

Introduction à la Programmation Orientée Objet

9 - Les classes internes et anonymes

Valentin Honoré

`valentin.honore@ensiie.fr`

FISA 1A

- La syntaxe utilisée pour l'héritage de classe/mise en œuvre d'interface est trop verbeuse pour des codes simples

```
interface Bird { void fly(); }
```

```
class MyBird implements Bird {  
    void fly() {  
        System.out.println("fly!");  
    }  
}
```

```
class Test {  
    void f() {  
        Bird bird = new MyBird();  
    }  
}
```

Nécessite une nouvelle classe,
donc un fichier,
le tout pour allouer
une **unique instance** et
peu de lignes de code

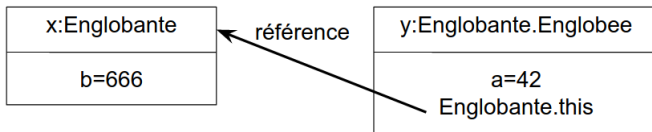
- 1 Classes internes
- 2 Classe interne de méthode
- 3 Classes anonymes
- 4 Les fonctions/expressions lambda

- ▶ Une classe interne est une classe définie dans une autre classe
 - permet de coupler fortement deux classes (la classe interne a accès aux champs de la classe externe)

```
class Englobante {  
    class Englobee {  
        int a;  
        void f() { a = 42; b = 666; }  
    }  
    private int b;  
    private Englobee englobee;  
}
```

Mise en œuvre d'une classe interne

- ▶ Une classe interne possède un champ ajouté automatiquement permettant d'accéder à la classe externe



```
void f() { a = 42; b = 666; }
```



```
void f() { this.a = 42; Englobante.this.b = 666; }
```

- ▶ À partir de la classe externe, allocation de façon normale

```
class Englobante {  
    class Englobee { ... }  
    Englobante() {  
        englobee = new Englobee();  
    }  
}
```

- ▶ À partir de l'extérieur de la classe englobante :

```
Englobante x = new Englobante();  
Englobante.Englobee y = x.new Englobee();  
x.englobee = y; /* à faire à la main */
```

- ▶ Les champs de la classe englobée sont toujours accessibles à partir de la classe englobante et vice-versa
- ▶ Composition des opérateurs de visibilité pour l'extérieur

```
public class Englobante { /* accessible partout */  
    class Englobee { /* accessible du package */  
        private int x; /* accessible de Englobante */  
    }  
    private int b; /* accessible de Englobee */  
    private Englobee englobee; /* idem */  
}
```

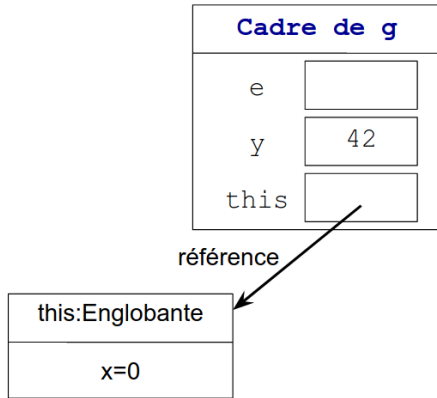
- 1 Classes internes
- 2 Classe interne de méthode
- 3 Classes anonymes
- 4 Les fonctions/expressions lambda

Classe interne de méthode

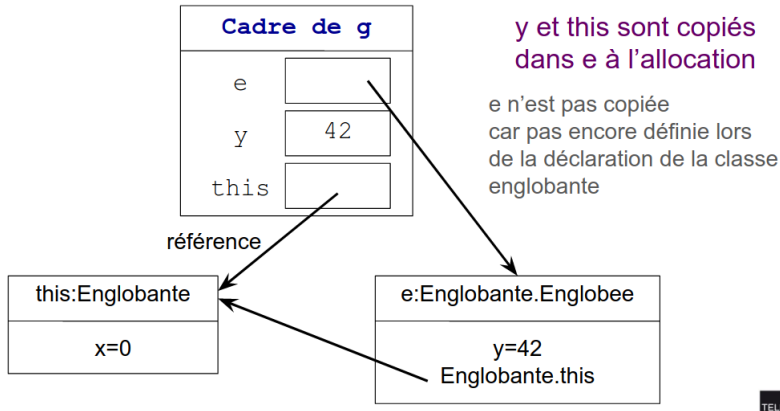
- ▶ Classe interne de méthode = classe définie dans une méthode
- ▶ La classe interne peut aussi accéder aux variables locales de la méthode (si accédées en lecture après la déclaration)

```
class Englobante {  
    int x;  
    void f() {  
        int y = 42; /* y en lecture seule dans Englobee */  
        class Englobee { void g() { x = y; }  
        };  
        Englobee e = new Englobee();  
        e.g();  
    }  
}
```

