

# INTRODUCTION A LA PROGRAMMATION

## Corrigé test semestre 1

### Exercice 1 : Concepts [33 points]

Répondez clairement et succinctement aux questions suivantes :

1. [10 points]

- (a) le programme affiche :

**false**

Justification : les classes **X** et **Y** ne redéfinissent pas la méthode **equals** héritée de **Object**. C'est donc cette dernière qui est utilisée et elle compare les références. Les objets **y1** et **y2** sont deux objets distinct et donc comparer leurs références retourne **false**.

- (b) Oui car en l'absence de cet appel c'est le constructeur par défaut de **X** (celui sans argument) qui est appelé.
- (c) Oui il s'agit bien d'une redéfinition (même nom, liste de paramètres identique et types de retour covariants/compatibles).
- (d) Non car : i) le type de retour ne fait pas partie de la signature et donc il y aurait ambiguïté entre les deux méthodes **m** de la classe **Y**, ii) il n'est pas possible de redéfinir une méthode avec un type de retour qui n'est pas covariant.
- (e) Non car les constructeurs de copie n'ont pas de version par défaut en Java. La ligne 21 du code ne compilerait plus.

2. [3 points]

- La ligne 8 est erronée car il n'existe pas de constructeur dans **X** prenant une **String** en argument.
- La ligne 11 est erronée car elle cause un appel récursif sans condition d'arrêt (le constructeur s'invoque lui-même).

3. [4 points]

- (a) 

```
public A(int val1, int val2) {  
    this(val1, val2, 10);  
}
```
- (b) 

```
public A(int val1) {  
    this(val1, 10);  
}
```

4. [3 points]

- (a) toute méthode définie dans une interface est nécessairement **public**. Une classe implémentant l'interface ne peut définir la méthode en question en restreignant ses droits d'accès (la méthode doit être explicitement déclarée comme **public** dans la classe). La méthode **draw** de **Ballon** n'est pas définie comme **public**.
- (b) ajouter le modificateur **public** à la méthode **draw** de la classe **Ballon**.

5. [3 points] Non car l'initialisation d'un attribut statique ne devrait pas dépendre de la construction d'un objet (sa valeur peut justement être utilisée indépendamment de la création de tout objet)

6. [3 points] Il s'agit d'erreurs qui généralement pourraient être corrigées par une meilleure programmation (débordement d'indices, division par zéro, accès aux membres d'un objet **null** etc.) et qui si elles se produisent, peuvent difficilement permettre à l'exécution de se poursuivre normalement. Java ne force donc pas le programmeur à les traiter en tant qu'exceptions.

---

7. [7 points]

- (a) parce que Java met en oeuvre la notion de portée de classe et non de portée d'instance (dans une classe tout membre, privé ou pas, peut-être accédé librement par toute instance de la classe, que ce soit `this` ou une autre instance de la classe)
- (b) La méthode `main` compile car : même si la classe `Disque` propose une surcharge (overload) et non une redéfinition (override) de la méthode `equals` héritée de `Object` (paramètre de type `Disque` au lieu de `Object`), la méthode héritée de `Object` reste bien sûr disponible. Ainsi le premier affichage de la méthode `main` appelle la surcharge de `equals` alors que la seconde appelle la méthode `equals` de `Object` (qui compare les références). Le programme affiche donc :

```
true
false
```

---

Exercice 2 : Conception OO et programmation [50 points + 15 bonus]

Il existe bien sûr de nombreuses variantes acceptables. (Pour une variante possible, voir le fichier source `Petri.java`)

Exercice 3 : Analyse de programme OO [11 points]

Paieement de bonus

Solution possible : on peut ajouter une méthode abstraite `bool deserveBonus()` retournant `false` pour un employé quelconque et `true` dans les sous-classes `technicien` et `développeur`. Il suffit alors d'itérer sur le tableau et de ne verser le bonus qu'à ceux pour lesquels la méthode `desrveBonus` retourne `true`.

Partage d'employés

Le constructeur met dans l'attribut `staff` la référence d'un tableau qui lui est passé depuis l'extérieur. Toute modification faite sur ce tableau à l'extérieur de la classe se répercute sur `staff`. De plus, chaque entrée du tableau est elle même une référence qui peut potentiellement être modifiée depuis l'extérieur (puisque les instances d'employés son mutables).

Pour corriger ces problèmes, il faut mettre dans `staff` la référence à une copie du tableau (créer un nouveau tableau en y copiant les entrées du tableau original, élément par élément). La copie devra être une copie profonde (chaque instance d'employé du tableau devra être elle-même copiée).

Le fait de devoir faire une copie d'employés nécessite de définir une méthode de copie polymorphique (`clone` ou autre) dans la hiérarchie d'employés.

Source de dépenses

Si l'on ne veut "cacher" que les sources de dépenses, il suffit de définir une interface `Depense` implémentée par `Employe` et `Materiel` et qui contient une méthode `montant`. Cette dernière retournerait le salaire de l'employé ou le prix d'achat du matériel par exemple.

La signature de la méthode `estimationGain` serait alors :

```
double estimationGain(Logiciel[] s, Depense[] d)
```

et l'on peut mettre dans le tableau `d` aussi bien un employé qu'un composant matériel.

## Exercice 4 : Exceptions [22 points]

- (a) la ligne 9 lancera une `InputMismatchException` qui n'est rattrapée nulle part et le programme s'arrête.  
(b) le ligne 21 lancera une `ArithmeticException` qui n'est rattrapée nulle part et le programme s'arrête.

2. Une solution possible :

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.InputMismatchException;

class MyReader
{
    private final static int MAX_NON_INT = 10;
    private final static Scanner clavier = new Scanner(System.in);

    public static int readInt() {
        int count = 1;
        do {
            try {
                int lu;
                System.out.println("Donnez un entier");
                lu = clavier.nextInt();
                return lu;
            }
            catch (InputMismatchException e){
                if (count < MAX_NON_INT) {
                    ++count;
                    clavier.nextLine();
                }
                else throw new InputMismatchException("Vous avez
                    saisi n'importe quoi trop de fois !");
            }
        }
        while (count <= MAX_NON_INT);
        return 0;
    }
}

class Container
{
    private final static int MAX_ZERO = 5;

    private ArrayList<Integer> collection;

    public Container() {
```

```
        collection = new ArrayList<Integer>();
    }

    public void read(int aSize) {
        int count = 1;
        for (int i = 0; i < aSize; ++i) {
            int value = MyReader.readInt();
            if (value == 0) {
                count++;
                if (count > MAX_ZERO)
                    throw new InputMismatchException("Remplissage
                        du tableau : trop de zéros");
            }
            else
                collection.add(1/value);
        }
    }
}

class ExceptionsCorr
{
    public static void main(String[] args) {
        Container myContainer = new Container();
        try {
            myContainer.read(20);
        }
        catch (InputMismatchException e){
            System.out.println(e.getMessage());
        }
    }
}
```