

Classe Interne, Anonyme & Enumération

Rémi Forax
forax@univ-mlv.fr

Rappel

Nous nous intéressons aujourd'hui à deux formes de type abstrait en Java

- Les interfaces
- Les classes abstraites

Les classes internes, anonymes et les énumérations sont des types permettant de définir facilement des type concrets implantant ces types abstraits

Classe internes

Il existe quatre sortes de classes internes :

- Les classes interne statique de classe
- Les *inner-class* de classe
- Les classes internes de méthode
- Les classes anonymes de méthode

Classe internes

```
public class A {  
    public static class B {  
    }  
}
```

Classe interne statique de classe

```
public class A {  
    public void m() {  
        class B {  
        }  
    }  
}
```

Classe interne de méthode

```
public class A {  
    public class B {  
    }  
}
```

Inner-class de classe

```
public class A {  
    public void m() {  
        new Object() {  
            ...  
        };  
    }  
}
```

Classe anonyme de méthode

Classe Interne Statique

Classe interne qui n'a pas besoin d'un objet de la classe englobante pour exister.

Classe utile pour cacher des détails d'implantation

La différence avec une classe ordinaire est le scope ou elle est définie.

Visibilité de package

```
public class Coords {  
    private final Pair[] array;  
  
    public void add(int index,int x,int y) {  
        array[index]=new Coords.Pair(x,y);  
    }  
    static class Pair {  
        private final int x,y;  
        ...  
    }  
}
```

Rapport avec le C

Une classe interne est complètement différente d'une sous-structure en C

- En C, les sous-structures sont toutes instanciées en **même temps** que la structure englobante, elles font partie de la structure
- En Java, les classes internes sont définies dans la classe englobante mais sont instanciées **séparemment** (comme une classe ordinaire). Dans le cas des Inner classe, les objets de la classes interne ont une **référence sur une instance de la classe englobante.**

Espace de nommage

Le nom de la classe interne est
ClassEnglobante.ClassInterne

```
public class Coords {
    private final Pair[] array;

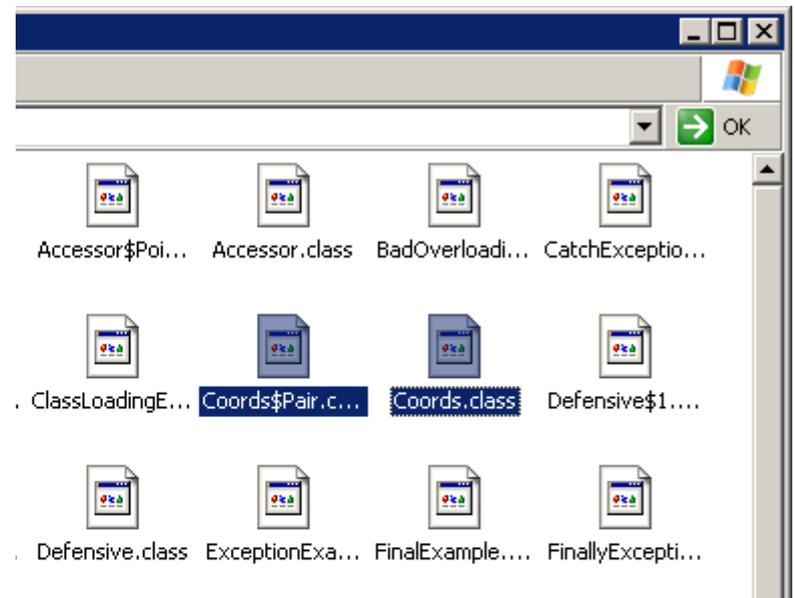
    public Pair add(int index,int x,int y) {
        return array[index]=new Pair(x,y);
        // return array[index]=new Coords.Pair(x,y);
    }
    public static class Pair {
        private final int x,y;
        ...
    }
}

public static void main(String[] args) {
    Coords coords=new Coords(10);
    Coords.Pair pair=coords.add(0,1,2);
}
```

Classe interne sur le disque

Le compilateur génère deux classes différentes :
Coords.class et Coords\$Pair.class

La machine virtuelle ne
fait pas la différence
entre une classe classique
et une classe interne



Inner Class

Une instance de Classe interne à **besoin** d'une instance de la classe englobante pour exister.

```
public class Sequence {
    private final char[] array;
    public class Sub { // sub sequence of sequence
        private final int offset;
        private final int length;

        public char charAt(int index) {
            if (index<0 || index>=length)
                throw new IllegalArgumentException(...);
            return array[offset+index];
        }
    }
}
```

L' *Inner Class* peut accéder aux champs de l'**objet**.