- Une instance d'une classe interne de méthode possède une copie de chaque variable de la méthode englobante



- Une instance d'une classe interne de méthode possède une copie de chaque variable de la méthode englobante



- 1 Classes internes
- 2 Classe interne de méthode
- 3 Classes anonymes**
- 4 Les fonctions/expressions lambda

- ▶ But : simplifier la syntaxe utilisée pour l'héritage de classe ou la mise en œuvre d'interface pour les codes simples
 - Allocation d'une **unique instance** de la classe dans le code
 - Peu de méthodes et de lignes de code dans la classe
- ▶ Principes :
 - Omettre le nom de la classe
 - Donner le code de la mise en œuvre au moment de l'allocation

Classes anonymes (1/2)

- ▶ Classe anonyme = classe interne de méthode sans nom

```
void f() { /* en version nommée */  
    class AvecNom extends Bird {  
        void fly() { System.out.println("fly!"); }  
    };  
    Bird bird = new AvecNom();  
}
```

↔

```
void f() { /* en version anonyme */  
    Bird bird = new Bird() {  
        void fly() { System.out.println("fly!"); }  
    };  
}
```

- ▶ Une classe anonyme est une classe interne de méthode
 - Si méthode d'instance, peut accéder aux champs de l'instance
 - Peut accéder aux variables locales de la méthode à condition que ces variables ne soient accédées qu'en lecture à partir de la déclaration de la classe anonyme

```
class Test {  
    int x; /* lecture/écriture à partir de classe  
           anonyme */  
    void f() {  
        int a = 42; /* lecture à partir de classe anonyme  
                     */  
        Bird bird = new Bird() { void fly() { x = a; } };  
        bird.fly();  
    }  
}
```

Les classes anonymes par l'exemple

```
interface Bird { void fly(); }
```

Définit une nouvelle
classe qui hérite de `Bird`
et qui n'a pas de nom

```
class MyBird {  
    void fly() {  
        System.out...;  
    } }
```

```
class Test {  
    void f() {  
        Bird bird = new Bird() {  
            void fly() {  
                System.out...  
            }  
        }  
    } }
```

Mise en œuvre
au moment de l'allocation

```
class Test {  
    void f() {  
        Bird bird = new MyBird();  
    } }
```


Pour aller plus loin : classe interne statique

- ▶ Si la classe interne est marquée `static`
 - ☐ pas liée à une instance de la classe englobante
 - ☐ pas d'accès aux champs d'instance d'une instance englobante
 - ☐ allocation sans instance d'une classe Englobante

```
class A {  
    static class B {  
        int a;  
        void f() {  
            a = 42; c = 666; /* peut pas accéder b */  
        }  
    }  
    private int b;  
    private static int c;  
} /* allocation avec A.B x = new A.B(); */
```

- 1 Classes internes
- 2 Classe interne de méthode
- 3 Classes anonymes
- 4 Les fonctions/expressions lambda**

- ▶ Pour les cas les plus simple, le code reste verbeux, même avec les classes anonymes
- ▶ But des fonctions lambda
 - ❑ écrire du code plus concis, plus rapide à écrire/relire
 - ❑ disponible depuis Java 8, introduction de la programmation fonctionnelle
- ▶ Expression lambda = expression anonyme
 - ❑ définition sans déclaration explicite du type de retour
 - ❑ ni de modificateurs d'accès ni de nom
 - ❑ permet de définir une méthode directement à l'endroit où elle est utilisée.
- ▶ Raccourci syntaxique qui simplifie l'écriture de traitements passés en paramètre
 - ❑ adaptée lorsque le traitement n'est utile qu'une seule fois
 - ❑ → évite d'avoir à écrire une méthode dans une classe.
 - ❑ une interface qui ne possède qu'une seule méthode abstraite : **interface fonctionnelle**

- ▶ Evaluation par le compilateur :
 - ☐ inférence du type vers l'interface fonctionnelle
 - ☐ récupération sur les paramètres, le type de retour, les exceptions etc
- ▶ Bilan :
 - ☐ Code plus compact
 - ☐ Code plus lisible
 - ☐ Abstraire un traitement pour le passer à d'autres traitements

