Programmation objet en Java.

Didier Rémy
2001 - 2002

http://cristal.inria.fr/~remy/mot/7/
http://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/mot/7/

Slide 1

Cours	**Exercices**
1. Classes, Constructeurs	1. Surcharge
2. Héritage	2. Méthodes binaires
3. Surcharge	3. Abstraction de type
4. Les interfaces	4. Listes
5. Le sous-typage	5. Gestionnaire de fenêtres
6. Les conversions	
7. Les packages	

Avertissement

Ce n'est pas un cours sur Java, ni sur la programmation Objet. On suppose l'essentiel du langage Java déjà connu.

On rappel simplement les choix de Java concernant la programmation objet. La comparaison avec Ocaml est souvent implicite.

Pour une spécification complète se reporter à la définition du langage :

Slide 2

> *The Java Language Specification.*
> James Gosling, Bill Joy, and Guy Steele.

disponible aussi en HTML

Les classes

```
class Point {
  int x; int y = 0;
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; }
}
```

Une classe est nommée (`Point`) ; son corps est une liste de déclarations en nombre et en ordre quelconques :

Slide 3

- de variables (`x`), qui peuvent être introduites avec une valeur par défaut.
- de constructeurs (`Point`), dont le corps est de type `void` ; un constructeur est reconnu au fait que son nom est le même que celui de sa classe et que son type de retour n'est pas indiqué.
- de méthodes (`bouge`) dont le corps est arbitraire.

Par exemple, la classe `class vide {}` est une classe vide sans méthodes et sans constructeurs.

Les constructeurs

On crée une instance d'une classe avec la construction `new` suivie d'un constructeur de classe et de ses arguments, par exemple `new Point (3)`.

Le typage ne garantit pas que toutes les variables sont initialisées. Cependant, le compilateur affecte aux variables non initialisées une valeur par défaut du type déclaré.

Slide 4

```
class Point {
  int x; int y = 0;
  Point () { }
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; }
}
```

est une définition correcte et `(new Point()).x` retourne 0.

This

Dans le corps d'une méthode, `this` désigne l'objet en train d'exécuter cette méthode.

Slide 5

```
class Point {
  int x = 0;
  Point () { }
  Point (int x0) { x = x0; }
  void bouge () { x = x + 1; }
  void bouge_bouge () { this.bouge(); this.bouge(); }
}
```

L'héritage

```
class Bipoint extends Point { int y;
  Bipoint () { }
  Bipoint (int x0, int y0) { super(x0); y = y0; }
  void bouge () { super.bouge (); y = y + 1; }
}
```

Slide 6

Dans une sous-classe, `super` désigne la classe parente. Un constructeur d'une sous-classe doit faire appel à un constructeur de la classe parente. Cet appel doit être la première instruction. Si ce n'est pas fait, le compilateur ajoute `super()`. Ainsi,

```
  Bipoint (int y0) { y = y0; }
  Bipoint (int y0) { super(); y = y0; }
```

sont deux phrases équivalentes (et c'eut été une erreur si `Point() { ... }` n'était pas défini dans la classe `Point`.)

La surcharge

Les constructeurs de classe et les méthodes sont surchargés.

Les constructeurs de classes ne sont pas hérités.

Les méthodes sont héritées. Ainsi, l'ensemble des définitions d'une méthode `m` dans une classe `A` est l'ensemble des définitions de `m` dans la classe `A` et dans la classe parente.

Slide 7

Lorsqu'une sous-classe redéfinit une méthode présente dans la classe parente avec le même type, la nouvelle définition cache (*override*) la précédente. Sinon elle ne fait que la surcharger.

Pour la résolution de la surcharge, voir le cours sur la liaison surchargée.

Interfaces et implémentations

Les interfaces sont des spécifications des implémentations.

```
interface Printable {
  void print ();
}
```

L'interface `Printable` n'est pas une classe. Elle n'a pas d'implémentation propre. Mais l'interface peut être implémentée (*i.e.* respectée) par différentes classes.

Slide 8

Une classe peut implémenter plusieurs interfaces.

```
class Printable_point extends Point implements Printable {
  Printable_point () { super(); }
  public void print () { System.out.println(x); }
}
```

Utilisation

La déclaration d'une interface I dans la définition d'une classe A :
– effectue des vérifications de typage.

– déclare le type des objets de la classe A un sous-type du type de l'interface I.

Cela permet par exemple, de voir tous les objets imprimables avec un type commun `Printable` et de leur envoyer des messages.

On peut donc écrire une méthode

```
  void print (Printable p) { p#print; }
```

qui imprime les objets imprimables.

