Programmation Objet
Java–Expressions Lambda

Michail Lampis
michail.lampis@dauphine.fr

Classes Internes

- A-t-on le droit de mettre une classe dans une autre classe ?
  - Oui ! Et on a plusieurs types des classes définies dans d’autres classes
- Classes Membres → Définies comme attributs d’une classe
  - Statiques
  - Non-statiques
- Classes Locales → Définies dans un bloc de code
  - Classes Anonymes
  - Expressions Lambda

Classes Membres

- Considérez l'exemple d'un programme de traitement des images

```
class MyImage{
        private int maxValue;
        class Pixel {
                private int red,green,blue;
        }
        int width, height;
        Pixel [][] data;
}
```

- Quel intérêt ?
  - Bonne organisation, encapsulation

Classes Membres

- Une classe membre (non-statique) a besoin d'un objet englobant (comme tous les membres non-statiques)
- Elle peut l'utiliser avec **ClasseEnglobante.this** (ou on n'écrit rien s'il n'y a pas d'ambiguïté)

```
class MyImage{

        private int maxValue;

        class Pixel {

                private int red,green,blue;

                private Pixel(int r, int g, int b){

                  red = r < maxValue ? r : maxValue;

                  green = g < maxValue ? g : maxValue;

                  blue = b < maxValue ? b : MyImage.this.maxValue;

        }
```

Classes Membres

- On peut utiliser la classe membre dans la classe englobante
  - Pour instancier, il faut un objet de la classe englobante
  - On écrite **this.new Pixel(..)** (ou simplement **new Pixel(..)**)

```
class MyImage{
        private int maxValue;
        class Pixel {...  private Pixel(int r, int g, int b){ ..} }
        Pixel [][] data;
        public MyImage() {
                data = new Pixel[1][1]; maxValue = 2 ;
                data[0][0] = this.new Pixel(1,2,3);
        }
}
```

Classes Membres Static

- Une classe membre static est une classe dont un objet peut être instancié sans un objet de la classe englobant
- Ex :

```
class myStack{

        private static class Node{  Node next;  Object data;   }

        Node head;

        void push(Object o) {Node n = new Node(); n.next=head;
n.data=o; }

        Object pop() { Object o = head.data; head=head.next ;
return o ; }

}
```

- Intérêt : la classe Node n'est pas censée être utilisée en dehors de la classe myStack

Static vs Non-Static

- Considérez une version générique de la classe **myStack**

```
class myStack<T>{
        public static class Node{
                Node next;
                T data;
        }
        Node head;
        void push(T o) { Node n = new Node(); n.next=head; n.data=o;}
        T pop() { return head.data; }
}
```

Static vs Non-Static

- Considérez une version générique de la classe **myStack**

```
class myStack<T>{
        public static class Node{
                Node next;
                T data;
        }
        Node head;
        void push(T o) { Node n = new Node(); n.next=head;
n.data=o;}
        T pop() { return head.data; }
}
```

error: non-static type variable T cannot be referenced from a static context

Static vs Non-Static

- Correction 1 : non-static

```
class myStack<T>{
        public class Node{
                Node next;
                T data;
        }
        Node head;
        void push(T o) { Node n = new Node(); n.next=head; n.data=o;}
        T pop() { return head.data; }
}
```

OK ! On a un objet englobant → T est forcément défini.

Static vs Non-Static

- Correction 2 : static + générique

```
class myStack<T>{
        public static class Node<U>{
                Node<U> next;
                U data;
        }
        Node<T> head;
        void push(T o) { Node<T> n = new Node<T>();
n.next=head; n.data=o;}
        T pop() { return head.data; }
}
```

Classes Membres

- On peut définir des classes qui sont accessible que dans la classe englobante
  - On peut contrôler l'accès en dehors de la classe avec private, protected,...
    - Tous les membres sont toujours accessibles dans la classe (peu importe le modificateur d'accès). On change seulement la visibilité en dehors...
- Ça nous permet de cacher une partie de la fonctionnalité de notre programme, réutiliser des noms
- Static vs Non-static
  - Non-static : la classe n'a pas de sens sans un objet concret de la classe englobante
  - Cet objet est toujours accessible dans la classe interne
  - Static : les membres static (même privés) de la classe englobante sont accessibles.

