



NOM Bloquet
 Prénom Romain
 Promo 2018
 Date 25/03/2016



2	0	0
---	---	---



BLOQUET Romain
L3 - 2015

Excellent Devoir!

MATIÈRE Java niveau 2.

Question de cours
7/25/10

2/35

Exercice 1.

- ① Le type de retour de la méthode `getContentPane()` est `JPanel` un conteneur hérité de la super classe `JFrame`.
- ② Elle décharge le programmeur du listener.
- ③ Le rôle du container nécessaire au bon déroulement du modèle événementiel de Swing est d'être le listener (écouteur).
- ④ La signature de la méthode de l'interface `ActionListener` est `ActionPerformed` public void ActionPerformed().
- ⑤ Il faut implémenter l'interface `MouseListener` et redéfinir les méthodes qui nous intéressent. `MouseListener`
- ⑥ lorsqu'un événement arrive, il est notifié. `odi`

2/3

Exercice 2.

- ⑥ La programmation concurrente consiste à partager des ressources par plusieurs thread, c'est donc le partage de ressource. (file d'exécution).
- ⑦ Pour créer un thread on peut hériter de la super classe `Thread`
 ex:

```
public class MyThread extends Thread {
    ...
}
```
- Ou alors implémenter l'interface `Runnable`.
 ex:

```
public class MyThread implements Runnable {
    ...
}
```

Le désavantage de la Première manière, c'est qu'on peut avoir besoin d'hériter d'une autre classe, hors l'héritage Multiple en Java n'est pas possible, c'est pour cela qu'on préfère implémenter l'interface Runnable.

Implémenter l'interface Runnable est pratique, car on peut faire un héritage de classe. Tout objet qui implémente l'interface Runnable peut être exécuté comme un thread.

⑧. extends Thread

```
public class MyThread extends Thread {  
    int compteur;  
    ⓧ  
    While (True)  
    {  
        System.out.println("Je suis processeur : " + compteur);  
    }  
}
```

// Constructeur ⓧ
public MyThread(int i) {
 this.compteur = i;
}

implements Runnable.

```
public class MyThread implements Runnable {  
    // même code qu'au dessus.  
    NOT  
}
```

9. La Synchronisation permet de créer un moniteur (Verrou), ce qui permet de sécuriser les méthodes et les attributs. C'est à dire que lorsque 'un thread utilise une partie de code, celle-ci est déverrouillée jusqu'à la fin de son utilisation, pour être enfin exécutée par un autre thread.

10. Le mot clé est synchronized.

11. Une classe définissant un élément doit hériter de la classe java.util.EventObject.

12. L'interface nécessaire à la réalisation du Java Beans est Serializable. Une fois qu'elle est implémentée le Savoyean est enregistrable. C'est une interface vide de marquage que l'on doit implémenter.

13.

14. Un Java bean doit avoir que des propriétés déclarées privées, des méthodes publiques, les interfaces de classe sont accessibles/modifiable qu'avec des méthodes faites pour (set() et get()).

15. La Méthode Serializable sert à charger la valeur d'un objet. elle appartient à la classe java.util.EventObject.

16. Un objet non mutable est thread safe
pas besoin de copie défensive.
pas besoin de redéfinir la méthode clone().
Non modifiable après instantiation.

17. Malgré que la classe soit final et l'attribut aussi, this.name prend name en entrée et on renvoie name en sortie. Or il faut faire avec de la copie défensive dans les set() et get() pour pas que l'objet soit mutable.

18.

```
package mutable;  
public final class Cat {  
    private final StringBuilder name;  
    public Cat (StringBuilder name) {  
        this.name = new StringBuilder(name);  
    }  
  
    public StringBuilder getName () {  
        return new StringBuilder (name);  
    }  
}
```

for
/

L3 / DE JAVA 2015-2016



DE JAVA Niveau 2

Aucun document autorisé, durée 1 heure 45 minutes

Questions de cours :

10 points

I Programmation graphique avec les classes Swing (3.5 points)

