dans les sous-classes
- Méthode(s) abstraite(s) $\Longrightarrow$ Classe abstraite

Méthodes abstraites

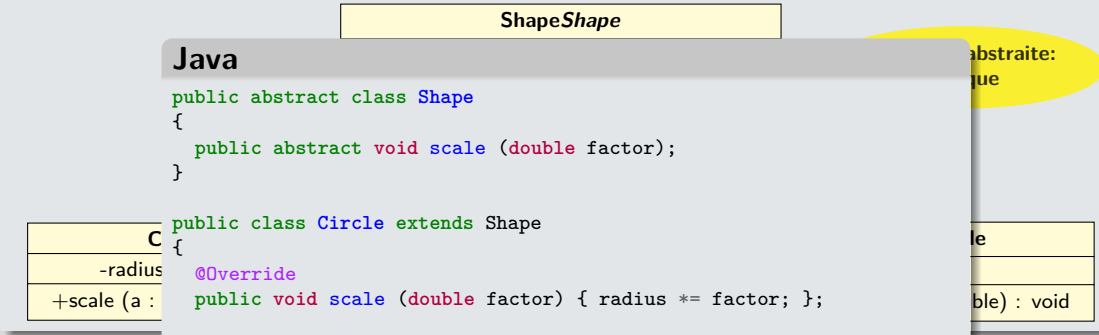
Exemple



- Méthode ... *ure »*)
- Impléme...
- Méthode(s) abstraite(s) $\Longrightarrow$ Classe abstraite

Méthodes abstraites

Exemple



Java

```java
public abstract class Shape
{
  public abstract void scale (double factor);
}

public class Circle extends Shape
{
  @Override
  public void scale (double factor) { radius *= factor; };

  private double radius;
}
```

- Méthode(s) abstraite(s) (« *virtual* » ou « *pure* »)
- Implémentation nécessaire dans les sous-classes
- Méthode(s) abstraite(s) $\Longrightarrow$ Classe abstraite

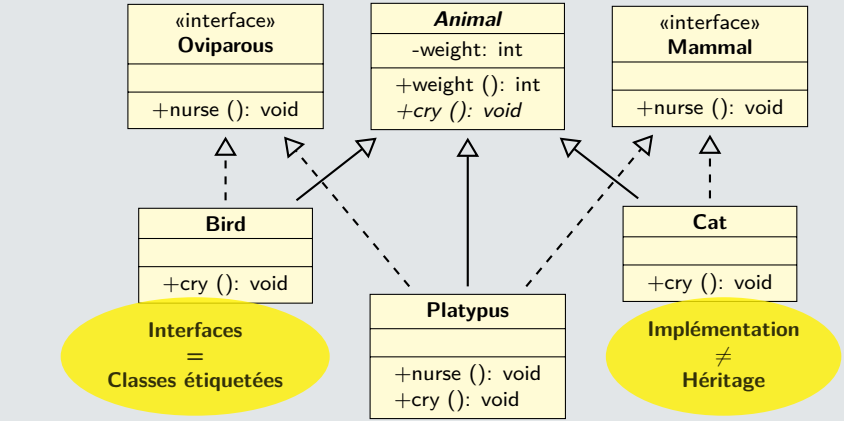
Plan

Corollaire : Interfaces Java

- Constantes et méthodes abstraites seulement
  - Tout est publique (mots-clés `abstract`, `public`, et `final` optionnnels)
  - Une classe `implements` une ou plusieurs interfaces
  - Un interface `extends` une ou plusieurs interfaces (héritage en diamant possible)
- Depuis Java 8
  - Méthodes statiques *non héritables* (implémentation requise)
  - Méthodes par défaut *héritables* (idem)
- Depuis Java 9
  - Méthodes privées *non héritables*, statiques ou non

Application

Exemple



Application

Exe

Java

```java
public interface Nurse
{
  public abstract void nurse ();
}
public interface Oviparous extends Nurse
{
  default void nurse ()
  {
    System.out.println ("I brood my eggs.");
  }
}
public interface Mammal extends Nurse
{
  default void nurse ()
  {
    System.out.println ("I suckle my offsprings.");
  }
}
```

Application

Exemple

«interface» **Animal** «interface»