Instantiation d'*Inner Class*

Instantiation d'une *inner class*

```
public class Sequence {
    private final char[] array;
    public Sequence(String s) {
        array=s.toCharArray();
    }
    public class Sub {
        private final int offset;
        private final int length;
        public Sub(int offset,int length) {
            this.offset=offset;
            this.length=length;
        }
        public char charAt(int index) {
            return array[offset+index];
        }
    }
}
```

Lors de la construction, une *inner class* doit être construite **par** un objet de la classe englobante

```
Sequence s=new Sequence("toto");
Sub sub=s.new Sub(1,3);
System.out.println(sub.charAt(0));
```

Instantiation d'Inner Class (2)

A l'intérieur de la classe englobante,
il n'est pas nécessaire de préfixé par **this**

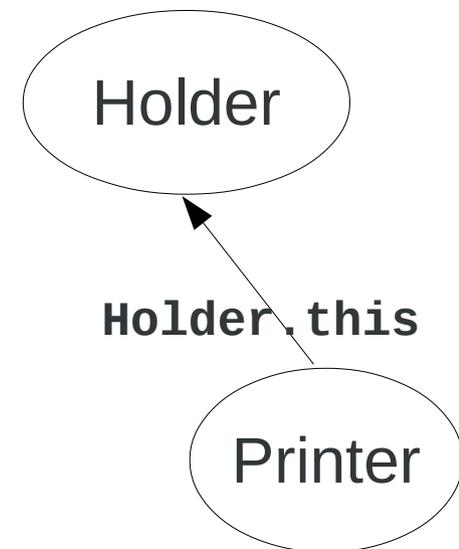
```
public class Sequence {
    private final char[] array;

    public class Sub {
        public Sub(int offset, int length) {
        }
    }
    public Sub subsequence(int offset) {
        return new Sub(
            offset, array.length - offset);
        // return this.new Sub(...)
    }
}
```

Référence sur la classe englobante

Pour obtenir une référence sur la classe englobante.

```
public class Holder {  
    ...  
    public void print() {  
        System.out.println(this);  
    }  
    public class Printer {  
        public void print() {  
            System.out.println(this);  
            System.out.println(Holder.this);  
        }  
    }  
}
```



référence: **OuterClass.this**

this et inner class

Une inner-class possède une référence sur le **this** de sa classe englobante (ici **Holder.this**)

```
Holder h1=new Holder();

System.out.println(h1);
h1.print();

Holder.Printer p=
    h1.new Printer();
p.print();
```

```
public class Holder {
    public void print() {
        System.out.println(this);
    }
    public class Printer {
        public void print() {
            System.out.println(this);
            System.out.println(Holder.this);
        }
    }
}
```

Génération par le compilateur

Le compilateur ajoute un champs vers la classe englobante et change l'appel au constructeur

```
Holder h1=new Holder();

System.out.println(h1);
h1.print();

Holder$Printer p=
    new Holder$Printer(h1);
p.print();
```

```
public class Holder {
    public void print() {
        System.out.println(this);
    }
}

public class Holder$Printer {
    private final Holder this$0;
    public Holder$Printer(Holder this$0) {
        this.this$0=this$0;
    }
    public void print() {
        System.out.println(this);
        System.out.println(this$0);
    }
}
```

Code généré

Inner-class et membre statique

Il est interdit de déclarer des membres statiques à l'intérieur d'une inner-class.

```
public class A {  
    public class B {  
        static void m() { // illégal  
        }  
    }  
}
```

```
public class A {  
    public class B {  
    }  
    static void m() { // ok  
    }  
}
```

Il est possible de déclarer ce membre dans la classe englobante.

Inner class en résumé

Une inner class est donc une façon **raccourci** d'écrire une classe ayant une **référence** sur une **classe englobante** avec en plus l'accès direct aux champs de l'objet englobant.

L'inner class a accès à l'ensemble des membres de la classe englobante (champs, méthode, autres classes internes) ce qui peut être syntaxiquement agréable.

