LU2IN002-2022oct
Éléments de programmation
par objets avec Java

Examen du 16 juin 2023 – Durée : 2 heures

Seul document autorisé : **feuille A4 manuscrite, recto-verso.**
Pas de calculatrice ou téléphone. Barème indicatif sur 43.

Partie 1 Exercices (14 points)

Exercice 1 (5 points) Compter des trucs...

Soit le programme suivant :

```
1  public class A { }
2  public class Truc {
3      private A a1, a2;
4      public Truc() {a1=new A(); a2=new A
           ();}
5      public Truc(A a) { a1=a; a2=a; }
6      public void empty() {a1=null; a2=
           null;}
7  }
```

```
8  public class Test {
9      public static void main(String []
           args){
10         Truc t1=new Truc();
11         Truc t2=t1;
12         Truc t3=new Truc(new A());
13         t2.empty();
14         t3=null;
15     }
16 }
```

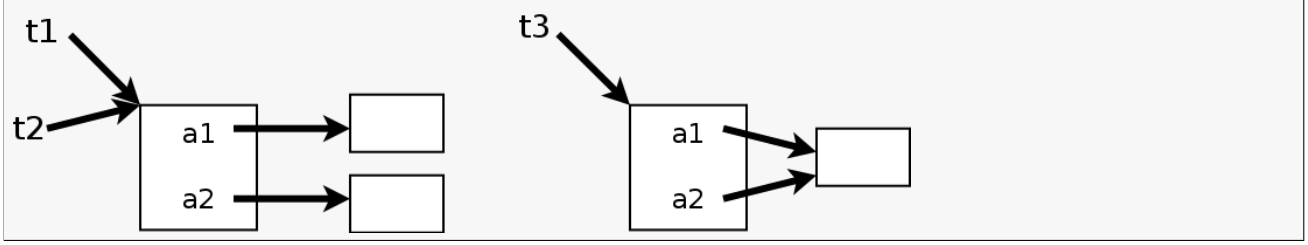
Q1.1 Combien y-a-t-il d'instances de la classe `Truc` créées ? Donner le numéro des lignes où sont créées chacune de ces instances.

Il y a 2 objets de type `Truc` créés aux lignes 10 et 12.

Q1.2 Mêmes questions pour les instances de la classe `A` créées.

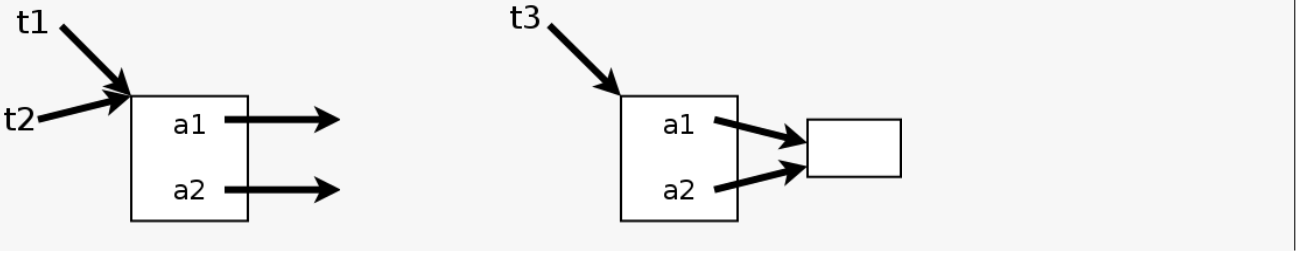
Il y a 3 objets de type `A` créés : deux objets créés à la ligne 4 (appelée par la ligne 10) et un objet créé à la ligne 12.

Q1.3 Faire un schéma des *handles* et des objets dans la mémoire après l'exécution de la ligne 12.

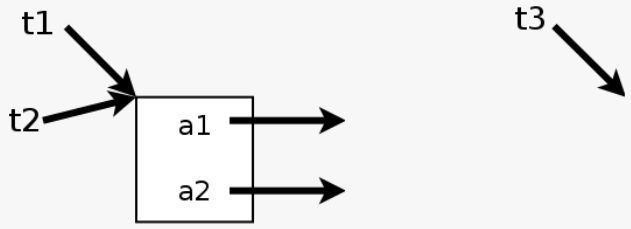


Q1.4 Après l'exécution de la ligne 14 et avant la fin du programme, donner le nombre d'instances de la classe `Truc` et le nombre d'instances de la classe `A` présentes. Expliquer.

