TD JAVA - Héritage, Polymorphisme, Transtypage (up & down).

Exercice 1 :

Soit `B` une sous-classe de `A`.

```
A a1 = new A();
A a2 = new A();
B b1 = new B();
a1 = a2;
b1 = a2;
a2 = b1;
```

Parmi les trois dernières instructions d'affectations, quelles sont celles qui sont correctes, quelles sont celles qui sont incorrectes, **Pourquoi ?**

Exercice 2 :

Soit une classe `Mother` et sa sous-classe `Daughter` définies comme suit :

```
public class Mother {
  ...
 public void methode(Mother m){...}
  ...
}
public class Daughter extends Mother {
  ...
  public void methode(Daughter d){...}
  ...
}
```

Soit l'extrait de code suivant :

```
Mother m1 = new Mother();
Mother m2 = new Mother();
Daughter d1 = new Daughter();
Daughter d2 = new Daughter();
m1.methode(m2);
m1.methode(d1);
d1.methode(m1);
d1.methode(d2);
```

Pour chacune des quatre dernières instructions, précisez la méthode qui sera invoquée (méthode de la classe `Mother` ou de la classe `Daughter`) et **expliquez**.

Exercice 3:

Soient les classes suivantes:

```
class EtreHumain {
  private int age;

  public EtreHumain(){
    age = 0;
  }

  public int getAge(){
    return age;
  }

  public void vieillir(){
    age++;
  }
}
abstract class HommePolitique extends EtreHumain{
  public abstract String debattre();
}

abstract class Chanteur extends EtreHumain{
  public abstract String chanter();
}
```

```
class Sarkozy extends HommePolitique{
  public String debattre(){
    return "Blablabla de droite";
  }
}

class Hollande extends HommePolitique{
  public String debattre(){
    return "Blablabla de gauche";
  }
}

class Sanseverino extends Chanteur{
  public String chanter(){
    return "La cigarette...";
  }
}

class Shakira extends Chanteur{
  public String chanter(){
    return "Bêêêêêê...";
  }
}
```

Question 1:

Dans chacun des cas suivants, **vous devez expliquer** ce que fait le code quand il est correct et s'il ne l'est pas, **vous devez expliquer pourquoi**.

Cas 1 :

```
Sanseverino s1 = new Sanseverino();
System.out.println(s1.chanter());
s1.vieillir();
```

Cas 2 :

```
Shakira m1 = new EtreHumain();
m1.vieillir() ;
```

Cas 3 :

```
HommePolitique hp1 = new Sarkozy();
System.out.println(hp1.chanter());
```

Cas 4 :

```
EtreHumain eh1 = new Sanseverino();
System.out.println(eh1.chanter());
```

Cas 5 :

```
EtreHumain eh1 = new Sanseverino();
System.out.println(((Chanteur)eh1).chanter());
```

Cas 6 :

```
EtreHumain eh1;
Chanteur c1;
Shakira m1 = new Shakira();
Sanseverino s1 = new Sanseverino();
c1 = m1;
System.out.println(c1.chanter());
c1 = s1;
System.out.println(c1.chanter());
eh1 = c1;
System.out.println(eh1.chanter());
```

Question 2:

Ajoutez une méthode `toString()` à la classe `EtreHumain` pour que l'affichage d'un objet `EtreHumain` soit : *« Je suis un être humain âgé de ? années »*.

Question 3 :

Redéfinissez la méthode `toString()` au niveau des classes `HommePolitique` et `Chanteur` pour que l'affichage soit respectivement : *« Je suis un être humain âgé de ? années et je peux debattre XXX »* ; *« Je suis un être humain âgé de ? années et je peux chanter XXX »*.