Interface interne

Il est possible de définir des interfaces à l'intérieurs de classe ou d'interface

```
public class MyClass {  
    public interface I { // interface interne statique  
    }  
}  
public interface MyInterface {  
    public class A { // classe interne statique  
    }  
}
```

Une interface interne est toujours statique

Une classe interne d'interface est toujours statique

Visibilité

Les classes internes et inner-class de classe ont une visibilité car ce sont des membres de la classe englobante

Le constructeur par défaut de cette classe possède la même visibilité

```
public class Coords {  
    private static class Pair {  
        // le compilateur génère un constructeur privé  
    }  
}
```

Accès et visibilité

Une classe englobante à accès à **tous** les membres de sa classe interne (même privés)

Une classe interne **statique** à accès à **tous** les membres **statique** d'une classe englobante

Une inner-class à accès à **tous** les membres d'une classe englobante

Exemple d'accès

La classe englobante **Coords** à accès au champs privée de la classe interne statique **Pair** car ceux-ci sont déclarés entre l'accolade ouvrante et fermante de la classe englobante

```
public class Coords {
    private final Pair[] array;
    ...
    public int getX(int index) {
        return array[index].x; // accès à x
    }
    private static class Pair {
        private final int x,y;
        ...
    }
}
```

Accès et visibilité

Il n'y donc pas de **private** entre classe englobante et class interne

Problème car pour la VM, les deux classes sont indépendantes

Solution :

Le compilateur génère des méthodes d'accès pour éviter les violations de visibilité.

On parle d'**accessseur synthétique**.

Accesseurs générés

La méthode synthétique permet **access\$0** d'accéder au champ privé **x**.

```
public class Coords {  
    private final Pair[] array;  
    public int getX(int index) {  
        return access$0(array[index]);  
    }  
}
```

```
class Pair {  
    private final int x,y;  
    static int access$0(Pair pair) {  
        return pair.x;  
    }  
}
```

Code généré

```
$ javap -private -classpath classes Coords$Pair  
class Coords$Pair extends java.lang.Object {  
    private final int x;  
    private final int y;  
    public Coords$Pair(int, int);  
    static int access$0(Coords$Pair);  
}
```

Accesseurs sur le constructeur

Le compilateur génère aussi un accesseur pour les constructeurs
(avec une astuce pour éviter le clash)

```
public class Holder {  
    public Printer createPrinter() {  
        return this.new Printer();  
    }  
    private class Printer {  
    }  
}
```

Code généré

```
public class Holder {  
    public Printer createPrinter() {  
        return new Holder$Printer(this, null);  
    }  
}  
class Holder$Printer {  
    private final Holder this$0;  
    private Holder$Printer(Holder h) {  
        this.this$0=h;  
        super();  
    }  
    Holder$Printer(Holder h, Holder$Printer notUsed) {  
        this(h);  
    }  
}
```

Classe interne et accesseur

La VM ne connaît pas les classes internes :
donc il y a génération par le compilateur de code
supplémentaire (access\$...)