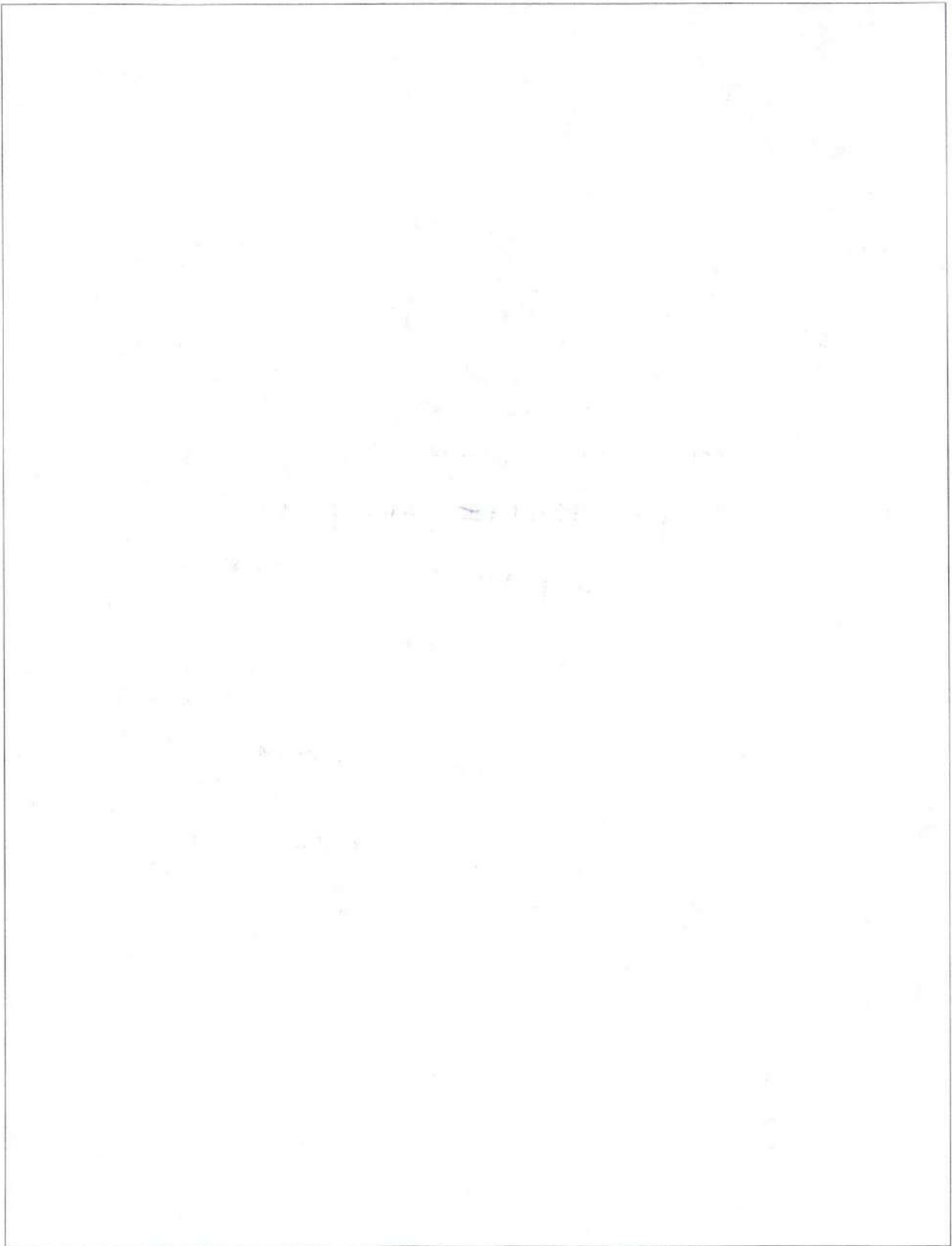
- 1) Quel est le type de retour de la méthode *getContentPane* et quel objet retourne-t-elle?
- 2) Le modèle événementiel des classes Swing décharge le programmeur de certaines tâches. Lesquelles?
- 3) Expliquer le rôle du container nécessaire au bon déroulement du modèle événementiel des classes Swing.
- 4) Donner la signature de la méthode de l'interface *ActionListener*.
- 5) L'interface *MouseListener* contient beaucoup de méthodes. Que faut-il mettre en place afin qu'une application graphique n'ayant besoin que de quelques unes de ces méthodes n'ait pas toutes à les définir?

