MOdélisation OBjet

4 - Les classes génériques et anonymes

Valentin Honoré

valentin.honore@ensiie.fr

FISA 1A

Héritage et réutilisation de code (rappel)

- On veut souvent réutiliser une structure de données
  - Exemple typique : le tableau extensible des premiers TPs

```
class LocationSet{
    private Location[] locations = new Location[2];
    private int nbLocations=0;
    /* Ajouter une nouvelle location */
    void add (Location l) {
        if (nbLocations == locations.length-1) {
            Location[] newLocations = new ... ;
            for (int i=0; i<locations.length; i++) {
                newLocations[i] = locations[i];
            }
            locations = newLocations;
        }
        locations[nbLocations] = location;
        nbLocations++;
    }}
```

Héritage et réutilisation de code (rappel)

- Pour rendre le code réutilisable dans d'autres contextes, il suffit de :
  1. Remplacer `Location` par `ArrayList`
  2. Remplacer `Location` par `Object` puisque `Location` hérite de `Object`

```
class ArrayList {
    Object[] content; /* tableau */
    int top; /* occupation tableau */
    void add(Object m) {
        if(top == content.length) {
            Object[] tmp = new Object[content.length * 2];
            for(int i=0; i<content.length; i++) {
                tmp[i] = content[i];
                content = tmp;
            }
            content[top++] = m;
        }
    }
}
```

Plan du cours

Pourquoi la programmation générique (1/2)

- Problème : nécessite des sous-typage explicites à l'utilisation
  - `ArrayList` renvoie des `Object` (car type pas connu avant exécution)
  - ce qui force un sous-typage explicite à l'utilisation

```
class ArrayList {
    ...
    Object get(int i) { return content[i]; }
}

class LocationSet {
    ArrayList locations;
    void display(int i) {
        /* Force un typage explicite ici */
        Location m = (Location)
            locations.get(i);
        m.display();
    }
}
```

Pourquoi la programmation générique (2/2)

- Ces sous-typages explicites ne sont pas idéals pour le développeur !
  - Les sous-typages explicites rendent le code confus et difficile à lire
  - Et ne permettent pas au compilateur de s'assurer que `ArrayList` ne contient que des `Location`

$\rightarrow$ risque d'erreurs difficile à détecter si le programme stocke par erreur autre chose qu'une `Location` dans le `ArrayList`

La classe générique

- Classe générique = **classe paramétrée par une autre classe**

```
class LocationSet {
    /* ArrayList stocke des Location et plus des Object */
    ArrayList<Location> locations;
    void display(int i) {
    /* plus besoin de sous-typer le résultat */
        Location m = locations.get(i);
    m.display();
    }
}
```

- Avantages
  - Détection des incohérences de types à la compilation (impossible de stocker autre chose qu'une `Location` dans `locations`)
  - Code plus facile à lire car plus de sous-typages explicites

Utilisation simple (1/2)

```
// la classe Bag est paramétrée par E
// E inconnu ici, il sera connu à la déclaration
class Bag<E> {
    private E[] elements;
    public Bag() { elements = new E[42]; }
    public E get(int i) { return element[i]; }
}
Bag<Potion> b; // déclare un Bag avec E valant Potion
b = new Bag<>(); // alloue un Bag (le paramètre Potion est
   automatiquement déduit à partir du type de b)

Bag<Potion> b = new Bag<>(); // équivalent
```

- b est un sac à potions, on peut faire :

```
Potion p = b.get(0);
```

Utilisation simple (2/2)

```
class Tree<E, F> { /* paramétré par E, F */
    private Tree<E, F> left;
    private Tree<E, F> right;
    private E key;
    private F value;
    ...
}
```

`Tree<String, int>` déclare un `Tree` avec E valant `String` et F valant `int`

Utilisation avancée

- Utilisation de extends s'il faut que le type paramètre hérite d'une classe précise

```
class Bag<E extends Item> {
    private E[] elmts;
    public int getWeight() {...
    for(...)
        tot += elmts[i].getWeight(); ...}
}
class Item {
    abstract public int getWeight();
}
```

E doit hériter de Item car E doit posséder la méthode getWeigth

Attention

Si un type paramètre doit mettre en œuvre une **interface**, on utilise quand même extends

Tableau de classes génériques

- Attention : pas d'allocation d'un tableau de classes génériques avec "<>"

```
Truc<String>[] t = new Truc<>[10]; /*
    interdit */
```

- Allocation du tableau "comme si "Truc" n'était pas générique

```
Truc<String>[] t = new Truc[10]; /*
    autorisé */
```

- En revanche, allocation des éléments de façon normale

```
t[0] = new Truc<>("Hello");
```

Notion avancée : déduction de type

- Le compilateur Java déduit automatiquement que le paramètre de new Bag<>() est Potion via le type de b dans

```
Bag<Potion> b = new Bag<>();
```

- Mais le compilateur n'est pas toujours capable de déduire le type

```
class Potion { void drink() { ... } }
class Bag<E> { E e; E get() { return e; } }
(new Bag<>()).get().drink(); // Déduction
    impossible
```

- Dans ce cas, il faut explicitement donner le paramètre lors de l'allocation

```
(new Bag<Potion>()).get().drink();
```

Notions clés

- Classe générique = classe paramétrée par d'autres types

```
class Truc<X, Y, Z extends Bidule>
```

