Corrigé de Examen Programmation par les Objets en Java
1A Juin 2017

NB. Certaines corrections sont détaillées plus qu'il n'a été demandé.

Exercice 1) Un fichier `Salut.java` contient le texte suivant

```
class Salut {
    static public void main(String args[]) {
        System.out.println("Bonjour");
    }
};
```

Lesquelles des lignes commandes suivantes sont incorrectes et pourquoi ?

1. `javac Salut.java`
2. `Java Salut.java`
3. `Javac Salut.class`
4. `java Salut.class`
5. `java Salut`

ligne 2. Java est une commande pour exécuter, et .java est un fichier source

ligne 3. Inversement, javac est une commande de compilation et .class est un bytecode

ligne 4. Pas besoin d'indiquer l'extension .class

Exercice 2)

Soit une interface Java *I*, et deux classes *C*1 et *C*2 qui l'implémentent. Les quelles des déclarations suivantes sont justes ou fausses ? Pourquoi ?

1. `I x = new I() ;` **faux, on n'instancie pas une interface**
2. `C1 y = new C1() ;` **juste, déclaration suivie d'instanciation par une classe qui implémente l'interface**
3. `I[] z = {new C1(), new C2()} ;` **idem, même si deux classes différentes. Elles implémentent la même interface**
4. `C1 w = new C2() ;` **faux, on déclare une classe et on instancie avec une autre (types incompatibles)**

Quel est l'intérêt de déclarer une interface pour ensuite l'implémenter avec des classes ? Ne peut-on déclarer directement des classes sans passer par une interface ?

L'intérêt est double :

1- Pouvoir choisir entre plusieurs implémentations possibles pour une MEME interface.

2- Une même variable de type Interface peut recevoir des instances de classes différentes.

Evidement, on peut toujours créer des classes sans utiliser des interfaces.

Exercice 3 : Le fichier source Java qui suit, déclare deux classes, une classe *Vehicule* qui porte une méthode *moi*() qui imprime un message, et une sous classe *Vehicule4x4* qui porte la même méthode. Une classe *GM* ensuite qui contient qui contient deux méthodes *demarrer*(*Vehicule*) et *demarrer*(*Vehucule4x4*) .

```
class Vehicule {
    void moi() {
        System.out.println ("J'ai 4 roues");
    }
}

class Vehicule4x4 extends Vehicule {
    void moi() {
        System.out.println ("J'ai 4 roues motrices");
    }
}

class GM {
    public static void demarrer(Vehicule v) {
        System.out.println (" Un vehicule démarre");
        v.moi();
    }
    public static void demarrer(Vehicule4x4 v) {
        System.out.println (" Un 4x4 démarre");
        v.moi();
    }
}

class EssaiVehicule  {
    static public  void main(String args[]) {
        GM.demarrer(new Vehicule());
        GM.demarrer(new Vehicule4x4());
        Vehicule x = new Vehicule4x4();
        GM.demarrer(x);    }
}
```

donne comme résultats :

```
 Un vehicule démarre            (1)
J'ai 4 roues                    (2)

 Un 4x4 démarre                 (3)
J'ai 4 roues motrices           (4)

 Un vehicule démarre            (5)
J'ai 4 roues motrices           (6)
```

Commenter ces résultats, surtout pourquoi les résultats (lignes (3) et (5)) sont différents, et les résultats (lignes 4 et 6) sont les mêmes.