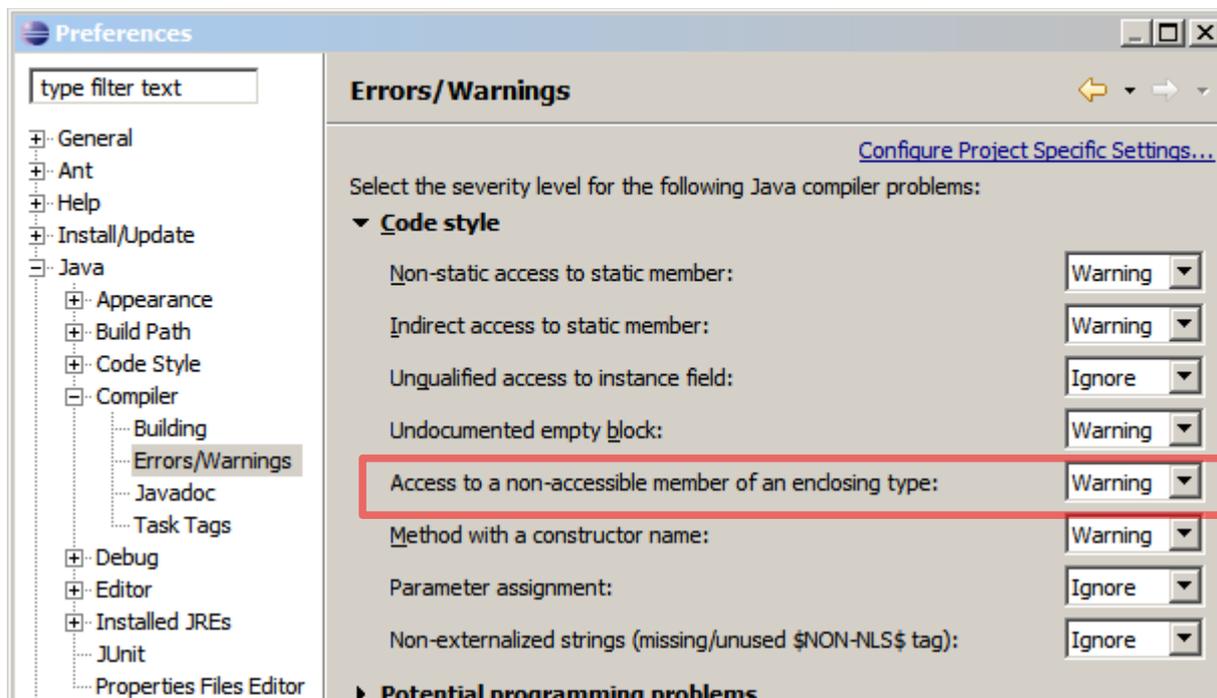
Inconvénients :

- Bytecode plus gros
- Code un peu plus lent
(en fait, pas d'impact car VM effectue inlining)

Conseil : éviter la génération d'accesseurs

Dans eclipse

On peut régler le compilateur pour qu'il emette un warning dans ces cas



Classes anonymes

Permet de **déclarer** une classe et de **créer** un objet de celle-ci en une expression.

La classe anonyme est un sous-type d'une interface ou d'une classe abstraite ou concrète

Syntaxe :

```
Type var=new Type(param1,param2...) {  
    //définition de membres  
    //(méthode/champs/classe)  
};
```

Classe Anonyme

Il est possible d'implanter une interface sans donner de nom à la classe

```
interface Filter {
    boolean accept(File file);
}
public class Test {
    public File[] subDirectories(File directory) {
        directory.listFiles(new Filter() {
            public boolean accept(File file) {
                return file.isDirectory();
            }
        });
    }
}
```

Ici, on crée un objet d'une classe implantant l'interface Filter

Classe anonyme de classe

On crée une sous-classe de BinOp anonyme

```
public abstract class BinOp {
    public BinOp(int v1,int v2) {
        this.v1=v1; this.v2=v2;
    }
    abstract int eval();
    int v1,v2;
}
public class OperationFactory {
    public static BinOp plus(int e1,int e2) {
        return new BinOp(e1,e2) {
            int eval() {
                return v1 + v2;
            }
        };
    }
}
```

Visibilité par
défaut



Variable locale et classe anonyme

La valeur des variables locales constantes est disponible dans la classe anonyme (capture)

```
interface Operation {
    int eval();
}
public class OperationFactory {
    public static Operation plus(int e1, int e2) {
        return new Operation() {
            @Override public int eval() {
                return e1 + e2;
            }
        };
    }
}
```

e1 et e2 ne sont pas modifiées
après initialisation,
ils sont “effectivement final”

Pour pouvoir capturer la valeur d'une variable, celle-ci doit être “effectivement final” (Java8) ou déclarée final

Capture de variable locale

Si une variable dont la valeur est capturé est modifié, le compilateur plante !

```
interface Operation {
    int eval();
}
public class OperationFactory {
    public static Operation plus(int e1, int e2) {
        e1 = 0; // e1 is not effectively final
        return new Operation() {
            @Override public int eval() {
                return e1+e2;
            }
        };
    }
}
```

Les variables doivent être **final** (ou **effectivement final**) car le compilateur effectue **une copie** lors de la création de la classe anonyme

Variable locale et classe anonyme

Le compilateur génère la classe correspondante

Les variables locales sont recopiés dans des champs finaux !

```
public class OperationFactory {
    public static Operation plus(int e1, int e2) {
        return new OperatorFactory$1(e1, e2);
    }
}
class OperatorFactory$1 implements Operation {
    private final int e1;
    private final int e2;
    public OperatorFactory$1(int e1, int e2) {
        this.e1 = e1;
        this.e2 = e2;
    }
    @Override public int eval() {
        return e1+e2;
    }
}
```