- Déclaration en spécifiant les paramètres

```
Truc<A, B, C> truc;
Allocation en ajoutant "<>" après le mot
    clé new
truc = new Truc<>();
```

- Pas de "<>" à côté du new pour allouer un tableau de classes génériques

```
Truc<A, B, C>[] trucs = new Truc[10];
```

Plan du cours

Le problème

- 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

Plan du cours

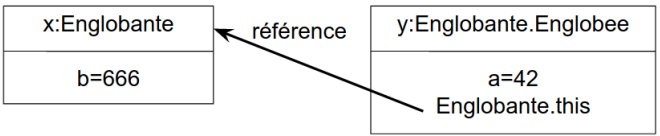
Classes internes

- 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; }
```

Allocation d'une classe interne

- 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 */
```

Classe interne et visibilité

- Les champs de la classe englobée sont toujours accessibles à partir de la classe englobante et vice-versa

- Composition des opérateurs de visibilités 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 */
}
```

Plan du cours

Classe interne de méthode

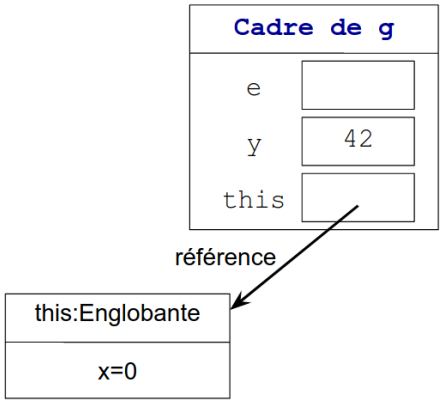
- Classe interne de méthode = classe définie dans une méthode

- 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();
  }
}
```

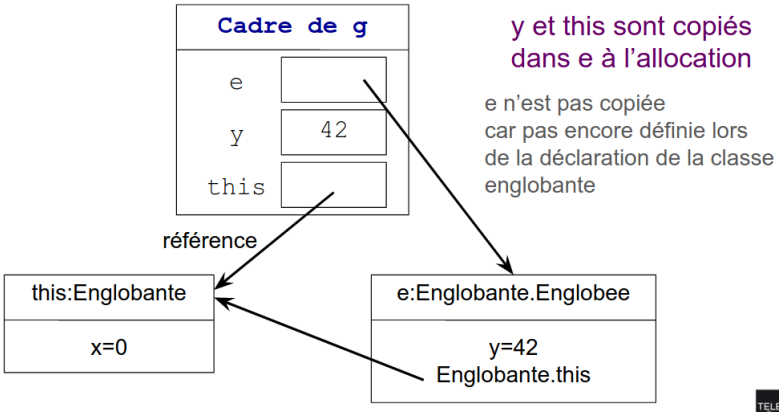
Mise en œuvre

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



Mise en œuvre

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



Plan du cours

Les classes anonymes

- 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();
}
```

$\leftrightarrow$

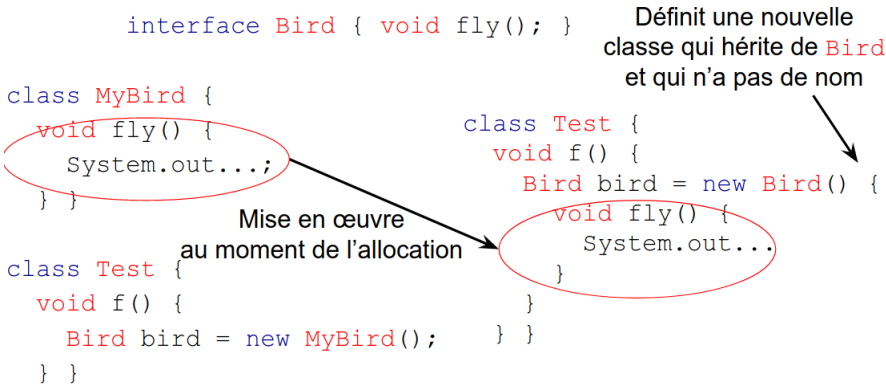
```
void f() { /* en version anonyme */
  Bird bird = new Bird() {
    void fly() { System.out.println("fly!"); }
  };
}
```

Classes anonymes (2/2)

- 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



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 acceder b */
    }
  }
  private int b;
  private static int c;
} /* allocation avec A.B x = new A.B(); */
```

Plan du cours

Les fonctions lambda (1/2)

- 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
  - $\rightarrow$ évite d'avoir à écrire une méthode dans une classe.
  - une interface qui ne possède qu'une seule méthode abstraite : **interface fonctionnelle**

Les expressions lambda (2/2)

- 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 `->`
  - le corps de la fonction

- Deux formes possibles
  - `(paramètres) -> expression;`
  - `(paramètres) -> { traitements; }`

- Quelques règles syntaxiques
  - paramètres entourés de parenthèses et séparés par des virgules
  - () si pas de paramètres
  - l'opérateur -> 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 placant en premier les nombres
   pairs
liste.sort((e1, e2) -> (e1 % 2) - (e2 % 2));

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

Expressions lambda & Interface fonctionnelle

- 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));

    }
}
```

La portée des variables

- 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; }
    };
  }
}
```

A retenir

- 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!"); }`