Instruction `GM.demarrer(new Vehicule());` **C'est la *première* méthode de GM qui est appelée (appel résolu à la compilation d'après le profile du paramètre). Un *véhicule* est donc instancié, il démarre (1) et il a 4 roues (2). Normal.**

Instruction `GM.demarrer(new Vehicule4x4());` **C'est la *deuxième* méthode de GM qui est appelée (appel résolu à la compilation d'après le profile du paramètre). Un *4x4* est donc instancié, il démarre (3) et il a 4 roues motrices (4). Normal.**

Dans l'instruction `GM.demarrer(x);` **x est un *véhicule* (**`Vehicule x`**). C'est donc la *première* méthode de GM qui est appelée. C'est un *véhicule* donc qui démarre (ligne 5). Même si l'instance est un 4x4 (**`x=new Vehicule4x4()` **C'est la différence avec la ligne 3. Mais, comme l'instance présente est un 4x4 justement, il a 4 roues motrices (ligne 6 comme ligne 4). Car c'est le polymorphisme ici. Instruction** `v.moi()` **de GM.**

cf. TD Java, « Surcharge et Redéfinition » (dans Divers)

Exercice 4)

On voudrait définir un objet vecteur de réels de dimension donnée. Les opérations qu'on voudrait effectuer, entre autres, sont

- Créer un vecteur origine de dimension *n* donnée
- Affecter une valeur *x* au *i*ème composant, *i* et *x* données
- Accéder au *i*ème composant, donnée *i*, résultat *x*, ce *i*ème composant
- Remplir le vecteur avec la même valeur donnée *d*
- Produit scalaire de deux vecteurs.

1. Définir une Interface pour un tel objet vecteur.

```
interface Vecteur {

     public void origine(int n);
          // Crée un vecteur origine de dimension n
     public void set(int i, double x) ;
          // met ieme composant à x (0≤i)
     public double get (int i);
          // retourne le ieme composant
     public void setConst (int d);
          // met tous les composants à d
     public double produitScalaire (Vecteur r);
     // produit scalaire de deux vecteurs (this et r)
}
```

2. Implémenter cette interface vecteur

On négligera ici les cas d'erreurs : indice i hors des bornes, n négatif etc. Elles devront faire l'objet de levée d'exception.

```
class  Tableau  implements Vecteur {
     double [] t;

public void origine(int n) {
     t = new double [n] ;
     setConst(0);
}

public void set(int i, double x){
     t[i] = x;
}

public double get (int i){
     return t[i];
}

public void setConst (int d){
     for(int i = 0; i< t.length; i++)
          t[i] = d;
}
```

```
public double produitScalaire (Vecteur r){
     double ps = 0;
     for(int i = 0; i< t.length; i++) {
               ps += t[i]*r.get(i);
     }
     return ps;
     // Tester si même dimension
}
} ;
```

3. Ecrire un programme `main` qui crée les deux vecteurs de trois dimensions *v*1 (1, 3, 2) et *v*2 (4, 4, 4) et calcule leur produit scalaire.

```
class TestVecteur {
     public static void main(String [] args){

          Vecteur v1 = new Tableau(),
                  v2 = new Tableau();

          v1.origine(3);
          v1.set (0,1);
          v1.set (1,3);
          v1.set (2,2);

          v2.origine(3);
          v2.setConst(4);

          System.out.println(v1.produitScalaire(v2));
     }
}
```

Question subsidiaire :

Que faut-il changer dans l'interface vecteur pour avoir un vecteur d'entiers ? de caractères ? Proposer une idée pour généraliser la définition d'un vecteur, pour accepter des éléments de type quelconque.

Pour avoir un vecteur d'autre chose, `int` **ou** `char`**, on peut remplacer dans le fichier source le type** `double` **par le type voulu. Cela peut marcher pour des types primitifs connus du compilateur.**

Car il faut considérer les opérations = ou == ou + etc. effectuées sur ces types, et qui ne sont peut-être pas définies pour un type quelconque (classe utilisateur). En plus, ici, le produit scalaire n'a pas de sens pour le type `char`**.**

Cet aspect de la programmation objets, s'appelle la généricité. On dira classe générique. C'est une classe paramétrée par un type formel. Par exemple, ici le type des éléments du vecteur. Il faut alors s'assurer que les opérations effectuées par la classe sur ce type formel sont toutes offertes dans le type effectif correspondant