Relation avec les lambdas

Si on veut implanter une interface avec 1 seule méthode abstraite (functional interface) alors il vaut mieux utiliser une lambda

```
@FunctionalInterface
interface Operation {
    int eval();
}

public class OperationFactory {
    public static Operation plus(int e1, int e2) {
        return () -> e1 + e2;
    }
}
```

La capture marche de la même façon

Classe anonyme/Inner-class

Comme les inner-class, les classes anonymes ont accès à tous les champs de la classe englobante

```
public class Sequence {
    private final char[] array;
    public int length() { return array.length; }
    public int get(int index) { return array[index]; }

    public Sequence suffix(final int offset) {
        return new Sequence() {
            public int length() {
                return array.length - offset;
            }
            public int get(int index) {
                return array[offset + index];
            }
        };
    }
}
```

Classes anonymes & Limitation

Au vu de la syntaxe, il est impossible de créer une classe anonyme :

- Implémentant plusieurs interfaces
- Héritant d'une classe et implémentant une ou plusieurs interfaces

Dans ces cas, il est toujours possible de déclarer une classe interne à une méthode

Les classes internes de méthode

La classe interne est définie dans une méthode
(pas utilisée souvent !)

```
public class Util {
    public static int getPrice(final Item item) {
        class Product extends Item implements Fiscal {
            public Product() {
                this.name = item.getName();
            }
            public int price() {
                return StoreManager.getStoreByProduct(name).price(name);
            }
            private final String name;
        }
        return new Product().price();
    }
}
```

Classification

interne statique de classe

Inner-class de classe

interne de méthode

anonyme de méthode

Modificateur de visibilité	X	X		
Accès à l'objet englobant		X	X	X
Accès aux variables locales			oui	oui
Sous-type de plusieurs types	X	X	X	

Type énuméré

Un type énuméré est un type définissant un nombre fini de valeurs de ce type

Exemple:

jeux de cartes,

options de l'appel système `open()`

En Java, les valeurs d'un type énuméré sont des **objets** (et pas des constantes entières comme en C).

Enum en Java vs C

En C, un enum est un int

En Java, un enum est un type

La syntaxe de Java est retro-compatible avec celle du C :

```
enum OpenOption {READ, WRITE, READ_WRITE}
```

En Java, un enum possède en plus, des constructeurs, méthodes, champs, classes internes, etc.

Valeur de l'énumération

Les valeurs de l'énumération sont des champs public final static.

Ils ont le type du type énuméré

```
public enum Option {
    l, a, v
}
...
public static void apply(Option... options) {
    for(Option option:options) {
        if (option == Option.l) {
            // long option
        } else {
            if (option == Option.a) {
                // all option
            } else { //Option.v
                // verbose option
            }
        }
    }
}
```

Switch

On peut faire un switch sur les valeurs d'une énumération

Attention, il ne faut pas qualifier (pas de '.') les valeurs dans les cases

```
public static void apply(Option... options) {
    for(Option option:options) {
        switch(option) {
            case l:           // case Option.l ne compile pas !!
                // long option
                break;
            case a:
                // all option
                Break;
            case v:
                // verbose option
        }
    }
}
```

Un type énuméré est un object

Tous les types énumérés hérite de `java.lang.Enum`

Ils possèdent donc deux méthodes :

- `int ordinal()` qui renvoie leur position dans l'ordre de déclaration (à partir de zéro)
- `String name()` qui renvoie leur nom

```
public enum Options {  
    l, a, v  
}  
...  
public static void main(String[] args) {  
    assert Options.l.ordinal() == 0;  
    assert Options.a.name().equals("a");  
}
```

Membres d'un type énuméré

Un type énuméré possède des membres :
méthodes, champs, classes internes

Les valeurs et les membres sont séparés par un ':'

Avoir des enums mutable est pas une bonne idée

```
public enum Age {  
    jeune, mure, agé, vieux, cadavérique;  
  
    private int année;        // argh un accent  
  
    public static void main(String[] args) {  
        Age.jeune.année = 20;  
        Age.jeune.année = 1200; // argh mutable  
        ...  
    }  
}
```