II Programmation concurrente avec les threads (3 points)

- 6) Expliquer très succinctement en quoi consiste la programmation concurrente.
- 7) Quelles sont les deux manières de créer un thread ? Quel inconvénient peut présenter l'une des deux manières et quels sont les avantages de l'autre?
- 8) Pour chacune des deux manières, donner un exemple d'instanciation d'un thread.
- 9) A quelle problématique correspond la synchronisation d'une application concurrente?
- 10) Quel mot-clé assure cette synchronisation? Quel est l'effet du mot-clé *volatile*?

III Modèle des JavaBeans (2 points)

- 11) De quelle classe doit hériter une classe définissant un événement?
- 12) Quelle est l'interface nécessaire à la sérialisation des JavaBeans ? Quelle est sa particularité?
- 13) Quelles sont les signatures des deux méthodes d'abonnement et de désabonnement d'un JavaBean? A quoi servent ces deux méthodes?
- 14) Citer au moins trois types de propriétés d'un JavaBean.
- 15) A quoi sert la méthode suivante et à quelle classe appartient-elle?
`public void firePropertyChange(String propertyName, Object oldValue, Object newValue);`



7) On souhaite qu'il y ait une réduction (de 20) si tous les cadeaux d'un colis sont des éditions spéciales.

Le code suivant affichera donc :

```
System.out.println("total price " + packet3.totalPrice()); // total price 390
```

Modifier la méthode totalPrice en conséquence.

```
public int Total Price ( ) {
    int prix total = 0;
    bool bool = True;
    for (int i : paquet) {
        prix total += paquet . price ();
        if ( bool paquet . edition == false ) {
            bool = false;
        }
    }
    if ( bool == false ) { return prix total; }
    if ( bool == True ) { return ( prix total * 100 / 20 ); }
}
}
```

$\frac{2}{2}$

1225 / 10 points

Exercice

On cherche à modéliser des colis pouvant contenir des cadeaux comme des briques légos ou des robots araignées. Puis dans un second temps, des colis qui peuvent eux-mêmes contenir d'autres colis.

Le but principal est de pouvoir calculer le prix d'un colis en fonction des cadeaux qu'il contient et des réductions possibles.

Toutes les classes à écrire devront être déclarées dans le package *colis*.

1) **Ecrire une classe *LegoBricks* qui modélise un ensemble de briques LEGO.**

Il existe deux versions de ces briques, la version classique et la version "special edition" qui possède des briques en or (au moins de couleur jaune).

Le prix d'un ensemble de briques LEGO est calculé en multipliant par 10 le nombre de briques puis en ajoutant 10 si les briques sont une édition spéciale.

Ecrire la classe *LegoBricks*, son constructeur qui prend en paramètre un booléen indiquant si les briques sont une édition spéciale et le nombre de briques ainsi qu'une méthode *price* qui renvoie le prix sous forme d'un entier de telle sorte à ce que le code suivant fonctionne

```
LegoBricks bricks = new LegoBricks(true, 30);
System.out.println(bricks.price()); // 310
```

Classe *LegoBricks* avec un constructeur et une méthode *price*
 package *colis*;

```
public class LegoBricks {
    private boolean edition;
    private int nbpieces;
    private int prix;
    public LegoBricks (boolean edition, int nb) {
        this.edition = edition;
        this.nbpieces = nb;
        this.prix = (nb * 10);
        if (edition == true) this.prix =
    }
}
```



6) Si vous ne l'avez pas déjà fait, faite en sorte que le code affichant si un ensemble de briques LEGO ou une araignée robot est une édition spéciale soit partagé (qu'il n'y ait qu'une seule occurrence de ce code).

Le code suivant devra donc afficher :

```
Packet packet3 = new Packet();
SpiderRobot spiderRobot2 = new SpiderRobot(true, 20);
packet3.add(bricks);
packet3.add(spiderRobot2);
System.out.println(packet3);    //LEGO 30 bricks (special edition)
                                //spider robot 20 kg (special edition)
```

Indiquer les modifications éventuelles dans les classes existantes.

Expliquer votre raisonnement en quelques lignes.

Il faut synchroniser les méthodes d'affichage car une Arraylist n'est pas thread safe. Il faut donc placer Synchronized au début de chaque méthode.

~~public Synchronized...~~

HORS SUJET!