Il reste 1 objet `Truc`, et aucun objet `A`.
Aprés l'exécution de l'instruction : `t2.empty()`, le garbage collector libère la mémoire pour les deux objets `A` de l'objet `Truc` référencé par `t1` et `t2`.



Aprés l'exécution de l'instruction : `t3=null`, le garbage collector libère la mémoire pour l'objet `Truc` référencé par `t3` et pour son objet `A` (car il n'est plus référencé par aucune variable).



Exercice 2 (5 points) Cocotiers et noix de coco.

On considère les classes suivantes :

```
1  public class NoixDeCoco {
2      private double poids;
3      public NoixDeCoco(){ poids=Math.random()*1000+500; } }
4  public class Cocotier {
5      private int taille;
6      private NoixDeCoco [] tab;
7      public Cocotier(int taille,int nbNoix) {
8          this.taille=taille;
9          tab=new NoixDeCoco[nbNoix];
10         for(int i=0;i<tab.length;i++) {
11             tab[i]=new NoixDeCoco();
12         }
13     } }
```

Q2.1 Donner le code – au choix – du constructeur par copie ou de la méthode `clone()` de la classe `NoixDeCoco`.

Solution : constructeur par copie

```
1  public NoixDeCoco(NoixDeCoco n) {
2      this.poids=n.poids;
3  }
```

Solution : méthode clone()

```
1  public NoixDeCoco clone() {
2      NoixDeCoco n=new NoixDeCoco();
3      n.poids=this.poids;
4      return n;
5  }
```

Q2.2 Donner le code – au choix – du constructeur par copie ou de la méthode `clone()` de la classe `Cocotier`.

Il faut penser à copier chacune des noix de cocos.

Solution : constructeur par copie

```
1  public Cocotier(Cocotier c) {
2      this.taille=c.taille;
3      tab=new NoixDeCoco[c.tab.length];
4      for(int i=0;i<tab.length;i++) {
5          tab[i]=new NoixDeCoco(c.tab[i]);
6      }
7  }
```

Solution : méthode clone()

```
1  public Cocotier clone() {
2      Cocotier c=new Cocotier(taille,tab.length);
3      for(int i=0;i<tab.length;i++) {
4          c.tab[i]=tab[i].clone();
5      }
6      return c;
```

```
}
```

Solution : méthode clone() sans création inutile d'objets `NoixDeCoco`

```
public Cocotier clone() {
    return new Cocotier(this);
}
```

Q2.3 Soit l'instruction `Cocotier c1=new Cocotier(10,50);` donner l'instruction pour créer une copie de l'objet référencé par `c1`.

Dépend des réponses aux questions précédentes.

```
Cocotier c1=new Cocotier(10,50);
Cocotier c2=new Cocotier(c1);
Cocotier c3=c1.clone();
```

Exercice 3 (4 points) Gérer un échiquier

Soient les classes suivantes d'un jeu d'échecs :

```
public abstract class Piece {
  public abstract void afficher();
}
public class Tour extends Piece {
    public void afficher() {
        System.out.println("Je suis
            une tour"); }
}
```

```
public class Cavalier extends Piece {
    public void afficher() {
        System.out.println("Je suis un
            cavalier"); }
}
```

On suppose que l'on se trouve dans la méthode `main` d'une classe `TestEchiquier`. On rappelle qu'un échiquier est constitué de 64 cases (correspondant à 8 lignes de 8 colonnes de cases) et qu'une case peut être soit vide soit occupée par une pièce (et une seule).

