Université de Paris **Programmation Orientée Objet**
L2 **Année 2020-2021**

TD - Séance n°2 - Correction

Révisions – Classes, Modélisation

Exercice 1 *Variables et méthodes statiques*
On définit une classe A par le code suivant.

```
1  public class A {
       public static int a = 3;
3      public int b;

5      public A (int c) {
           this.b = c;
7      }

9      public void g() {
           a = a+1;
11         b = b+1;
       }
13 }
```

1. On définit une méthode statique h dans la classe A. Quels champs peut-elle utiliser ? Même question pour une méthode non statique.

   Correction : h peut utiliser uniquement les champs statiques, en l'occurrence a. Une méthode non statique aurait accès à tous les champs, donc a et b.

2. On utilise la classe A dans une classe Test, comme ci-dessous. Qu'obtient on à l'exécution ?

```
1  public class Test {
       public static void main(String[] args) {
3          A u = new A(0);
           A v = new A(0);
5          u.g();
           System.out.println("u: a=" + u.a + "; b=" + u.b);
7          v.g();
           System.out.println("v: a=" + v.a + "; b=" + v.b);
9          u.g();
           System.out.println("u: a=" + u.a + "; b=" + u.b);
11     }
}
```

Correction :

```
u: a=4; b=1
v: a=5; b=1
u: a=6; b=2
```

Exercice 2 *Passage d'argument*

1. Qu'obtient-on à l'exécution du code suivant ?

```
public class Test2 {
    public static int g(int i) {
        i = i+1;
        return i;
    }
    public static void main(String[] args) {
        int i = 0;
        g(i);
        System.out.println(i);
    }
}
```

Correction :

```
0
```

2. On suppose maintenant avoir la classe C suivante (elle encapsule un entier).

```
public class C {
    private int a;

    public C(int a) {
        this.a = a;
    }

    public String toString() {
        return Integer.toString(a);
    }
    public void setNumber(int j) {
        this.a = j;
    }
}
```

— Qu'obtient on si on exécute le code suivant ?

```
public class Test3a {
    public static void h(C k) {
        k.setNumber(5);
    }
    public static void main(String[] args) {
        C k = new C(0);
```

```
            h(k);
8           System.out.println(k);
        }
10  }
```

Correction :
`5`

— Et si on exécute le code suivant ?

```
public class Test3b {
2       public static C h(C k) {
            k = new C(5);
4           return k;
        }
6       public static void main(String[] args) {
            C k = new C(0);
8           C l = h(k);
            System.out.println("k=" + k + ", l=" + l);
10      }
}
```

Correction :
`k=0, l=5`

3. Expliquez en quoi les exemples précédents illustrent le passage par valeur utilisé par java.

   Correction : `Test2` et `Test3b` illustrent le fait que dans une méthode, un paramètre est une variable fraîche initialisée avec une *copie* de la valeur de l'argument passé lors de l'appel de la méthode. Par exemple dans `Test2`, cela explique pourquoi l'instruction `i = i+1` dans le corps de la méthode `g` ne modifie pas la variable `i` déclarée dans le `main` lors de l'appel `g(i)`.
   `Test3a` illustre le fait que pour un paramètre qui représente un objet, puisque sa valeur est une *référence*[1], il pointe vers le même objet que l'argument passé lors de l'appel. Cela explique pourquoi l'instruction `k.setNumber(5)` dans le corps de la méthode `h` modifie le champ `a` de l'objet pointé par la variable `k` du `main` lors de l'appel `h(k)`.

Exercice 3 *L'horloge – Compteur cyclique*

Le but de cet exercice et du suivant est de définir une classe pour créer des horloges. On va commencer par la création d'une classe `CompteurCyclique` qui va nous aider à la conception de l'horloge.

Un *compteur cyclique* est un compteur de nombres entiers ayant une valeur maximale **fixe** qu'il peut atteindre. Un compteur cyclique s'initialise par zéro ; il

1. c'est-à-dire l'adresse dans le *tas* où l'on peut trouver les valeurs des différents membres de l'objet

possède, en plus, la fonctionnalité `incrementer` qui le fait augmenter sa valeur courante par un jusqu'à ce qu'il atteigne sa valeur maximale. Lorsque cette valeur est atteinte, le compteur se réinitialise automatiquement à 0.

