TD - Séance n°2

Révisions – Classes, Modélisation

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

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

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

    public void g() {
        a = a+1;
        b = b+1;
    }
    public String toString(){
        return  "a = " + a + ", b = "+ b;
    }
}
```

1. On souhaite ajouter à la classe `A` une méthode statique. Quels champs peut-elle utiliser ? Même question pour une méthode non statique.
2. On utilise la classe `A` dans une classe `Test`, comme ci-dessous. Qu'obtient on à l'exécution ?

```
public class Test {
    public static void main(String[] args) {
        A u = new A(0);
        A v = new A(0);
        System.out.println("u: " + u);
        System.out.println("v: " + v);
        u.g();
        System.out.println("u: " + u);
        v.g();
        System.out.println("v: " + v);
        u.g();
        System.out.println("u: " + u);
    }
}
```

Exercice 2 *Passage d'argument*

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

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

2. On définit la classe `C`, encapsulant un entier, par le code suivant :

```
public class C {
    private int a;

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

    public String toString() {
        return Integer.toString(a);
    }

    public void setNumber(int a) {
        this.a = a;
    }
}
```

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

```
public class Test3a {
    public static void g(C u) {
        u.setNumber(5);
    }

    public static void main(String[] args) {
        C v = new C(0);
        g(v);
        System.out.println(v);
    }
}
```

— Et si on exécute le code suivant ?

```
public class Test3b {
    public static C g(C u) {
```

```
        u = new C(5);
        return u;
    }

    public static void main(String[] args) {
        C v = new C(0);
        C w = g(v);
        System.out.println("v=" + v + ", w=" + w);
    }
}
```

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

Exercice 3 *L'horloge – Compteur cyclique*

Le but de cet exercice et du suivant est de définir une classe pour créer des horloges.

Un *compteur cyclique* est un compteur prenant des valeurs entières dans un certain intervalle $[0, \ldots,$ `sup`$[$ où `sup` est une borne **fixe**. Il s'initialise à zéro. Il possède une fonctionnalité `incrementer` qui lui fait augmenter sa valeur courante d'une unité, à moins que cette incrémentation ne produise la valeur `sup` : dans ce cas, le compteur revient à 0.

On souhaite définir ici une classe `CompteurCyclique` permettant de représenter ces compteurs.

1. Déterminer les champs de la classe `CompteurCyclique`. Quels champs doivent être définis `final` ?
2. En choisissant un nom adéquat, ajouter une méthode de type *setter* permettant de modifier la valeur courante du compteur.

   Question supplémentaire : gérer le cas où la valeur passée au *setter* n'est pas dans l'intervalle $[0, \ldots,$ `sup`$[$. On la transformera en l'unique valeur de cet intervalle égale à la valeur passée modulo `sup`.
3. Peut-on définir une méthode de type *setter* pour la borne du compteur ?
4. Créer un constructeur `public CompteurCyclique(int sup)` qui produit une instance de borne `sup` et initialise sa valeur courante à 0.

   Ecrivez un second constructeur qui permette en plus d'initialiser la valeur courante du compteur cyclique à une autre valeur passée en argument. Remarquez que vous pouvez faire dépendre le premier constructeur du second, et invoquer la méthode précédente.
5. Ajouter une méthode de type *getter*, en choisissant un nom adéquat, qui permette d'obtenir la valeur courante du compteur.
6. Définir les méthodes suivantes :
   — `public void reinitialiser()` qui remet à zéro la valeur courante du compteur.

— `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 remise à zéro du compteur, et `false` sinon.

7. 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 (`sup - 1`), en rajoutant autant de zéros que nécessaire.

   • Remarque : Pensez à utiliser la méthode `String.format`. Par exemple, `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`.

Exercice 4 *L'horloge – La classe principale*

On peut voir une horloge comme une combinaison des objets que l'on vient de définir. En effet les décomptes respectifs des *heures* et des *minutes* peuvent être vus comme deux compteurs cycliques (de bornes respectives 24 et 60).

1. Définir et donner des constructeurs à une classe `Horloge`.
2. Définir une méthode *setter* `public void setHeure(int h, int m)`.
3. Surcharger la méthode `setHeure` pour qu'elle modifie les heures. Peut-on surcharger la méthode à nouveau pour modifier les minutes ?
4. Définir une méthode `public void incrementer()` qui simule l'ajout d'une minute à l'heure courante de l'horloge (utiliser la méthode `incrementer` de la classe `CompteurCyclique`).
5. Redéfinir la méthode `public String toString()` pour qu'elle affiche l'heure actuelle de l'horloge.

Si vous avez le temps

Ces exercices de modélisation vous sembleront ouverts. L'intérêt est de ne pas avoir de solution à priori et d'explorer des approches différentes, en discutant leurs avantages, leurs inconvénients et leur réalisme. Vous pourrez terminer chez vous.

Exercice 5 *Le parking - Ticket*

On veut modéliser le fonctionnement de parkings automobiles caractérisés par :
— leur nombre de places,
— leur nombre de places libres,
— les emplacements de ces places.
Lorsqu'une voiture entre dans un parking, elle reçoit un ticket qu'elle devra rendre en sortant.

1. Discuter la nature des emplacements et proposer plusieurs modélisations pour un parking. Une fois fixée la nature des emplacements, que pouvez-vous dire des autres caractéristiques d'un parking ?
2. Si on s'intéresse plus particulièrement aux objets qui représentent les tickets, on peut décider que c'est lorsqu'elle rentre dans un parking qu'une voiture reçoit un ticket, et que 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`.

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 :