- déclarer la variable `echiquier` comme un tableau de `Piece` à 2 dimensions de 8 cases sur 8 cases.
- ajouter un cavalier sur la première ligne, deuxième colonne de l'échiquier.
- ajouter une tour dans la première colonne, le numéro de la ligne étant choisi aléatoirement.
- écrire les instructions pour afficher toutes les pièces présentes sur l'échiquier (on suppose que d'autres pièces peuvent se trouver sur l'échiquier).

```
Piece [][] echiquier=new Piece [8][8];

echiquier[0][1]=new Cavalier();

echiquier[(int)(Math.random()*echiquier.length)][0]=new Tour();

for(int i=0;i<echiquier.length;i++) {
    for(int j=0;j<echiquier[i].length;j++) {
        if (echiquier[i][j]!=null) {
            echiquier[i][j].afficher();
        }
    }
}
```

Partie 2 Problème : livraisons de colis (29 points)

Remarque : *Si la méthode* `toString` *n'est pas demandée dans la question, il n'est pas nécessaire de la fournir, par contre, n'oubliez pas de définir les méthodes* `abstract` *quand cela est nécessaire.*

On veut modéliser un système de livraisons qui permet de livrer toutes sortes de choses livrables (colis, meubles livrables, repas livrables...). On suppose que l'on a des camions qui récupèrent des livrables dans des usines ou dans des dépôts, et qui les livrent dans des maisons de particulier ou dans d'autres dépôts. Les seuls bâtiments de la modélisation sont les usines, les maisons et les dépôts. Un dépôt permet de

stocker des livrables. Un camion stocke les livrables le temps du transport. Un colis payant est un colis qui a en plus un prix. On suppose qu'un meuble ne peut pas être livré sauf si il est livrable.

On veut pouvoir connaître le numéro, la dimension et le poids de tous les livrables. On considère pour cela l'interface `Livrable` ci-contre, ainsi que la classe `Dimension` ci-dessous.

```java
public interface Livrable {
   public int getNumero();
   public Dimension getDimension();
   public double getPoids();
}
```

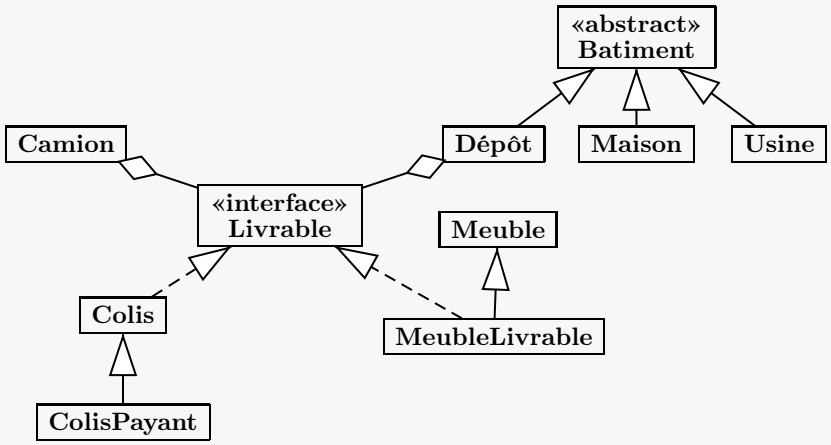
```java
public final class Dimension {
   public final int longueur;
   public final int largeur;

   public Dimension(int lng, int lrg) {
      longueur=lng;
      largeur=lrg;
   }
   public Dimension() {
      this((int)(Math.random()*100)+1,
           (int)(Math.random()*100)+1);
   }
   public String toString() {
      return longueur+"x"+largeur;
   }
}
```

Q4.1 (2 points) (a) Dessiner le diagramme de classes UML (sans les attributs ni les méthodes) entre les classes `Batiment`, `Camion`, `Colis`, `ColisPayant`, `Depot`, `Maison`, `Meuble`, `MeubleLivrable`, `Usine` et l'interface `Livrable`. **(b)** Indiquer sur le schéma la (ou les) classe(s) qui pourraient être abstraites.

(a) Voir diagramme UML ci-contre. On pourrait accepter une composition entre `Usine` et `Livrable` si on suppose que l'usine stocke les livrables qu'elle produit. La classe `Dimension` et la classe `Casier` (vue plus loin) ne sont pas demandées.

(b) La classe qui pourrait être abstraite est `Batiment` parce que dans cette modélisation : "Les seuls bâtiments de la modélisation sont les usines, les maisons et les dépôts". Il ne peut pas y avoir de bâtiment, qui ne soit seulement qu'un bâtiment.