Enumération et champs static

Les constructeurs ne peuvent accéder aux champs statiques (problème d'ordre d'initialisation)

```
public enum MyColor {
    RED, GREEN, BLUE;
    static final HashMap<String, MyColor> map=
        new HashMap<String, Color>();

    private MyColor() {
        map.put(name(), this); // si c'était possible
                               // NullPointerException
    }
}
```

Utiliser un bloc static (qui sera exécuté après l'initialisation des champs de l'enum)

Enumération et champs static

Les constructeurs ne peuvent accéder aux champs statiques (problème d'ordre d'initialisation)

```
public enum MyColor {  
    RED, GREEN, BLUE;  
    static final HashMap<String, MyColor> map=  
        new HashMap<String, Color>();  
  
    MyColor() {  
        map.put(name(), this); // si c'était possible  
                                // NullPointerException  
    }  
}
```

Utiliser un bloc static (qui sera exécuté après l'initialisation des champs de l'enum)

Enumération et champs static

Les constructeurs ne peuvent accéder aux champs statiques (problème d'ordre d'initialisation)

```
public enum MyColor {  
    RED, GREEN, BLUE;  
    static final HashMap<String, MyColor> map;  
    static { map = new HashMap<String, Color>(); }  
  
    MyColor() {  
        map.put(name(), this); // si c'était possible  
                               // NullPointerException  
    }  
}
```

Utiliser un bloc static (qui sera exécuté après l'initialisation des champs de l'enum)

Enumération et champs static

Utiliser un bloc static (qui sera exécuté après l'initialisation des champs de l'enum) initialisés en premier

```
public enum MyColor {
    RED, GREEN, BLUE;
    static final HashMap<String,MyColor> map;
    // le code suivant est inutile comme la méthode valueOf
    // existe (mais pour le fun)

    static {
        HashMap<String,Color> map =
            new HashMap<String,Color>();
        for(MyColor m:values()) {
            map.put(m.getName(),m);
        }
        MyColor.map = map;
    }
}
```

Constructeur d'un type énuméré

Un enum peut avoir des constructeurs

Ils doivent être private ou de package

Les parenthèses ne sont pas obligatoire pour appeler le constructeur sans paramètre

```
public enum Age {
    jeune, autreJeune(), mure(40), agé(60), vieux(80), cadavérique(999);
    // les valeurs de ordinal restent 0,1,2,3,4,5
    private final int année;

    private Age() {
        this(20);
    }
    private Age(int année) {
        this.année = année;
    }
}
```

Méthodes générées

Tous les types énumérés possèdent aussi deux méthodes **statiques** générées par le compilateur

- *TypeEnuméré*[] **values()**
renvoie une copie d'un tableau contenant toutes les valeurs de l'énumération dans l'ordre de déclaration
- *TypeEnuméré* **valueOf**(String name)
renvoie la valeur de l'énumération dont le nom est name ou lève une exception `IllegalArgumentException`.

Exemple d'utilisation

```
import java.util.*;
import static Option.*; // import static

public class Main {
    public static void main(String[] args) {
        Set<Option> set = EnumSet.noneOf(Option.class);

        for(Option opt:Option.values()) {
            if (opt==a) { // ok, avec import static
                Collections.addAll(set, l, v);
            }
            else
                set.add(opt);
        }

        System.out.println(set);
    }
}
```

Attention **values()** renvoie la copie d'un tableau à chaque appel

Révision - import static

Permet d'éviter :

```
double r = Math.cos(Math.PI * theta);
```

remplacé par

```
import static java.lang.Math.PI;  
import static java.lang.Math.cos;
```

...

```
double r = cos(PI * theta);
```

à utiliser avec parcimonie, uniquement si cela améliore la lisibilité du code

Exemple d'énumération

```
public enum Option {
    l, a, v;

    public static Option getOption(String s) {
        if (s.length() != 2 || s.charAt(0) != '-')
            return null;
        return Option.valueOf(s.substring(1));
    }

    public static void main(String[] args) {
        for (String s : args) {
            Option option = getOption(s);
            if (option != null)
                System.out.println(option.name() + ':' + option.ordinal());
        }
    }
}
//java Option -a -v -l
// a:1 v:2 l:0
```