```
1 import java.util.LinkedList;
2 import java.lang.Integer;
3 public class Parking {
4     private final Voiture[] places;
5     private final LinkedList<Integer> libres;
6 }
```

Les emplacements seront confondus avec les indices du tableau `places`. Une place occupée sera représentée dans ce tableau par une référence vers une instance de `Voiture`, une place libre par une référence nulle. La liste `libres` contiendra l'ensemble des emplacements encore libres dans le tableau.

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

— Une liste est associée à un type de référence permettant de manipuler de manière uniforme les objets qu'elle contient.

— La classe `Integer` permet de construire des objets encapsulant une valeur de type `int`. La méthode statique `Integer valueOf(int n)` et la méthode (non statique) `int intValue()` de la classe `Integer` permettent de passer explicitement d'un `int` à un `Integer` et réciproquement.

— Sur les `LinkedList<Integer>` vous pouvez utiliser les méthodes : `boolean add(Integer)`, `boolean isEmpty()`, `Integer remove()` [1] et le constructeur `LinkedList()`.

1. Ces listes étant doublement chaînées, les ajouts et suppressions se feront en fin de liste, mais en temps constant.

Dans la classe `Parking` :

1. Pourquoi les champs sont-ils déclarés `final` ? Le nombre de voitures dans le parking pourra-t-il quand même varier ?
2. Proposer un constructeur `Parking (int n)`, où `n` désigne le nombre de places du parking.
3. Écrire une méthode `public int prendPlace(Voiture v)` qui place la voiture `v` dans l'une des places libres du parking si celle-ci existe, et retourne son emplacement. Si aucune place n'est libre, la méthode renverra `-1`.
4. Écrire une méthode `public Voiture liberePlace(int n)` qui fait sortir du parking la voiture se trouvant à l'emplacement `n`, à condition que cet emplacement soit valide et effectivement occupé – dans le cas contraire, la méthode renverra `null`.

Exercice 7 *Le parking - Places* Dans cet exercice, plus difficile, on choisit de ne pas utiliser de `LinkedList` mais de coder la liste des places libres d'un parking par le chaînage simple d'éléments d'un tableau. Dans leurs versions de base, les deux classes principales seront :

```
public class Place {
    private Voiture voiture;
    private int suivante;
}
public class Parking {
    private final Place[] places;
    private int libre;
}
```

Le principe de l'encodage est le suivant :

— Chaque élément du tableau `places` est une instance de `Place`. Les places occupées ont un champ `voiture` pointant vers une instance de `Voiture`, les places libres ont un champ `voiture` égal à `null`.

— Le champ `libre` est la position dans le tableau `places` d'une première place libre. Le champ `suivant` de cette place indique la position d'une seconde place libre. Le champ `suivant` de cette seconde place indique la position d'une troisième place libre, etc. La dernière place libre atteinte par ce chaînage a un champ `suivant` égal à `-1`.

— L'implémentation doit garantir que l'ensemble des places libres accessibles à partir la valeur courante `libre` soit l'ensemble de toutes les places encore libres du parking, et seulement celles-ci. Peu importe la valeur du champ `suivant` des places occupées.

La suppression (par occupation) ou l'ajout (par libération) d'une place libre dans cette chaîne se fera toujours au début du chaînage :

— Pour ajouter une voiture au parking, on la place dans le première place libre, et le successeur de cette place devient la nouvelle première place libre.

— Pour supprimer une voiture du parking, on annule le champ `voiture` de sa place, la première place libre devient le successeur de cette place libérée, et la place libérée devient la nouvelle première place libre.

1. Completer la classe `Place` en lui ajoutant :
   — un constructeur `Place(int suivant)` (une place est toujours initialement vide),
   — des *getters* et des *setters* pour les différents champs.
   — éventuellement, une méthode `void liberer(int suivant)` permettant simultanément de libérer une place et d'indiquer son successeur.
2. Compléter la classe `Parking`, avec comme précédemment :
   — `Parking(int n)`.
   — `int prendPlace(Voiture v)`
   — `Voiture liberePlace(int n)`