Q4.2 (3 points) Écrire la classe `Colis` qui implémente l'interface `Livrable`. Le numéro du colis sera généré automatiquement à l'aide d'un compteur et commencera à 10001 (le premier colis aura le numéro 10001, le deuxième 10002...). Cette classe contient un constructeur prenant en paramètre la dimension et le poids du colis, ainsi qu'un deuxième constructeur prenant en paramètre seulement le poids. Écrire aussi une méthode `toString()`.

Attention : implémenter l'interface = écrire `"implements Livrable"`, mais aussi définir les méthodes de l'interface, sinon la classe doit être abstraite

```java
public class Colis implements Livrable {
    private static int cpt=10000;
    private int numero;
    private double poids;
    private Dimension dim;
    public Colis(Dimension dim, double poids) {
        cpt++;
        numero=cpt;
        this.dim=dim;
        this.poids=poids;
    }
    public Colis(double poids) {
        this(new Dimension(),poids);
    }
```

```
15      public int getNumero() { return numero; }
16      public Dimension getDimension() { return dim; }
17      public double getPoids() { return poids; }
18      public String toString() {
19          return "Colis "+numero+" "+dim+" "+poids+"kg";
20      }
21  }
```

Q4.3 (3 points) Un colis payant est un colis qui a un prix. Écrire la classe `ColisPayant` avec :
- une constante `PRIX_STANDARD` initialisée à 10 euros correspondant au prix standard d'un colis payant (le prix standard peut être connu par tous),
- un attribut `prix` (`double`),
- un constructeur à 3 paramètres,
- un constructeur à 1 paramètre qui initialise le prix au prix standard,
- une méthode `toString()`.

```
1  public class ColisPayant extends Colis {
2      public static final double PRIX_STANDARD=10;
3      private double prix;
4      public ColisPayant(Dimension dim, double poids, double prix) {
5          super(dim,poids);
6          this.prix=prix;
7      }
8      public ColisPayant(double poids) {
9          super(poids);
10         prix=PRIX_STANDARD;
11     }
12     public String toString() {
13         return super.toString()+" prix="+prix+" euros";
14     }
15 }
```

Interfaces `Receveur` et `Livreur` Les maisons, les dépôts et les camions peuvent recevoir des livrables. Les usines, les dépôts et les camions peuvent livrer des livrables.

Q4.4 (2 point) (a) Écrire l'interface `Receveur` qui contient une méthode `recevoir` qui prend en paramètre un livrable et qui retourne vrai si le livrable a bien été reçu, faux sinon. **(b)** Écrire l'interface `Livreur` qui contient une méthode `livrer` sans paramètre qui retourne le premier livrable accessible ou `null` sinon.

```
1  public interface Receveur {
2      public boolean recevoir(Livrable liv);
3  }
4  public interface Livreur {
5      public Livrable livrer();
6  }
```

Q4.5 (4 points) Un camion peut recevoir et livrer des livrables. Le chargement du camion est représenté par un tableau de livrables. Quand un camion reçoit un livrable, il est inséré dans la première case libre du tableau seulement si le livrable n'est pas `null`, si le poids maximal du chargement n'est pas dépassé et s'il reste de la place. Écrire la classe `Camion` avec notamment les attributs et méthodes suivants :
- `tabLiv` : un tableau de livrables,
- `poidsMax` : le poids maximal que le camion peut transporter,
- constructeur prenant en paramètre `nbMax` le nombre maximal de livrables qu'il peut transporter et `poidsMax` le poids maximal qu'il peut transporter,
- méthode `double getPoidsChargement()` qui retourne la somme des poids de tous les livrables dans le camion.

Attention : certaines cases du tableau peuvent être vides.