Syntaxe d'une expression lambda (1/2)

► Trois parties

- ☐ un ensemble de paramètres (de 0 à n)
- ☐ l'opérateur $\rightarrow$
- ☐ le corps de la fonction

► Deux formes possibles

- ☐ `(paramètres)  $\rightarrow$  expression;`
- ☐ `(paramètres)  $\rightarrow$  { traitements; }`

► Quelques règles syntaxiques

- ☐ paramètres entourés de parenthèses et séparés par des virgules
- ☐ `()` si pas de paramètres
- ☐ l'opérateur $\rightarrow$ permet de séparer les paramètres des traitements qui les utiliseront

Syntaxe d'une expression lambda (2/2)

- ▶ Si le corps est simplement une expression, celle-ci est évaluée et le résultat de cette évaluation est renvoyé (s'il y en a un)

```
BiFunction<Integer, Integer, Long> addition = (p1,  
    p2) -> (long) p1 + p2;
```

- ▶ Jamais nécessaire de préciser explicitement le type de retour
 - le compilateur doit être en mesure de le déterminer selon le contexte
 - si ce n'est pas le cas → erreur de compilation

- ▶ Possible d'utiliser `return` ou lever une exception.

```
BiFunction<Integer, Integer, Long> addition = (p1,  
    p2) -> {return ((long) p1 + p2); };  
long r = addition.apply(1,2)
```

Exemples d'expressions lambda

```
() -> 123
```

```
() -> { return 123 };
```

```
(x, y) -> x + y
```

```
(int x, int y) -> x + y
```

```
c -> { int s = c.size(); c.clear(); return s; }
```

```
() -> { System.out.println("Hello World"); };
```

```
() -> { for (int i = 0; i < 10; i++) traiter(); }
```

```
(val1, val2) -> { return val1 >= val2; }
```

```
(val1, val2) -> val1 >= val2;
```

```
(int valeur) -> new Integer(valeur)
```

```
(String s) -> new Integer(s) /* exemples avec constructeur */
```

Exemples d'expressions lambda

```
() -> 123
```

```
() -> { return 123 };
```

```
(x, y) -> x + y
```

```
(int x, int y) -> x + y
```

```
c -> { int s = c.size(); c.clear(); return s; }
```

```
() -> { System.out.println("Hello World"); };
```

```
() -> { for (int i = 0; i < 10; i++) traiter(); }
```

```
(val1, val2) -> { return val1 >= val2; }
```

```
(val1, val2) -> val1 >= val2;
```

```
(int valeur) -> new Integer(valeur)
```

```
(String s) -> new Integer(s) /* exemples avec constructeur */
```


Exemples d'expressions lambda

```
() -> 123
```

```
() -> { return 123 };
```

```
(x, y) -> x + y
```

```
(int x, int y) -> x + y
```

```
c -> { int s = c.size(); c.clear(); return s; }
```

```
() -> { System.out.println("Hello World"); };
```

```
() -> { for (int i = 0; i < 10; i++) traiter(); }
```

```
(val1, val2) -> { return val1 >= val2; }
```

```
(val1, val2) -> val1 >= val2;
```

```
(int valeur) -> new Integer(valeur)
```

```
(String s) -> new Integer(s) /* exemples avec constructeur */
```

Exemples d'expressions lambda

```
() -> 123
```

```
() -> { return 123 };
```

```
(x, y) -> x + y
```

```
(int x, int y) -> x + y
```

```
c -> { int s = c.size(); c.clear(); return s; }
```

```
() -> { System.out.println("Hello World"); };
```

```
() -> { for (int i = 0; i < 10; i++) traiter(); }
```

```
(val1, val2) -> { return val1 >= val2; }
```

```
(val1, val2) -> val1 >= val2;
```

```
(int valeur) -> new Integer(valeur)
```

```
(String s) -> new Integer(s) /* exemples avec constructeur */
```

Exemples d'expressions lambda

```
() -> 123
```

```
() -> { return 123 };
```

```
(x, y) -> x + y
```

```
(int x, int y) -> x + y
```

```
c -> { int s = c.size(); c.clear(); return s; }
```

```
() -> { System.out.println("Hello World"); };
```

```
() -> { for (int i = 0; i < 10; i++) traiter(); }
```

```
(val1, val2) -> { return val1 >= val2; }
```

```
(val1, val2) -> val1 >= val2;
```

```
(int valeur) -> new Integer(valeur)
```

```
(String s) -> new Integer(s) /* exemples avec constructeur */
```