1. Déterminer les champs de la classe `CompteurCyclique`. Quels champs doivent être définis `final` ?

   Correction :

```
1 private int valeur;
  private final int max;
```

2. Créer un constructeur `public CompteurCyclique(int max)`, qui définit un compteur cyclique de valeur maximale `max` et initialise sa valeur courante à 0 ; un autre constructeur qui permette en plus d'initialiser la valeur courante du compteur cyclique à une autre valeur passée en argument.

   Correction :

```
  public CompteurCyclique(int max) {
2     this(max, 0);
  }
4 public CompteurCyclique(int max, int valeur) {
      this.max = max;
6     this.valeur = valeur;
  }
```

3. Ajouter une méthode *getter* pour la valeur courante du compteur.

   Correction :

```
1 public int getValeur() {
      return valeur;
3 }
```

4. Peut-on définir une méthode *setter* pour la valeur maximale ?

   Correction : Non, le champ `max` est final.

5. Définir les méthodes suivantes :
   - `public void reinitialiser()` qui met la valeur courante du compteur à zéro.
   - `public boolean incrementer()` qui augmente la valeur courante du compteur de la manière décrite dans la définition ci-dessus. La méthode retourne `true` dans le cas de réinitialisation, `false` autrement.

     Correction :

```
1 public void reinitialiser() {
      valeur = 0;
3 }
  public boolean incrementer() {
5     boolean reinit = false;
```

```
        valeur++;
7       if (valeur >= max) {
            reinitialiser();
9           reinit = true;
        }
11      return reinit;
    }
```

6. Redéfinir la méthode `public String toString()` pour qu'elle affiche la valeur du compteur en utilisant deux chiffres (si la valeur est <10, ajouter un 0 devant la valeur).

   • Facultatif : Plutôt que 2 chiffres, on peut utiliser le nombre de chiffres de la valeur maximale, en rajoutant autant de zéros que nécessaire.

   • Remarque : Penser à utiliser la méthode `String.format`. Par example, `String.format("%05d", 24);` affiche 00024. Pour trouver le nombre de chiffres d'un entier n, utiliser soit `Integer.toString(n).length()`, soit `String.valueOf(n).length()`, soit `(int)Math.log10(n)+1`.

   Correction :

```
    @Override
2   public String toString() {
        int nbChiffres = Integer.toString(max).length();
4       String fmt = "%0" + Integer.toString(nbChiffres) + "d";
        return String.format(fmt, valeur);
6   }
```

Exercice 4 *L'horloge – La classe principale*

Maintenant, on va concevoir la classe Horloge avec ses méthodes. Une horloge consiste de deux champs, les *heures* et les *minutes*, qui sont de type `CompteurCyclique`.

1. Donner un constructeur de la classe Horloge.

   Correction :

```
    public Horloge(int h, int m) {
2       heures = new CompteurCyclique(12, h);
        minutes = new CompteurCyclique(60, m);
4   }
```

2. Définir une méthode setter `public void setHeure(int h, int m)`.

   Correction :

```
    public void setHeure(int h, int m) {
2       heures = new CompteurCyclique(12, h);
        minutes = new CompteurCyclique(60, m);
4   }
```

3. Surcharger la méthode `setHeure` pour qu'elle modifie les heures. Peut-on surcharger la méthode à nouveau pour modifier les minutes ?

   Correction :

```
public void setHeure(int h) {
    heures = new CompteurCyclique(12, h);
}
```

   On ne peut pas la surcharger, car les signatures seraient identiques. En effet, seuls le nom de la méthode et les types des arguments sont pris en compte dans la signature, et les heures et les minutes sont toutes deux de type `int`. Un *workaround* serait de créer une classe *wrapper* `Minute` qui simule un entier. Cela a pour inconvénient d'alourdir l'usage de la méthode, mais pour avantage qu'on peut définir une logique particulière pour les minutes (par exemple restreindre dans le constructeur de `Minute` aux entiers compris entre 0 et 59).

4. Définir une méthode `public void ticTac()` qui simule le passage d'un minute. Utiliser la méthode `public void incrementer()` de la classe `CompteurCyclique`.