Java

```java
public class Bird extends Animal implements Oviparous
{
  @Override
  public void cry () { System.out.println ("Tweet! Tweet!"); }
}

public class Cat extends Animal implements Mammal
{
  @Override
  public void cry () { System.out.println ("Meow! Meow!"); }
}
```

Classes étiquetées

+nurse (): void
+cry (): void

Héritage

Application

Exemple

Java

```java
public final class Platypus extends Animal
  implements Oviparous, Mammal
{
  @Override
  public void cry () { System.out.println ("Platty! Platty!"); }

  // Required for disambiguation, even if never called (!= C++)
  @Override public void nurse ()
  {
    // Default method calls
    Oviparous.super.nurse ();
    Mammal.super.nurse ();
  }
}
```

+cry (): void

Plan

Plan

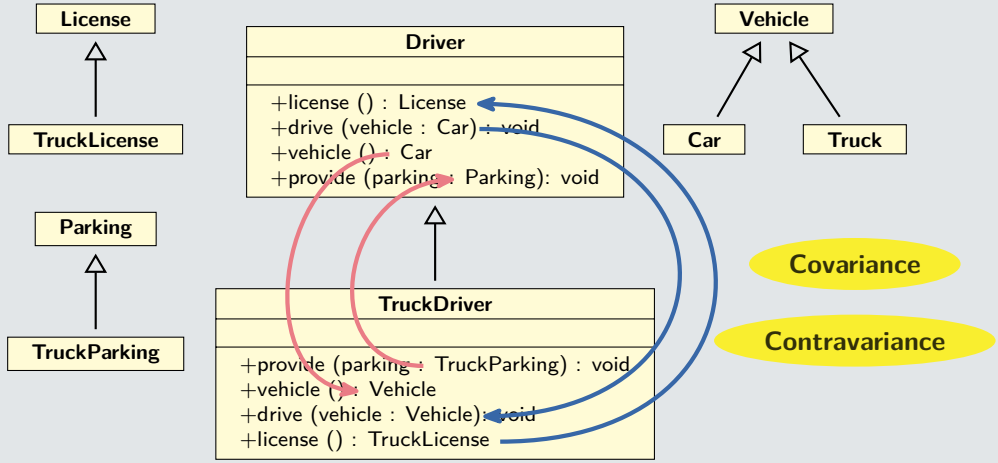
Rappels sur la Notion d'Héritage

- Relation de type « est un »
  - Tout objet d'une sous-classe peut être vu comme un objet d'une super-classe
- Ambivalence
  - Héritage d'implémentation
  - Héritage d'interface (entraîné par l'héritage d'implémentation)
- Principe de substituabilité
  - Si $S <: T$, alors tout terme de type S peut être utilisé en toute sécurité dans un contexte où un terme de type T est attendu
  - 😕 Pour une certaine définition de « en toute sécurité » et « contexte »...
- Relation forte entre héritage (sous-classage) et sous-typage

Plan

Application aux Méthodes

Exemple



Plan

Principe de Substitution de Liskov

- Principe de substituabilité appliqué aux objets
- Sous-typage *comportemental fort*

Soit $\phi(x)$ une propriété prouvable sur les objets x de type T.
Alors, $\phi(y)$ doit être vraie pour tout objet y de type S tel que $S <: T$.

- Covariance des types de retour dans S
- Contravariance des types d'arguments dans S
- Les exceptions levées dans S doivent être des sous-types de celles levées dans T
- Les invariants de T doivent être préservés dans S
- Les pré-conditions ne peuvent pas être renforcées dans S
- Les post-conditions ne peuvent pas être affaiblies dans S
- *Contrainte historique :* la mutation n'a lieu qu'à travers l'interface. Aucune méthode de S ne doit permettre des mutations interdites dans T.

Plan

Bibliographie

Bibliographie

Christopher Strachey.
Fundamental Concepts in Programming Languages.
International Summer School in Computer Programming, Copenhagen, 1967.
Higher-Order and Symbolic Computation, 13(1–2) : 11-–49, 2000.

Ole-Johan Dahl and Kristen Nygaard.
The Development of the SIMULA Languages.
History of Programming Languages Conference, 1978.

Barbara Liskov.
Data Abstraction and Hierarchy.
OOPSLA'87 Keynote, 1988.