Exemples d'expressions lambda

```
() -> 123
```

```
() -> { return 123 };
```

```
(x, y) -> x + y
```

```
(int x, int y) -> x + y
```

```
c -> { int s = c.size(); c.clear(); return s; }
```

```
() -> { System.out.println("Hello World"); };
```

```
() -> { for (int i = 0; i < 10; i++) traiter(); }
```

```
(val1, val2) -> { return val1 >= val2; }
```

```
(val1, val2) -> val1 >= val2;
```

```
(int valeur) -> new Integer(valeur)
```

```
(String s) -> new Integer(s) /* exemples avec constructeur */
```

Exemples d'expressions lambda

`() -> 123`

`() -> { return 123 };`

`(x, y) -> x + y`

`(int x, int y) -> x + y`

`c -> { int s = c.size(); c.clear(); return s; }`

`() -> { System.out.println("Hello World"); };`

`() -> { for (int i = 0; i < 10; i++) traiter(); }`

`(val1, val2) -> { return val1 >= val2; }`

`(val1, val2) -> val1 >= val2;`

`(int valeur) -> new Integer(valeur)`

`(String s) -> new Integer(s) /* exemples avec constructeur */`

Encore d'autres exemples

- ▶ Depuis Java 8, méthodes = fonctions de premier ordre acceptant d'autres fonctions en paramètre (donc les lambdas!)
- ▶ Exemple : méthode sort de l'interface [List](#)
 - besoin d'appliquer une comparaison sur les éléments deux à deux (relation d'ordre)
 - définition par une lambda

```
List<Integer> liste = new ArrayList<>();
liste.add(1);
liste.add(2);
liste.add(3);
liste.add(4);

// trie la liste en plaçant en premier les nombres
// pairs
liste.sort((e1, e2) -> (e1 % 2) - (e2 % 2));

// [2, 4, 1, 3]
System.out.println(liste);
```

- ▶ Interface fonctionnelle = interface avec une unique méthode
- ▶ Remplacement de la mise en œuvre de l'interface fonctionnelle par une expression lambda

```
interface Bird { void fly(); }
```

```
Bird bird = () -> { System.out.println("fly!"); };
```

⇔

```
Bird bird = new Bird() {  
    void fly() {  
        System.out.println("fly!");  
    }  
};
```

On ne garde que la
partie « intéressante »
de la déclaration

Retour sur les collections : la méthode "pour chaque"

- ▶ L'interface `Iterable` (parent de l'interface `Collection`) a une méthode `forEach()` dans Java 8.

- ☐ fournit un autre moyen, plus fonctionnel, d'itérer sur les collections

- ☐ `void forEach(Consumer action)`

- ▶ Code de l'interface fonctionnelle `Consumer`

```
@FunctionalInterface
public interface Consumer<T> {
    void accept(T t);
}
```


Un exemple complet

```
import java.util.ArrayList;
import java.util.List;
import java.util.function.Consumer;

public class Test {
    public static void main(String[] args){
        List epees = new ArrayList();
        epees.add("Glamdring"); epees.add("Enduril");
        epees.add("Narsil");

        Consumer afficherEpee = new Consumer() {
            public void accept(String epee) {
                System.out.println(epee);
            };
        };

        epees.forEach(afficherEpee);
    }
}
```

Le même avec les lambda !

```
import java.util.ArrayList;
import java.util.List;

public class Test {

    public static void main(String[] args){

        List epees = new ArrayList();
        epees.add("Glamdring");
        epees.add("Enduril");
        epees.add("Narsil");

        //fonction lambda qui remplace le Consumer!
        epees.forEach(epee -> System.out.println(epee));

    }
}
```

- ▶ Accès aux champs de la classe anonyme, mais aussi de la classe englobante et aux variables de la méthode englobante
- ▶ Accès en lecture seule aux variables de la méthode

```
class Nid {  
    int x; /* accès en lecture/écriture */  
    void f() {  
        int y; /* en lecture seule */  
        Bird bird = new Bird() {  
            int z; /* accès en lecture/écriture */  
            void fly() { z = x + y; }  
        };  
    }  
}
```

► Classe anonyme

- Simplifie l'héritage de classe et la mise en œuvre d'interface dans les cas simples

- `Bird bird = new Bird() { void fly() { ... } };`

► Expressions lambda :

- Simplifie encore le code pour les interfaces fonctionnelles
- Interface fonctionnelle = interface avec une unique méthode

- `Bird bird = () -> { System.out.println("fly!"); }`