0/2

L3 / DE JAVA 2015-2016

```

public int price ( ) {
    int p ;
    p = nbpieces * 10 ;
    if ( edition == True ) { p + 10 ; }
}
return p ;

```

3/3

2) Ajouter une méthode dans la classe `LegoBricks` de sorte que l'on puisse afficher les briques LEGO avec le code suivant :

```
System.out.println(bricks); // LEGO 30 bricks (special edition)
```

L'affichage doit indiquer le nombre de briques ainsi que la mention "special edition" si les briques sont une édition spéciale et respectant le format de l'exemple ci-dessus.

Méthode de la classe `LegoBricks` pour pouvoir afficher les briques LEGO comme ci-dessus

```

package eding
public class @Override
    public String toString ( ) {
        if ( edition == True ) {
            return "LEGO " + nbpieces + " bricks (special edition)";
        }
        if ( edition == False ) {
            return "LEGO " + nbpieces + " bricks";
        }
    }
}

```

175/2



3) Ecrire la classe *Packet* qui modélise un colis contenant des briques LEGO.

La classe doit posséder une méthode *add* qui permet d'ajouter des ensembles de briques LEGO au colis, une méthode *numberOfItems* qui renvoie le nombre d'ensembles de briques LEGO dans le colis et une méthode *totalPrice* qui calcule le prix total d'un colis sachant que le prix total est calculé en faisant la somme du prix de chaque ensemble de briques LEGO. Le code suivant devra donc fonctionner :

```

LegoBricks bricks2 = new LegoBricks(false, 50);
Packet packet = new Packet();
packet.add(bricks);
packet.add(bricks2);
System.out.println(packet.numberOfItems() + " items"); // 2 items
System.out.println("total price " + packet.totalPrice()); // total price 810

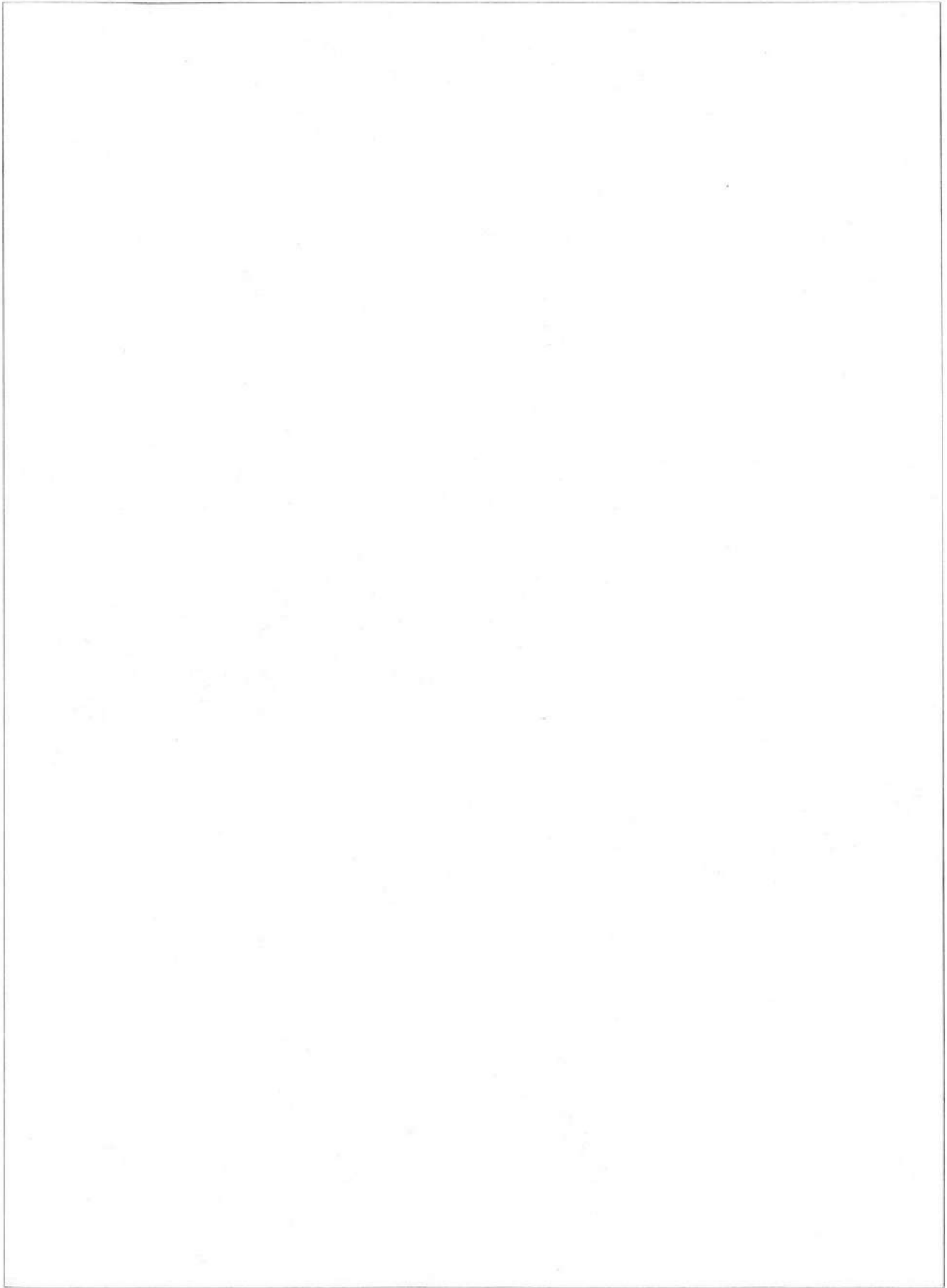
```