Remarques : les affichages ne sont pas demandés

```
public class Camion implements Receveur, Livreur {
    private Livrable [] tabLiv;
    private double poidsMax;

    public Camion(int nbMax, double poidsMax) {
        tabLiv=new Livrable[nbMax];
        this.poidsMax=poidsMax;
    }
    public double getPoidsChargement() {
        double somme=0;
        for(Livrable liv : tabLiv) {
            if(liv!=null) {
                somme+=liv.getPoids();
            }
        }
        return somme;
    }
    public boolean recevoir(Livrable liv) {
        if (liv==null) {
            System.out.println("Erreur : le livrable est null");
            return false;
        }
        if (getPoidsChargement()+liv.getPoids()>=poidsMax) {
            System.out.println("Erreur : trop lourd : "+getPoidsChargement()+"+"
               +liv.getPoids()+">="+poidsMax);
            return false;
        }
        for(int i=0;i<tabLiv.length;i++) {
            if(tabLiv[i]==null) {
                tabLiv[i]=liv;
                return true;
            }
        }
        System.out.println("Erreur : plus de place");
        return false;
    }
    public Livrable livrer() {
        for(int i=0;i<tabLiv.length;i++) {
            if (tabLiv[i]!=null) {
                Livrable liv=tabLiv[i];
                tabLiv[i]=null;
                return liv;
            }
        }
        System.out.println("Erreur : aucun livrable dans le camion");
        return null;
    }
}
```

Q4.6 (3 points) Un bâtiment a une adresse (`String`) et un type (chaîne de caractères, par exemple, "habitation","industriel") qui dépend des classes filles. On ne veut pas définir de variable d'instance `type` dans la classe `Batiment` ni dans ses classes filles, mais on veut pouvoir connaître le type de toutes les classes filles de `Batiment`. Écrire une classe `Batiment` avec obligatoirement un seul attribut appelé `adresse` (l'adresse peut être connue par les classes filles, mais ne doit pas pouvoir être modifiée) et un constructeur prenant un seul paramètre l'adresse du bâtiment. Ajouter une méthode pour que l'on puisse connaître le type de tous les bâtiments. Écrire aussi la méthode `toString()` qui doit retourner une chaîne avec l'adresse et le type du bâtiment.

Solution : on ajoute une méthode abstraite `getType()` et on déclare la classe `Batiment abstract`. Même si elle est abstraite, on peut utiliser `getType()` dans `toString()`, car les classes filles doivent obligatoirement la redéfinir pour être concrètes.

```
public abstract class Batiment {
    protected final String adresse ;
    public Batiment(String adresse) {
        this.adresse=adresse;
    }
    public abstract String getType() ;

    public String toString() {
        return adresse+" "+getType();
    }
}
```

Q4.7 (2 points) Écrire une classe `Maison` sans attribut. Quand une maison reçoit un livrable, elle affiche un message avec son adresse (uniquement l'adresse, pas le type) et le numéro du livrable (uniquement le numéro).

```
public class Maison extends Batiment implements Receveur {
    public Maison(String adresse) {
        super(adresse);
    }
    public String getType() {
        return "habitation";
    }
    public boolean recevoir(Livrable liv) {
        if (liv==null)
            return false;
        System.out.println("Le livrable "+liv.getNumero()+" a été reçu à l'
            adresse "+adresse); // OK car adresse protected
        return true;
    }
}
```

Dépôts avec casiers Un dépôt contient des casiers. Chaque casier peut contenir un seul livrable.

Q4.8 (1 point) Un casier (défini question suivante) peut être vide, plein ou trop petit, ce qui dans certains cas peut poser un problème. Écrire une classe `CasierException` contenant un constructeur prenant en paramètre un casier et une information sur le type de problème (par exemple, "vide", "plein", "trop petit"). Le message de l'exception doit être, par exemple, `"«C» est vide"`, `"«C» est plein"`, `"«C» est trop petit"` où `«C»` doit être remplacé par le `toString()` du casier.

```
public class CasierException extends Exception {
    public CasierException(Casier c, String information) {
        super(c+" est "+information);
    }
}
```

Q4.9 (3 points) Un casier a un numéro (`int`), une dimension (`Dimension`) et peut contenir un livrable (`Livrable`). Écrire une classe `Casier` avec un constructeur prenant en paramètre le numéro du casier et les méthodes (la méthode `toString()` n'est pas demandée) :