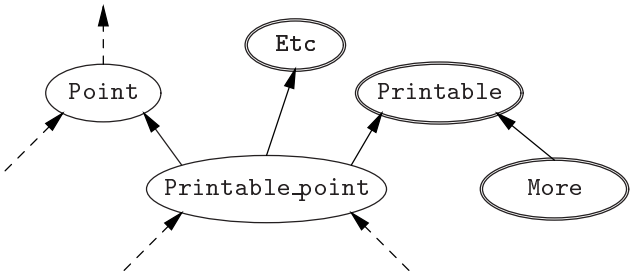
Slide 9

Interfaces et implémentations

La relation de sous-typage Une classe ne peut hériter que d'une seule classe, mais elle peut simultanément implémenter une ou plusieurs interfaces.

La relation d'héritage est un arbre, mais la relation de sous-typage est un graphe (treillis)

Slide 10



Le sous-typage

En java, le sous-typage est par nom.

- Une déclaration de classe `class A extends B implements` $B_1, ...B_k$ déclare un nouveau type `A`, et le déclare sous-type de `B` et de chaque B_i.
- Une déclaration d'interface `interface A extends` $B_1, ...B_k$ déclare un nouveau type `A`, et le déclare sous-type de chaque B_i.

La relation de sous-typage est fermée par reflexivité et transitivité.

Slide 11

Les casts

Le cast d'un objet change son type statique, mais ne change pas sa représentation (valeur dynamique).

Un cast se note `(A) (v)` où `A` est le type cible et `v` l'argument.

Lorsque le type statique de `v` est un sous-type de `A`, alors le cast est statique, et il réussit toujours.

Slide 12

Lorsque le type statique de `v` est un sur-type de `A`, alors il est possible que le type dynamique de `v` soit en fait plus précis et un sous-type de `A`. Le cast est autorisé, mais il nécessite un test à l'exécution qui peut échouer.

Tout `cast` qui peut réussir à l'exécution est permis.

Lorsqu'un cast `(cast)(v)` est rejeté à la compilation, il est toujours possible de le forcer par `(cast)((Object)(v))`, mais ce cast est certain de produire une erreur à l'exécution (sinon, il n'aurait pas été rejeté à la compilation) !

Les conversions

Un cast effectue une conversion de type explicite.

Certaines conversions de types sont implicites. Par exemple, une affectation d'une valeur à une variable de type `A` effectue une conversion implicite vers le type `A` pourvu que la valeur soit un sous-type de `B`. Les conversions implicites ne peuvent pas échouer à l'exécution.

Slide 13

Les règles de conversion sont complexes (voir The Java Language Specification pour une description complète...)

(Voir exercise sur la surcharge)

Cast (exemple)

Il y a essentiellement trois types d'utilisation des `cast` :

- Les conversions statiques, pour changer le comportement d'un programme. En affaiblissant le type statique d'un objet vers le type des objets d'une classe parente, on change la résolution de la surcharge. L'objet se comporte *statiquement* comme s'il était de la classe parente.
- Les conversions dynamiques mais sûrs. Pour pallier une faiblesse du système de typage, il y a des situations où le type statique d'un objet est celui d'un classe parente de sa classe réelle. L'utilisateur peut le savoir (parce qu'il connait le code et peut faire un calcul mental) alors que le compilateur ne sait pas le vérifier statiquement (sans faire de calcul).
- Les tests de classes. Lorsque des objets de plusieurs classes sont mis dans un conteneur, à leur retrait, il faut parfois renforcer leur type à l'aide d'une conversion combinée avec un typecase.

Slide 14

Exercise 1 (Méthodes binaires) *On considère le programme suivant en Ocaml.*

```
class point x0 =
  object (this : 'mytype)
    val x : int = x0
    method getx = x
    method min (z : 'mytype) =
      if this#getx < z#getx then this else z
  end;;
```

Slide 15

```
class bipoint x0 y0 =
  object (this : 'mytype)
    inherit point x0 as super
    val y : int = y0
    method gety = y
    method min z =
      if this#gety < z#gety then this else super#min z
  end

let p = new bipoint 1 2;;
```

```
let q = new bipoint 2 1;;
let r = p#min q in r # gety;;
```

Écrire en Java le programme le plus proche possible. Commenter, et discuter éventuellement les différentes solutions. Answer □

Slide 16

Typecase

La construction `v instanceof A` permet de tester si l'objet est d'une sous-classe de la classe `A`.

En fait `v instanceof A` retourne `true` si et seulement si l'expression `(A) (v)` ne produira pas d'exception.

De plus l'une ou l'autre expression sont rejetées par le compilateur dans exactement les mêmes conditions.

Slide 17

Deux utilisations typiques

- En conjonction avec une conversion, pour recouvrer le vrai type d'un objet qui a été perdu (en général suite au sous-typage).
- Seul, pour distinguer des objets de classes différentes (vues sous une même interface compatible.)

