Classes Internes & Enumération

Rémi Forax

POO en Java

La POO en Java est basée sur 3 concepts

- Polymorphisme
- Interface pour le typage
- Classe pour l’encapsulation

La notion de classe interne est un raffinement de la notion de classe et d’encapsulation

Classe interne / locale / anonyme

Java 1.0: une classe publique doit être définie dans un fichier ayant le même nom

Java 1.1: une classe peut, de plus, être définie

- dans une classe, en tant que **classe interne**
- dans une méthode, en tant que **classe locale** ou **classe anonyme**

Classe interne / locale / anonyme

Classe interne

```
class Foo {
  class Bar { … }  // classe interne
}
```

Classe locale

```
void method() {
  class Bar { ... }  // classe locale
}
```

Classe anonyme

```
void method() {
  Itf I = new Itf() { … };   // classe anonyme
}
```

Classe Interne

Classe Interne: Java vs C

Une classe interne en Java

range les classes à l'intérieur d'une autre

=> change la visibilité

passe outre la rêgle

un fichier .java <==> une classe

```
class Foo {
  class Bar { … }  // visible si Foo est visible
}
```

Classe interne est un membre

Comme un champ ou une méthode, une classe interne est un **membre** de la classe englobante

- Elle accepte donc les modificateurs de visibilité
  - private, (package), protected, public

```
public class Foo {
  private class Bar {  // pas visible de l'extérieur
    ...
  }
}
```

Nommage

Une classe interne a son nom préfixé par le nom de la classe englobante

```
package fr.umlv.example;

class Foo {
  class Bar {
  }
}
```

la classe Bar s’appel fr.umlv.example.Foo.Bar

Visibilité

La notation de private s'étend aux classes internes

- Une classe englobante voit les membres privés de la classe interne
- Une classe interne voit les membres privés de classe englobante

```
class Foo {
  private int a;
  void m(Bar bar) {  bar.b /* OK */ }
  class Bar {
    private int b;
    void m(Foo foo) { foo.a /* OK */ }
  }
}
```

Pour la VM

La machine virtuelle n’a pas de notion de classe interne

Sur le disque la classe interne et la classe emglobante sont deux fichiers séparés

Le compilateur ajoute

dans la classe englobante

- un attribut NestMembers avec la liste des classes internes

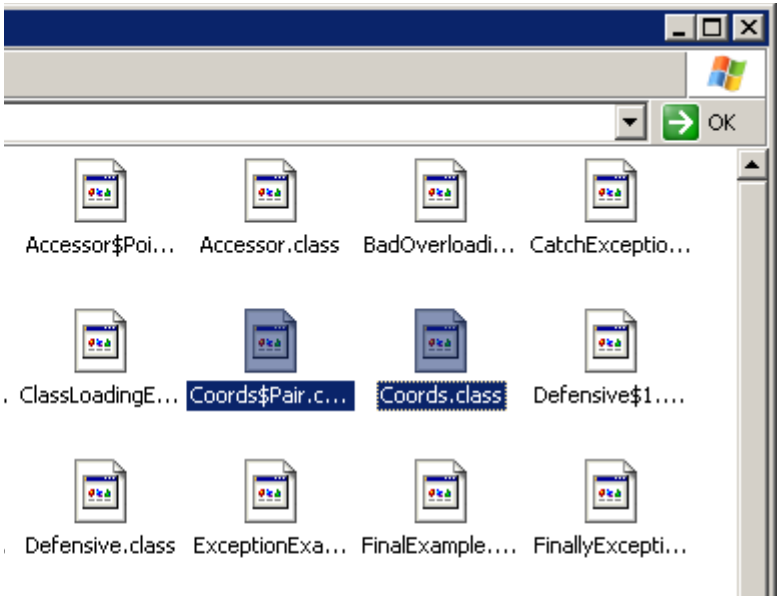
dans la classe interne

- un attribut NestHost avec le nom de la classe englobante

pour indiquer qu’une classe interne et sa classe englobante sont partie du même *nest*

Sur le Disque

```
class Coords {
  class Pair {
  }
}
```



L'implantation actuelle utilise le symbole '$'

- Coords.class est la classe englobante
- Coords$Pair.class est la classe interne

Classe interne statique / *inner class*

Classe interne statique

```
class Foo {
  static class Bar { … }  // static internal class
}
```

Classe interne d'instance (*inner class*)

```
class Foo {
  class Bar { ... }  // inner class
}
```

Classe Interne Statique

Classe interne statique

Le mot-clé static veut ici dire indépendant de la classe englobante

```
class Foo {
  static class Bar { … }
}

…
Foo.Bar bar = new Foo.Bar();
```

la classe Foo n’est pas chargée en mémoire

Classe Interne statique: Java vs C

En C, une struct à l'intérieur d'une struct a la même durée de vie que la struct englobante