- méthode `stocker` dont le but est de stocker un livrable dans le casier. Cette méthode prend en paramètre un livrable, ne retourne rien, mais lève l'exception `CasierException` quand le casier est plein (contient déjà un livrable) ou trop petit (si la longueur du casier est plus petite que la longueur du livrable ou si la largeur du casier est plus petite que la largeur du livrable),
- méthode `obtenir` dont le but est d'obtenir le livrable dans le casier. Cette méthode sans paramètre enlève le livrable du casier et le retourne. Elle lève l'exception `CasierException` quand le casier est vide.

```java
public class Casier {
    private int numero;
    private Dimension dimCasier;
    private Livrable liv;
    public Casier(int numero) {
        this.numero=numero;
        dimCasier=new Dimension();
        liv=null;
    }
    public void stocker(Livrable lx) throws CasierException {
        if (liv!=null) {
            throw new CasierException(this,"plein");
        }
        Dimension dimLx=lx.getDimension();
        if(dimLx.longueur>=dimCasier.longueur
           ||dimLx.largeur>=dimCasier.largeur) {
            throw new CasierException(this,"trop petit");
        }
        liv=lx;
    }
    public Livrable obtenir() throws CasierException {
        if (liv==null) {
            throw new CasierException(this,"vide");
        }
        Livrable lx=liv;
        liv=null;
        return lx;
    }
    public String toString() { // pas demandée
        String s="";
        if(liv==null) s+=" (vide)"; else s+=" contient "+liv;
        return "Casier No"+numero+" "+dimCasier+s;
    }

}
```

Q4.10 (4 points) Un dépôt est un bâtiment industriel qui peut recevoir et livrer des livrables, ces livrables sont stockés dans des casiers. Écrire une classe `Depot` avec :

- un (seul) attribut de type `ArrayList` de casiers,
- un constructeur prenant en paramètre une adresse et le nombre de casiers que doit contenir le dépôt au départ,
- une méthode `recevoir` qui stocke si possible le livrable dans le premier casier qui accepte le livrable. Pour cet examen, il ne vous ait pas demandé d'écrire la méthode `livrer`.
- On veut aussi écrire une méthode qui permet de charger plusieurs livrables dans un `Receveur` (par exemple, pour charger un camion). Écrire une méthode `charger(Receveur r)` qui parcourt tous les casiers et donne au receveur les livrables des casiers tant que le receveur accepte des livrables : si le receveur refuse un livrable, alors le livrable est remis dans le casier et la méthode s'arrête.

```java
impor java.util.ArrayList; // pas demandé
public class Depot extends Batiment implements Receveur, Livreur {
    private ArrayList<Casier> alCasiers=new ArrayList<Casier>();
    public Depot(String adresse, int nbCasiers) {
        super(adresse);
        for(int i=0;i<nbCasiers;i++) {
            alCasiers.add(new Casier(i+1));
        }
    }
    public String getType() {
        return "industriel";
    }
```

```
13      public boolean recevoir(Livrable liv) {
14          for(Casier c : alCasiers) {
15              try {
16                  c.stocker(liv);
17                  return true;
18              } catch(CasierException e) {
19                  System.out.println(e.getMessage()); // casier plein, trop petit
20              }
21          }
22          return false;
23      }
24      public void charger(Receveur r) {
25          Livrable liv=null;
26          for(Casier c : alCasiers) {
27              try {
28                  liv=c.obtenir();
29                  boolean accepte=r.recevoir(liv);
30                  if (! accepte) {
31                      c.stocker(liv);
32                      return;
33                  }
34              } catch (CasierException e) {
35                  System.out.println(e.getMessage()); // casier vide
36              }
37          }
38      }
39      public Livrable livrer() {... } // pas demandé
40  }
```

Q4.11 (2 points) Ajouter dans la classe `Camion` une méthode `livraison(ArrayList<Batiment> alBat)` qui contient en paramètre une liste de bâtiments que le camion doit tous visiter pour faire sa livraison. Le camion reçoit un livrable des bâtiments livreurs. Le camion livre un livrable aux bâtiments receveurs.

```
1   public void livraison(ArrayList<Batiment> alBat) {
2       for(Batiment bat : alBat) {
3           if (bat instanceof Livreur) {
4               Livreur l=(Livreur)bat;
5               Livrable lx=l.livrer();
6               recevoir(lx);
7           }
8           if (bat instanceof Receveur) {
9               ((Receveur)bat).recevoir(livrer());
10          }
11      }
```