Classes Locales

- Classe locale : une classe qu’on définit dans un bloc de code (i.e. dans une méthode) et ne peut être utilisé que dans ce bloc
  - Normalement, une telle classe est une sous-classe d’une classe existante (sinon, son type est inconnu en dehors du bloc)
  - … et redéfinit une méthode (sinon on aurait utilisé la classe originale)

Classes Locales

```
class MaCollection {
        Object[] data;
        MaCollection(Object [] l){ data=new Object[l.length];
              for(int i = 0; i < l.length; i++) data[i] = l[i];  }
       public Iterator<Object> parcourir(boolean up){
            class Iter implements Iterator<Object>{ … }
            return new Iter();
  }}
```

- Intérêt : la méthode **parcourir** retourne un objet d'une sous-classe de **Iterator**
- Cet objet doit avoir un comportement différent selon le paramètre

Classes Locales

```
class MaCollection {
    ...
    public Iterator<Object> parcourir(boolean up){
        class Iter implements Iterator<Object>{
            int i = up?0:data.length-1;
            public Object next() {
                if(up) return data[i++]; else return data[i--]; }
            public boolean hasNext(){
                if(up) return i<data.length; else return i>=0; }}
        return new Iter();
}}
```

Méthodes de l'interface Iterator

Classes Locales

```
class MaCollectionTest {
        public static void afficher(Iterator it){
                while(it.hasNext()){
                        System.out.println(it.next());
        }
}
public static void main(String[] args) {
        MaCollection m = new MaCollection(1,2,3,5,6,7);
        afficher(m.parcourir(true));
        afficher(m.parcourir(false));
}}
```

- Affiche 1,2,3,4,5,6,7 puis 7,6,5,4,3,2,1

Classes Locales - Règles

- Une classe locale peut utiliser
  - Les membres de la classe englobante
  - Les variables locales visibles dans le bloc
    - Mais : sous la conditions qu'elles soient « effectively final »
- Effectively final
  - Une variable qui est déclarée explicitement **final**
  - Ou une variable dont la valeur ne change jamais (aucune opération = effectuée)
- Dans l'exemple précédent :
  - up est effectively final
  - data est membre de la classe englobante

Classes Locales Anonymes

- Dans l’exemple précédent on a construit une classe locale appelée Iter
  - On n’utilise jamais ce nom (classe locale → nom n’existe pas en dehors de la méthode)
  - Ça sert à quoi ?
- Classe anonyme : une classe dont on a l’intention d’instancier un seul objet → pas besoin de nommer la classe

Classes Locales Anonymes

```
public Iterator<Object> parcourir(boolean up){
   return new Iterator<Object>(){
      int i = up?0:data.length-1;
      public Object next() { if(up) return data[i++]; else return
                                                    data[i--]; }
      public boolean hasNext(){ if(up) return i<data.length; else
                                                    return i>=0;
   };}
```

- Traduction : on définit une nouvelle classe (anonyme) qui implémente Iterator<Object>, on instancie un seul objet avec le constructeur par défaut **de la classe mére** (Object), on retourne cet objet.

Classes Locales Anonymes

```
class TypeDeBase {
    int i;
    TypeDeBase (int i) {this.i = i;};}
class AnonymousTest {
    public static void main (String args[]) {
    TypeDeBase obj = new TypeDeBase (3) {
        int j = 1;
        public String toString () { return i + " " + j; }
};
    System.out.println(obj); // 3 1
}}
```

- Une classe anonyme ne peut pas avoir de constructeur puisqu'elle n'a pas de nom.
  - initialisation de la classe mère : par new
  - initialisations des champs additionnels : par les initialisateurs ou dans les blocs d'initialisation