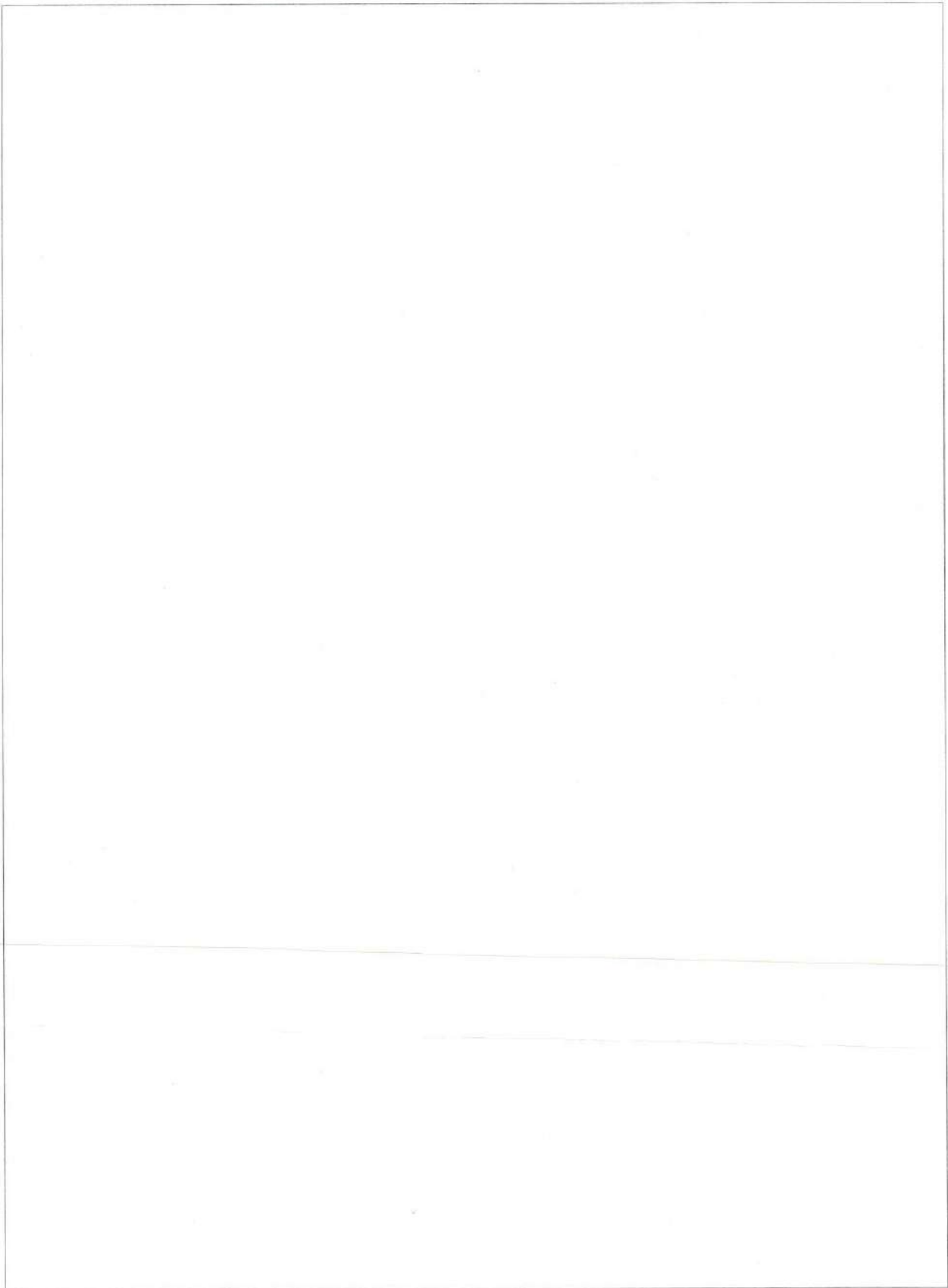
Classe *Packet* avec un constructeur, des méthodes *add*, *numberOfItems* et *totalPrice*

```

public class Packet {
    private int cpt = 0;
    private List<LegoBricks> paquet = new ArrayList<LegoBricks>();
    private List<LegoBricks> paquet = new ArrayList<LegoBricks>();
    public void add(LegoBricks brick) {
        this.paquet.add(brick);
        cpt++;
    }
}