Classe interne et énumération

Une énumération ne peut être définie **que dans un contexte statique** donc pas possible dans une *inner class* ou dans une méthode

```
public class Myclass {
    class Inner {
        enum Arg { // interdit
            toto
        }
    }
    void f() {
        enum Arg2 { // interdit
            toto
        }
    }
}
```

Enumération et java.lang.Enum

Les enums héritent de **java.lang.Enum**

La class Enum est paramétré

Enum<E extends Enum<E>>, E est le type de l'énumération

```
public enum Option {  
    l, a, v;  
  
    public static void main(String[] args) {  
        Option opt1=Option.l;  
        Enum<Option> opt2=Option.a;  
    }  
}
```

Enumération et héritage

Le compilateur garantit que seules les enum hérite de java.lang.Enum.

- Une classe ne peut hériter de Enum
- Une classe ne peut hériter d'une énumération
- Une énumération ne peut hériter d'une classe

```
public class A extends Enum { } // erreur
public class A extends Option { } // erreur
public enum Option extends A { } // erreur
public enum Option2 extends Option { } // erreur
```

Valeur d'enum anonyme

Il est possible d'ajouter du code spécifique à une valeur particulière d'un enum

La syntaxe entre les { } est la même que pour les classes anonymes

```
public enum Option {  
    l,  
    a,  
    v {  
        @Override  
        public String toString() {  
            return "hello v";  
        }  
    };  
  
    public static void main(String[] args) {  
        System.out.println(Option.v); // hello v  
    }  
}
```

Valeur d'enum anonyme

Le compilateur génère une classe anonyme ! Pour chaque comportement différent

Noter que le constructeur **n'est pas privé** !

Le code précédent correspond au pseudo-code-généré suivant:

```
public class Option extends java.lang.Enum {
    Option(int ordinal, String name) {
        super(ordinal, name);
    }

    public static final Option l = new Option(0, "l");
    public static final Option a = new Option(1, "a");
    public static final Option v = new Option(2, "v") {
        @Override
        public String toString() {
            return "hello v";
        }
    };
};
```

Enumération et Initialisation

Il n'est pas possible de déclarer un constructeur pour un champs spécifique d'une énumération

```
public enum MyEnum {  
    max {  
        max() { // erreur, considérée comme  
                // une méthode pas un constructeur  
        }  
        int f() { // ok  
            return 3;  
        }  
    },  
    min  
}
```

Les contraintes sont les mêmes que pour les classes anonymes

Methode non atteignable

La valeur d'une enum anonyme est typé par le type énuméré

```
public enum Option {  
    l,  
    a,  
    v {  
        void f() {  
            // inatteignable  
        }  
    };  
}
```

dans l'exemple Option.v est de type Option, donc on ne peut pas appeler f().

Énumération abstraite

Une énumération peut être abstraite
(mieux que le switch)

```
public enum Option {
    l {
        public void performs() {
            System.out.println("long");
        }
    }, a {
        public void performs() {
            System.out.println("all");
        }
    }, v {
        public void performs() {
            System.out.println("verbose");
        }
    };

    public abstract void performs();
}

public static void performs(Set<Option> set) {
    for(Option option:set)
        option.performs();
}
```

L'énumération ne doit pas
être déclarée **abstract** !!

Énumération et interface

Une énumération peut donc implanter une ou plusieurs interfaces

```
public interface Performer {
    public void performs();
}

public enum Option implements Performer {
    l {
        public void performs() {
            System.out.println("long");
        }
    },
    a {
        public void performs() {
            System.out.println("all");
        }
    };
}
```

Et avec des lambdas

L'exemple précédent peut aussi utiliser la délégation

```
public enum Option {
    l(() -> System.out.println("long")),
    a(() -> System.out.println("all"))
;

    interface Performer {
        public void performs();
    }

    private Performer performer
    private Option(Performer performer) {
        this.performer = performer;
    }
    public void performs() {
        performer.performers();
    }
}
```

Énumération résumé

Les énumération en Java ne sont pas des entiers mais des objets qui possède une méthode **ordinal()**

On peut faire un switch dessus

On peut ajouter des champs, des méthodes, des classes, etc, redéfinir des méthodes avec les enums abstraits

Mais un enum représente un nombre fini de valeurs donc **n'est pas un type extensible.**