Université Paris Diderot **Programmation Orientée Objet**
L2 **Année 2024-2025**

TD – Séance n° 7 - Correction

Classes Internes

Exercice 1 *Compréhension détaillée du cours*

```
public class A {
  private int iA;
  public static int sA;
  public void miA() {}
  public static void msA() {}

  public class B {
    private int iB;
    public static final int sB = 42;
    public void miB() {}
  }

  static class C {
    private int iC;
    public static int sC;
    public void miC() {}
    public static void msC() {}
  }
}
```

1. On se place à un point du code quelconque à l'extérieur de la classe `A`.
   (a) Que faut-il écrire pour créer une instance `b` de la classe interne `B` ?
   (b) Et pour créer une instance `c` de la classe interne `C` ?
   (c) Quelles sont les syntaxes :
   — d'une invocation sur `b` de la méthode d'instance `miB` ?
   — d'une invocation sur `c` de la méthode d'instance `miC` ?
   — d'une invocation de la méthode statique `msC` ?

   Peut-on alléger l'écriture si ce code est écrit dans la classe `A` ?

   Correction : Une instance de `B` ne peut être créée que comme englobée dans une instance de `A`. Une instance de `C` est créée indépendamment de toute instance de `A` (et ne peut qu'accéder aux éléments statiques de `A`) :

```
A a = new A();
A.B b = a.new B();
A.C c = new A.C();
```

```
b.miB();
c.miC();
A.C.msC();
```

Si l'on se trouve dans la classe `A`, les "`A.`" deviennent inutiles :

```
A a = new A();
B b = a.new B();
C c = new C();

b.miB();
c.miC();
C.msC();
```

2. On se place successivement :
   (a) dans la méthode d'instance `miB` de la classe interne `B`,
   (b) dans la méthode d'instance `miC` de la classe interne `C`,
   (c) dans la méthode statique `msC` de `C`.

   Dans chaque cas, peut-on écrire l'instruction `miA();` ?
   — Si la réponse est oui, sur quel objet la méthode `miA` sera-t-elle invoquée, et quelle syntaxe permet, au même emplacement dans le code, d'obtenir une référence vers cet objet ?
   — Si la réponse est non, pourquoi ?

   Correction : Dans le premier cas, la méthode sera invoquée sur `A.this`, l'instance de `A` englobant l'instance de `B` – il est d'ailleurs peut-être préférable d'écrire explicitement `A.this.miA()` (ce qui serait nécessaire en cas de conflit de noms de méthodes).

   Dans le second cas, il n'y a pas d'instance de `A` englobant l'instance de `C`, et l'instance de `C` n'accède qu'aux éléments statiques de la classe `A`.

   Dans le dernier cas, on est dans une méthode statique, qui comme d'habitude ne peut invoquer une méthode non statique, cette impossibilité n'étant pas même lié à la notion de classe interne.

3. Si l'on se place à l'extérieur de la classe `A`, quelle syntaxe permet d'accéder, en lecture (*e.g.* `int n =` *expr*`;`) aux champs statiques `sA`, `sB`, `sC` de la classe `A` et des classes internes `B` et `C` ? Cette écriture peut-elle être allégée si l'on se place dans la classe `A` ?

   Correction : `A.sA`, `A.B.sB`, `A.C.sC`, sans le `A.` si l'on se trouve dans la classe.

4. Une instance de la classe interne `B` possède-t-elle un champ `iA` ?

   Correction : Non. C'est l'instance de `A` englobant l'instance de `A.B` qui possède ce champ. On peut écrire `iA = 42` dans `miB`, mais cette écriture est trompeuse : elle signifie `A.this.iA = 42`.

5. Le champ statique `sB` pourrait-il être déclaré non final ?

Correction : Non, les champs statiques d'une classe interne non statique doivent être en `final`. C'est un choix (pas forcément très clair) des concepteurs du langage. *Mise à jour :* On peut les déclarer non `final` depuis la version 17 de JDK.