Les packages

Les packages sont un moyen de grouper plusieurs classes ensembles, en limitant la visibilité de certains composants à un package.

Les composants des packages sont des classes, des interfaces, ou des sous-packages.

Slide 18

La fabrication et l'utilisation des packages dépend du système sous-jacent (conventions de nommage, de placement, et d'accès), mais le langage fixe les déclarations correspondantes dans le code source, ainsi que les règles de visibilité.

Les packages

Utilisation Pour accéder aux composantes `x`, `y` d'un package `P` on utilise la notation pointée `P.x`, `P.y`, *etc.*

La déclaration `import P.Q.x;` rend visible la déclaration `x` du package `P.Q` dans l'unité de compilation sans avoir à utiliser la notation `P.Q.x`.

Slide 19

La déclaration `import P.Q.*;` rend visible toutes les déclarations du package `P.Q` dans l'unité de compilation.

Fabrication La déclaration `package P.Q;` rend l'ensemble des déclarations qui suivent comme une partie du package `P.Q`.

Tout programme fait partie d'une unité de compilation nommée ou anonyme.

Les règles de visibilité

Classes, champs, méthodes et constructeurs peuvent être :
- `public`, *i.e.* visible partout ;
- non annoté, *i.e.* visible dans le package (nommé ou annomyme) auquel appartient l'unité de compilation.

De plus, un champ, une méthode ou un constructeur peut être :
- `private`, *i.e.* visible seulement dans la classe dans laquelle il est défini.

Slide 20

- `protected`, *i.e.* visible dans le package et dans les sous-classes de celle dans laquelle il est défini si celle-ci est publique.

La redéfinition dans une sous-classe d'une méthode d'une classe parente doit avoir une visibilité au moins aussi grande.

Contrainte Une classe `A` utilisée dans un autre fichier que celui où elle est définie doit être définie dans un fichier du nom `A.java`.

Type abstraction

Les règles de visibilité fournissent une forme faible de type abstrait.

En effet, contrairement à Ocaml, les composantes privées d'une classe `C` restent visibles dans la définition de la classe : en particulier dans le corps de la classe `C` un object de la classe `C` même autre que *this*, expose ses composantes privées.

Slide 21

Dans tout autre classe (y compris une sous-classe de `C`) les composantes privées de C ne sont pas visibles. Ainsi, le type d'une classe est partiellement abtrait (les méthodes privées ne sont pas visibles) en dehors de la classe où il est défini.

En Ocaml, la seule façon d'obtenir cet effet est d'utiliser le mécanisme d'abstraction des modules. La méthode est laissée publique mais son type est rendu abstrait.

Exercise 2 *On donne le programme suivant :*

```
class Codage {
  private int code;
```

```
  Codage (int n) { code = n; }
  boolean decode (Codage z) { return (z.code == code); }
  }

class Test_Codage {
  public static void main (String arg[]) {
  Codage p = new Codage (1);
  Codage q = new Codage (2);
  if (p.decode(q)) System.out.println("Please, enter");
  else System.out.println("Go away!");
  }
}
```

Slide 22

Vérifier que la variable d'instance `code` *n'est pas accessible dans les objets codés.*

Donner un programme en Ocaml assurant le même niveau de sécurité. Answer □

Packages (exemple)

Slide 23

P/A.java

```
package P;
class A {
  private int x;
  public int y = x;
  A() { }
}
```

P/A.java

P/B.java

```
package P;
public class B {
  public A a = new A();
  protected int m ()
    { return a.y; }
}
```

P/B.java

P/C.java

```
package P;
public class C {
  B b = new B();
  int m () { return b.m(); }
}
```

P/C.java

User.java

```
import P.B;
class User extends P.B {
  User () { m(); }
  B b = new B();
  int bm () { return b.m(); }
  int bay () { return b.a.y; }
}
```

User.java

Packages (exemple)

Visibilités

Slide 24

Classe	Importé	Publique	Package	Classe
A		A.y	A	A.x
B	A, A.y	B, B.a	B.m	
C	A, A.y, B, B.a, B.m	C	C.b, C.m	
User	P.B	User		

Errors

(1) La méthode `m` de la classe `B` est protégée dans `b.m`.

(2) Le type `A` de `b.a` n'est pas publique dans `b.a.y`.

La méthode `m` de la classe `User` est celle de la class `B`. Elle est visible dans `User` qui est une sous-classe de la classe `B` visible ; mais elle n'est pas visible dans un objet de la classe `B` elle-même.

Exercices

Reprendre quelques exercices Ocaml en Java...

Exercise 3 (Listes) *Écrire les listes sous forme de classe. Les spécialiser avec un constructeur `Append`.* □

Exercise 4 (Gestionnaire de fenêtres) *Reprendre le modèle maitre/esclave et le gestionnaire de fenetre.* □

Slide 25