Expressions Lambda

- Introduites en Java 8
- Définissent un bloc de code qui peut être affecté / passé en paramètre, pour être exécuté plus tard
  - exemple typique : dans les interfaces graphiques on construit un objet “bouton” et on lui passe en paramètre la fonction qui doit être exécutée quand le bouton sera cliqué par l’utilisateur
- Les expressions lambda se rapprochent de la notion de passage des fonctions en paramètre qui existe dans les langages fonctionnels
- Il a toujours été possible en Java de passer du code en paramètre en créant expressément un objet qui contient ce code dans une méthode
- Mais cela est lourd et artificiel (même en utilisant des classes anonymes…)
- Les expression Lambda ont été introduites pour simplifier et rendre plus directe cette tâche

Interfaces Fonctionnelles

- On appelle une interface Fonctionnelle si elle possède une seule méthode.
- La bibliothèque de Java a plusieurs telles interfaces
  - **Comparator<T>**
    - **→ int compare(T o1, T o2)**
  - **Function<T,R>**
    - **→ R apply(T t)**
  - **Predicate<T>**
    - **→ boolean test(T t)**
  - **Supplier<T>**
    - **T get()**
  - **Consumer<T>**
    - **void accept(T t)**

Expressions Lambda

- Expression lambda = méthode raccourci de
  - Définir un sous-type (anonyme) d'une interface fonctionnelle
  - Redéfinir sa (seule) méthode
- Notation :
  - ([[Type]] arg1, [[Type]] arg2) - > { corps de la méthode }

    ou
  - ([[Type]] arg1, [[Type]] arg2) - > (expression de retour)
- Normalement on affecte l'expression lambda à une variable ou on la passe directement comme paramètre
  - → L'interface fonctionnelle implémentée peut être inférée par le compilateur, ainsi que les types des paramètres de la méthode.

Expressions Lambda

- Exemple :

  Comparator<String> c = (String first, String second) - >

  (first.length() - second.length()) ;
- Ou même

  Comparator<String> c = (first, second) - >

  (first.length() - second.length()) ;
- Quel est le type réel de la variable c ?
  - On définit une nouvelle classe (anonyme)
  - Cette classe implémente Comparator<String> (pourquoi?)
  - Elle redéfinit la méthode **compare** (seule méthode de l'interface)
    - → Les types des paramètres sont connus
  - On construit un objet de cette classe, l'affecte sur c
- Pour utiliser : **c.compare(''abc'', ''a'') ;** //Donne → 2

Expressions Lambda

- Exemple :

  Predicate<Point> pos = p - > (p.getX()>0 && p.getY()>0) ;

- Utilisation

  pos.test(new Point(2,3))// → true

  pos.test(new Point(-1,2))// → true

- Généralement : une expression lambda peut être utilisée où on attend un objet de type I, ou le type I est une interface fonctionnelle (définit une seule méthode)
- Le compilateur infère l'interface fonctionnelle pertinente selon l'usage

Exemple

- Dans la classe **Arrays** de la bibliothèque on a la méthode

  public static <T> void sort(T[] a, Comparator<? super T> c)

- Utilisation : on donne un tableau, est une fonction de comparaison pour trier le tableau
  - Plus précisément : un objet qui implémente l'interface Comparator, qui donne une méthode **compare**
- **NB** :
  - Méthode générique
  - Utilisation de <?super T> → on peut trier un tableau de **T** même si on a un **Comparator** pour une autre classe **P**, mais il **faut** que cette classe soit une super-classe de T (alors, le Comparator qu'on a fonctionne pour T)

Exemple

```
String [] t = {"Rome", "Athens", "Barcelona" };
Arrays.sort(t);
for(String s:t) System.out.println(s+", ");

Arrays.sort(t, (s1,s2)->(s1.length()-s2.length()));
for(String s:t) System.out.println(s+", ");

Arrays.sort(t,
     (s1,s2)→(s1.charAt(s1.length()-1)-s2.charAt(s2.length()-1)));
for(String s:t) System.out.println(s+", ");
```

Exemple

- La classe ArrayList contient la méthode **removeIf**

  public boolean removeIf(Predicate<? super E> filter)

- Cette méthode supprime de la liste tous les éléments pour lesquels le predicate donné retourne true

```
List<String> l = new ArrayList<>();
l.add("Rome"); l.add("Athens"); l.add("Nice"); l.add("Barcelona");
l.removeIf( s-> s.length()>5 );
System.out.println(l);
```

- Affiche [Rome, Nice]