6. Peut-on déclarer une méthode statique dans `B` ?

   Correction : Non, là encore, c'est un choix de conception. *Mise à jour :* idem, c'est possible depuis la version 17 de JDK.

Exercice 2 *Itérateurs, classes internes, locales et anonymes.*

Un *itérateur* en Java est un objet capable de délivrer, à la demande, une certaine suite de valeurs. Un itérateur doit disposer de deux méthodes : une méthode permettant de déterminer s'il reste des valeurs à délivrer ; une méthode délivrant la valeur suivante.

L'interface prédéfinie `Iterator` est un exemple d'un tel couple de méthodes. Spécialisée à des instances de la classe `Integer`, l'interface `Iterator<Integer>` contiendra les deux méthodes suivantes :

```
// returns true if the iteration has more elements :
boolean hasNext();
// returns the next element in the iteration :
Integer next();
```

Le but de cet exercice est de construire, à l'aide de classes internes, plusieurs sortes d'itérateurs délivrant tout ou partie des valeurs d'un tableau d'instances de `Integer` : suite des valeurs de position paire, celles de position impaire, suite des valeurs inversée. Ce tableau sera encapsulé dans une instance de la classe suivante :

```
import java.util.Iterator;
public class Data {
  private Integer[] data;
  public Data(Integer... data) {
    this.data = data;
  }
  static private void print(Iterator<Integer> i) {
    while (i.hasNext()) {
      System.out.print(i.next() + " ");
    }
    System.out.println();
  }
  // à compléter
}
```

Les trois méthodes à ajouter à cette classe, qui afficheront les valeurs de `data` suivant les trois manières décrites ci-dessus, sont :

```
public void printEven()
public void printOdd()
public void printBackwards()
```

Chacune devra construire un itérateur approprié, qui sera passé à la méthode statique `print`.

1. Écrire dans la classe `Data` une classe interne privée `EvenIterator` implémentant l'interface `Iterator<Integer>`. Sa méthode `next()` devra délivrer la suite des valeurs du tableau `array` de position paires (`0`, `2`, ... `14`). En déduire une implémentation de `printEven`.

   La classe `EvenIterator` peut-elle être statique, ou doit-elle être non statique ?

2. Écrire la méthode `printOdd`, en définissant cette fois directement dans le corps de cette méthode une classe interne `OddIterator` implémentant l'interface `Iterator<Integer>`. Sa méthode `next()` devra délivrer la suite des valeurs du tableau `array` de position impaires (`1`, `3`, ... `15`).

3. Implémenter la méthode `printBackward`, en créant cette fois dans le corps de cette méthode une instance d'une classe interne *anonyme* implémentant l'interface `Iterator<Integer>`. Sa méthode `next()` devra délivrer la suite inversée des valeurs du tableau (`15`, `14`, ... `0`).

   Correction :

```
import java.util.Iterator;
public class Data {
  Integer[] data;
  public Data(Integer... data) {
    this.data = data;
  }

  private static void print(Iterator<Integer> i) {
    while (i.hasNext()) {
      System.out.print(i.next() + " ");
    }
    System.out.println();
  }

  private class EvenIterator implements Iterator<Integer
     > {
    int i = 0;
    public boolean hasNext() {
      return i < data.length;
    }
    public Integer next() {
      Integer retValue = data[i];
```

```
      i += 2;
      return retValue;
    }
  }

  public void printEven() {
    print(new EvenIterator());
  }

  public void printOdd() {

    class OddIterator implements Iterator<Integer> {
      int i = 1;
      public boolean hasNext() {
  return i < data.length;
      }
      public Integer next() {
  Integer retValue = data[i];
  i += 2;
  return retValue;
      }
    }

    print(new OddIterator());
  }

  public void printBackwards() {
    Iterator<Integer> it = new Iterator<>() {
      int i = data.length - 1;
      public boolean hasNext() {
  return (i >= 0);
      }
      public Integer next() {
  return data[i--];
      }
    };
    print(it);
  }

  public static void main(String s[]) {
    Data d = new Data
    (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,
       16);
    d.printEven();
    d.printOdd();
    d.printBackwards();
  }
}
```