   Correction :

```
public void ticTac() {
    if (minutes.incrementer()) {
        heures.incrementer();
    }
}
```

5. Redéfinir la méthode `public String toString()` pour qu'elle affiche l'heure actuelle de l'horloge.

   Correction :

```
@Override
public String toString() {
    return heures.toString() + ":" + minutes.toString();
}
```

Si vous avez le temps

Exercice 5 *Le parking - Ticket*

On veut modéliser le fonctionnement des parkings automobiles, qui se caractérisent par leurs :

— nombre de places
— nombre de places libres
— emplacements (l'endroit où se trouvent les places)
— pourcentage de remplissage

Lorsqu'une voiture entre dans un parking, elle reçoit un ticket qu'elle devra rendre en sortant.

1. Discutez la nature des emplacements et proposez plusieurs modélisations dans le cadre du parking. Une fois fixée la nature des emplacements, que pouvez vous dire des autres caractéristiques d'un parking ?
2. On s'intéresse à présent aux tickets, que nous n'avons pas encore modélisé. On décide que c'est lorsqu'elle rentre dans un parking qu'une voiture reçoit un ticket, s'il n'y a pas de place elle n'en recevra pas. Quelles propositions parmi les suivantes pourriez vous retenir ?
   (a) avoir un constructeur `Ticket(int n,Parking p)`
   (b) avoir un constructeur `Ticket(Parking p)`
   (c) avoir une méthode `Ticket getTicket()` dans la classe *Voiture*
   (d) avoir une méthode `Ticket entreParking(Parking p)` dans la classe *Voiture*
   (e) avoir une méthode `Ticket entreParking(Voiture v)` dans la classe *Parking*
   (f) proposez une autre méthode et l'évaluer.

Exercice 6 *Le parking - Places libres*

Dans cet exercice, on supposera la classe `Voiture` déjà écrite et on ne se préoccupera pas du ticket. Notre objectif est d'optimiser la recherche d'une place libre. Ainsi on aura :

```
import java.util.LinkedList;
public class Parking {
    private final Voiture[] places;
    private final LinkedList<Integer> libres;
}
```

L'emplacement d'une voiture sera confondu avec l'indice qu'elle occupe dans le tableau `places`, et la liste `libres` contiendra l'ensemble des emplacements encore libres.

Rappels sur `LinkedList<T>` et `Integer` :

— Les listes sont associées à un type, ce qui assure l'uniformité des objets contenus

— La classe qui permet de manipuler des `int` non pas comme type primitif mais comme des objets est `Integer`. Vous pouvez utiliser le constructeur `Integer(int)` et la méthode `int intValue()` de la classe `Integer` pour passer explicitement d'un `int` à un `Integer` et réciproquement.

— Sur les `LinkedList<Integer>` vous pourrez utiliser les méthodes : `boolean add(Integer)`, `boolean isEmpty()`, `Integer removeFirst()` et le constructeur `LinkedList()`

Ecrivez dans la classe `Parking` :

1. Pourquoi les champs de `Parking` sont-ils déclarés `final` ? Le nombre de voiture dans le parking pourra-t-il quand même varier ?
2. Un constructeur `Parking (int n)` où l'entier n désigne la taille du parking
3. Une méthode `public int prendPlace(Voiture x)` qui place la voiture x à un endroit libre, et retourne cet emplacement
4. Une méthode `public Voiture exitPlace(int n)` qui fait sortir la voiture qui se trouve à l'emplacement n

Exercice 7 *Le parking - Emplacements*

Dans cet exercice on choisit de ne pas utiliser de `LinkedList` mais de coder la liste des places libres dans l'objet `Emplacement`.

En voici le fonctionnement : un emplacement se définit par une référence vers un véhicule, et par le numéro du prochain emplacement libre. Un parking se compose d'un tableau d'emplacements, et d'une variable indiquant un emplacement libre particulier, duquel on peut déduire un second emplacement libre etc...

On pourra utiliser -1 pour identifier un numéro d'emplacement non valable.

1. Ecrivez une modélisation de la classe `Emplacement` :
   - un constructeur `Emplacement(int x)` qui prend en argument le numéro du prochain emplacement libre.
   - une méthode `void setNextLibre(int x)` qui permet de modifier cet attribut
   - un accesseur `int getNextLibre()`
   - une méthode `void positionne(Voiture v)`
   - une méthode `Voiture setFree(int n)` qui en plus de libérer la voiture prend n pour prochain emplacement libre.
2. Ecrivez une modélisation de la classe `Parking` :
   - un constructeur `Parking(int taille)`.
   - `int prendPlace(Voiture x)`
   - `Voiture exitPlace(int n)`