```





L3 / DE JAVA 2015-2016

```
public int numberofItems () {  
    return this.cpt;  
}
```

```
public int totalPrice () {  
    int prixtotal = 0;  
    for (int prix : paquet) {  
        prixtotal += paquet.prix();  
    }  
    return prixtotal;  
}
```

}

3
3
3

4) Ajouter une méthode dans la classe *Packet* de sorte que le code suivant affiche tous les ensembles de briques LEGO :

```
System.out.println(packet); // LEGO 30 bricks (special edition)
                             // LEGO 50 bricks
```

Utiliser la classe *StringBuilder* afin que votre code n'alloue pas trop d'objets!

Utiliser une boucle "foreach" pour parcourir une collection d'objets.

Méthode permettant l'affichage de tous les ensembles de briques LEGO comme ci-dessus :

```
public StringBuilder Afficher () {
    for (String s : packets packets) {
        System.out.println(s);
    }
}
```

0/2



5) On souhaite pouvoir ajouter à un colis des robots araignées (SpiderRobot) ayant eux aussi une édition spéciale (l'araignée est alors dorée) ainsi qu'un poids en kilogramme. Le prix d'une araignée est alors calculé en multipliant son poids par 5. Le fait qu'une araignée soit une édition spéciale ne change pas son prix.

On veut donc que le code suivant fonctionne :

```
SpiderRobot spiderRobot = new SpiderRobot(false, 20); ✓
Packet packet2 = new Packet();
packet2.add(bricks);
packet2.add(spiderRobot);
System.out.println(packet2.numberOfItems() + " items"); // 2 items
System.out.println(packet2); // LEGO 30 bricks (special edition)
// spider robot 20 kg
System.out.println("total price " + packet2.totalPrice()); // total price 410
```

Note: il n'y a qu'une seule méthode *add* dans la classe *Packet*!

Ajouter les classes nécessaires et indiquer les modifications éventuelles des classes *LegoBricks* et *Packet*.

Expliquer votre raisonnement en quelques lignes.

```

public class Jouet {
    private boolean edition;
    private int prix;
}

// Constructeur?
public class SpiderRobot extends Jouet {
    private int poids;

    public SpiderRobot (boolean edition, int nb) {
        this.edition = edition;
        this.poids = nb;
    }
}
    
```

HERITAGE?

```
public int price () {
    int p;
    p = poids * 5;
    return p;
}
```

// Modif de la classe Packat.
 public class Packat {

```
private int cpt = 0;
private List paquet = new ArrayList ();
public void add (Object o) {
    this.paquet.add (o);
    cpt++;
}
```

accepte tout type.
pour
des points!

// les attributs de lego Bricks changent aussi car
 public class legoBricks extends Somet {

int nbpièces;
 // } après le reste est pareil.

Mal compris!!

25/4