Exercice 3 *Prédicats et classes anonymes.*

Considérons l'interface suivante :

```
interface Predicate {
  boolean test(int n)
}
```

Un objet implémentant cet interface peut être vu comme un "prédicat" sur les entiers, une condition vraie ou fausse selon la valeur de `n` – on dit que `n` *statisfait* le prédicat lorsque cette condition est vérifiée.

1. Montrez comment, dans une méthode quelconque, créer des objets implémentant l'interface `Predicate` et représentant les prédicats suivants :
   — "`n` est non nul".
   — "`n` est positif".
   Correction :

```
Predicate p = new Predicate() {
  public int test(int n) {
    return n != 0;
    // return n > 0;
  }
}
// ou mieux :
Predicate p = n -> n != 0;
// n -> n > 0;
```

2. Donner l'implémentation de la méthode suivante :

```
public static filter(int[] tab, Predicate pred)
```

   Cette méthode doit créer puis renvoyer un nouveau tableau, dans lequel seront placées la suite des valeurs de `tab` satisfaisant le prédicat représenté par `pred`.
   Correction :

```
public static int[] filter(int[] tab, Predicate pred) {
  int n = 0;
  for (int v : tab) {
    if (pred.test(v)) {
      n++;
    }
  }
  int[] tf = new int[n];
  int k = 0;
  for (int v : tab) {
    if (pred.test(v)) {
      tf[k++] = v;
    }
```

```
    }
    return tf;
}
```

Exercice 4 *Comparateurs, classes anonymes et lambda-expressions*

L'opérateur `<` permet de comparer des valeurs numériques suivant l'ordre naturel sur ces valeurs, mais rien n'empêche de définir, même sur de simples entiers, des ordres plus exotiques.

On peut se servir, pour définir un tel ordre, d'un *comparateur*, un objet implémentant un unique méthode permettant de déterminer, étant donnés deux entiers quelconques, si le premier est ou non plus petit que le second pour l'ordre considéré :

```
interface Comparator {
    // returns true if n is less than m
    // for the specified ordering :
    boolean isLessThan(int n, int m);
}
```

Le code d'un tri à bulles, par exemple, peut facilement être modifié pour trier les éléments d'un tableau suivant un ordre quelconque. Il suffit de passer de ceci :

```
static void sort(int[] tab) {
  // ...
  if (tab[i + 1] < tab[i]) {
    // ...
  }
  // ...
}
```

à cela :

```
static void sort(int[] tab, Comparator cmp) {
  // ...
  if (cmp(tab[i + 1], tab[i])) {
    // ...
  }
  // ...
}
```

où `cmp` sera une référence vers tout objet implémentant l'interface `Comparator` suivant un ordre donné.

1. Écrire les expressions permettant de créer trois instances de classes anonymes implémentant l'interface `Comparator`, et permettant respectivement de trier les éléments d'un tableau :
   (a) par ordre croissant,
   (b) par ordre decroissant,

(c) par ordre croissant en valeur absolue.

Correction :

```
new Comparator() {
  public boolean lessThan(int n, int m) {
    return n < m;
  }
}
// (n, m) -> n < m);

new Comparator() {
  public boolean lessThan(int n, int m) {
    return n > m;
  }
}
// (n, m) -> n > m);

new Comparator() {
  public boolean lessThan(int n, int m) {
    return Math.abs(n) < Math.abs(m);
  }
};
// (n, m) -> Math.abs(n) < Math.abs(m));
```

Remarque. Vous verrez, au prochain cours, comment alléger les trois écriture des trois invocations précédentes à l'aide de ce qu'on appelle les *lambda-expressions*.