En Java, la classe interne statique et la classe englobante sont indépendantes

Classe interne statique et membre d'instance

Une classe interne statique ne peut pas accéder directement aux membres non-statiques de la classe englobante

```
class Foo {
  int a;
  static int b;

  static class Bar {
    public void m() {
      … a … // erreur de compilation
      … b … // ok, it's Foo.b
    }
  }
}
```

Inner Class

Inner Class

Contrairement à une classe interne statique, une *inner class* (classe interne non-statique) possède une référence cachée sur la classe englobante

```
class Foo {
  int a;
  static int b;

  class Bar {  // non static
    public void m() {
      … a … // ok   <- there is something magic here
      … b … // ok, it's Foo.b
    }
  }
}
```

Délégation Implicite

Une *inner class* possède un champ caché qui contient une référence sur la classe englobante

Syntaxe pour accéder au champ caché: EnclosingClass.this

```
class Foo {
  int a;
  class Bar {
    // Foo Foo.this;   // hidden field !
    public void m() {
      … a … <==> … Foo.this.a ...
    }
  }
}
```

Inner Class : Instantiation

Une instance de la classe englobante doit être disponible lors de la création d'une instance de l'*inner class*

```
class Foo {
  class Bar { … } // non static

  public static void main(String[] args) {
    Foo.Bar bar = new Foo.Bar();  // compile pas

    Foo foo = new Foo();
    Bar bar = foo.new Bar();
  }
}
```

Inner Class : Instantiation (2)

Dans la classe englobante, créer une instance d'un classe interne peut utiliser "this" implictement

```
class Foo {
  class Bar { … } // non static

  public void m() {
     var bar = new Bar();  // ok <=> this.new Bar();
  }
}
```

This vs Foo.this

Ne pas confondre “this” et “Foo.this”

this est la référence sur laquelle on appelle la méthode

Foo.this est la référence sur laquelle on a créé l’instance de la classe interne

```
class Foo {
  class Bar {
     public void m() { … this … Foo.this ... }
  }
  public static void main(String[] args) {
     var foo = new Foo();
     var bar = foo.new Bar();
     bar.m();
  }
}
```

Inner class : membre statique

Une *inner class* ne peut pas définir de membre statique

```
class Foo {
  class Bar {
    static int a;  // erreur
    static void m() { … } // erreur
    static class Baz { … } // erreur
  }
}
```

Il y a pas de vrai raison, cela peut changer dans une version future de Java (prévu pour 16)

Inner class vs record/enum, ...

Un record ou un enum interne est toujours statique

```
class Foo {
   record Bar(int a) { … }  // implicitement static
}
```

Une classe interne est toujours static à l'intérieur d'une interface

```
interface Foo {
  class Bar { … } // implicitement static
}
```

Classe Locale

Classe Locale

Une classe locale est une classe définie dans une méthode

```
class Foo {
  void m() {
    class Bar { … }
  }
}
```

Elle n’est visible qu’à l’intérieur de la méthode

Classe Locale == *Inner Class*

Une classe locale accède à tous les membres de la classe englobante

```
class Foo {
  int a;

  void m() {
    class Bar {
      void hello() {  … a … }  // <==> Foo.this.a
    }
    new Bar().hello();   // <==> this.new Bar().hello()
  }
}
```

Classe locale et variable locale

Comme une lambda, une classe locale peut capturer la valeur d'une variable locale si elle est effectivement final

```
interface Expr {
  int result();

  public static Expr add(Expr left, Expr right) {
    class Add implements Expr {
      public int result() {
        return left.result() + right.result();
      }
    }
    return new Add();
  }
}
```

Classe locale vs record/enum

Un record ou un enum peut être local, il est alors static

- Ne peut pas accéder aux membres d'instance
- Ne peut pas accéder aux variables locales

```
interface Expr {
  int result();
  public static Expr add(Expr left, Expr right) {
    record Add(Expr left, Expr right) { … }
    return new Add(left, right);
  }
}
```

Classe locale static ?

Il n’est pas possible de déclarer une classe locale statique

devrait être possible avec Java 16 !

```
class Foo {
  void m() {
    static class Bar { … }  // erreur
  }
}
```

Sealed Interface

Une classe locale ne peut pas implanter une interface scellée

Une interface scellée est censée lister (permits) l'ensemble de ses implantations !

```
sealed interface Foo {
  static void m() {
     class Bar implements Foo { … }  // erreur
  }
}
```

Classe Anonyme

Classe Anonyme

Une classe anonyme est une classe locale sans nom

- elle doit implanter/hériter une interface/classe
- possède une syntaxe spécial new Type(…) { … }

```
class Foo {
   void m() {
     Foo foo = new Foo() { … };
   }
}
```

Classe Anonyme == Inner Class

Une classe anonyme accède à tous les membres de la classe englobante

```
class Foo {
  int a;

  void m() {
    new Object() {
      void hello() {  … a … }  // <==> Foo.this.a
    }.hello();
  }
}
```

Classe Anonyme et var

Comme la classe n’a pas de nom, elle ne peut pas être référencée comme un type

mais on peut utiliser “var”

```
class Foo {
  void m() {
    Object o = new Object() { void hello() { ... } };
    o.hello();  // compile pas

    var o2 = new Object() { void hello() { ... } };
    o2.hello();  // Ok !
  }
}
```

Classe anonyme et variable locale

Comme une classe locale, une classe anonyme peut capturer la valeur d'une variable locale si elle est effectivement final

```
interface Expr {
  int result();

  public static Expr add(Expr left, Expr right) {
    return new Expr() {
      public int result() {
        return left.result() + right.result();
      }
    };
  }
}
```

Relation avec les lambdas

Implanter une interface

Classe Interne

```
class Foo {
  class Bar implements Itf { public void m(…) { … } }
}
```

Classe Locale

```
Itf m() {
  class Bar implements Itf { public void m(…) { … } }
  return new Bar();
}
```

Classe Anonyme (un seul supertype uniquement)

```
Itf m() {
  return new Itf() { public void m(…) { … } };
}
```

Lambda (interfaces fonctionnelles uniquement)

```
Itf m() {
  return (...) -> { … };
}
```

Enumération

Type énuméré

Un type énuméré (enum en Java) est un type qui liste **l'ensemble des instances** d'un type par leur **nom**

enum Pet { DOG, CAT, LION }

DOG, CAT et LION sont des instances de Pet (le new est implicite)

Enum == Classe

En Java, un enum est une classe qui hérite implicitement de java.lang.Enum

La syntaxe est rétro-compatible avec celle du C

```
enum Pet { DOG, CAT, LION }
```

Après un point virgule, on peut ajouter des champs et des méthodes

```
public enum Pet {
  DOG, CAT, LION;

  public boolean isDangerous() { return this == LION; }
}
```

Ordinal() et name()

java.lang.Enum définе 2 champs

- int ordinal qui indique le numéro de l'instance (à partir de 0)
- String name qui indique le nom de la constante

```
enum Pet { DOG, CAT, LION }

assert Pet.DOG.ordinal() == 0;
assert Pet.CAT.name().equals("CAT");
```

Type fermé

Un enum est un type fermé, il n’est pas possible d’ajouter des instances a celles prédéfinies

Ce n’est donc pas un type extensible
(Il est soit **final** soit **sealed**)

Un enum ne participe pas à l’héritage

- On ne peut pas en hériter
- Il ne peut pas hériter d’une classe

.values()

Il est possible d’énumérer toutes les valeurs possible en utilisant la méthode values()

```
enum Pet { DOG, CAT, LION }

…
Pet[] pets = Pet.values();
```

Attention performance, comme values() renvoie un tableau, il y a une copie défensive qui est faite à chaque appel

.valueOf()

Il est possible de demander la constante en fonction de son nom avec la méthode valueOf()

```
enum Pet { DOG, CAT, LION }

…
Pet dog = Pet.valueOf("DOG");
```

Les méthodes values() et valueOf() sont automatiquement ajoutées par le compilateur

Constructeur d’un enum

Un constructeur est soit private soit de package

```
public enum Coin {
  DIME(10), QUARTER(50);

  private final int value;

  private Coin(int value) {
    this.value = value;
  }
}
```

Java permet de créer des enums mutables mais c’est une très mauvaise idée !

Constante d’Enum et la syntaxe des classes anonymes

Syntaxe des classes anonymes

Une constante peut utiliser la syntaxe des classes anonymes

```
public enum Pet {
  DOG {
    public void sound() { System.out.println("bark"); }
  }, CAT {
    public void sound() { System.out.println("meow"); }
  }, LION {
    public void sound() { System.out.println("roars"); }
  };

  public abstract void sound();
}
```

Syntaxe des classes anonymes (2)

Si une constante utilise la syntaxe des classes anonymes, elle est une sous-classe de la classe de l'enum

Dans ce cas, l'enum est déclaré sealed au lieu d'être déclaré final

Délégation au lieu de l’héritage

En fait, on peut utiliser la délégation,
ce qui est plus propre !

```
public enum Pet {
  DOG(() -> { System.out.println("bark"); }),
  CAT(() -> { System.out.println("meow"); }),
  LION(() -> { System.out.println("roars"); });

  private final Runnable action;

  private Pet(Runnable action) { this.action = action; }

  public void sound() { action.run(); }
}
```