Exercice 4 : Location de biens

On suppose définie la classe `Personne` du paquetage `location` dont voici le diagramme UML :

Personne
- nom : String - age : int - salaire : int - permis : boolean
+ Personne(nom : String, age : int, salaire : int, permis : boolean) + getNom() : String + getAge() : int + getSalaire() : int + hasPermis() : boolean

L'interface `Louable` permet de définir des biens que l'on peut louer. En voici le code :

```
package location;
/**
* Cette interface identifie les Objets pouvant être loués. Ces objets doivent
* fournir un prix de location et proposer une condition à satisfaire pour une
* personne pour être loué
*/
public interface Louable {
   /** retourne Le prix de location
   * @return Le prix de location
   */
   public float prixLocation();
   /** retourne vrai si l'objet peut être loué par la personne
   * @param loueur la personne qui veut louer l'objet
   * @return true si cet objet peut être loué par la personne, false sinon
   */
   public boolean peutEtreLoue(Person loueur);
}
```

On s'intéresse à deux sortes de biens louables : des voitures et des chambres d'hôtel.

- les **voitures** ont un prix de location variable fixé à la construction de l'objet `Voiture`. Une voiture peut être louée si le locataire a le permis de conduire, et le prix n'excède pas 50% du salaire
- les **chambres d'hôtel** ont un numéro (un entier) et un prix de location fixés à la construction et peuvent être louées si le locataire a plus de 18 ans, a des revenus supérieurs à 1000 (euros) et le prix n'excède pas 20% de son salaire.

Q 1. Donnez le code java du type `Chambre`, Donnez le code java du type `Voiture`.

Q 2. On considère l'expression suivante : `Type uneVoiture = new Voiture(73);`
Quels sont tous les types autorisés possibles pour *Type* ?

Q 3. Donnez le code d'une méthode `filtreLouable` d'une classe `LouableManager` qui prend en paramètre une liste de biens louables `l`, une personne `p` et qui retourne la liste des biens de `l` qui peuvent être loués par `p`.

Exercice 5 : Mon super jeu de dé et de stratégie

On souhaite simuler un jeu de dé entre 4 joueurs. Les joueurs jouent à tour de rôle. A chaque tour, un joueur avance ou recule en fonction du score qu'il réalise. Un score positif implique un déplacement vers l'avant, un score négatif implique un déplacement vers l'arrière. On appelle position du joueur le cumul des scores réalisés. Le premier joueur qui atteint la position 500 a gagné la partie et le jeu est terminé.

Le score d'un joueur à chaque tour est déterminé par

1) Le jet d'un dé
2) La position relative du joueur par rapport aux autres joueurs
3) La stratégie associée au joueur. Il y a trois types de stratégies possibles. Chaque joueur est associé à une des stratégies. Les joueurs peuvent avoir la même stratégie. De temps à autre, le jeu attribue une nouvelle stratégie au joueur. (ca peut être la même)

C'est la stratégie associée au joueur qui détermine comment le résultat du dé et la position du joueur par rapport aux autres influent sur le calcul du score. Les trois types de stratégies sont décrits ci-dessous

Stratégie Type 1

La valeur du déplacement est obtenue par le calcul suivant :
Valeur du dé + (position du Premier – Position du joueur) /2
Si le dé à pour valeur 3, 4, 5 ou 6 le déplacement est positif sinon il est négatif.

Stratégie Type 2

La valeur du déplacement est obtenue par le calcul suivant :
3 fois la valeur du dé si cette valeur est paire, une fois la valeur du dé si celle ci est impaire.
Ce changement de position est toujours positif

Stratégie Type 3

La valeur du déplacement est obtenue par le calcul suivant :
Valeur du dé + (position du joueur – Position du dernier) / 2
Si le dé à pour valeur 1 ou 2 le déplacement est positif sinon il est négatif

Une stratégie est attribuée aléatoirement au début du jeu à chaque joueur pour une durée aléatoire (entre 2 et 5 tours). A l'issue de cette durée le jeu réattribue une des trois stratégies au joueur pour une durée aléatoire (entre 2 et 5 tours)

Après chaque tour de jeu une ligne est affichée sur la console pour préciser le numéro du joueur, et sa position. A chaque changement de stratégie une ligne est affichée précisant la stratégie attribuée et la durée. Quand le jeu est terminé une ligne est affichée indiquant le joueur gagnant

```
// Exemple de déroulement du jeux
J'affecte au Joueur 1, la stratégie 'class StrategieTypeTrois', pour une durée de 5
J'affecte au Joueur 2, la stratégie 'class StrategieTypeUne', pour une durée de 3
J'affecte au Joueur 3, la stratégie 'class StrategieTypeDeux', pour une durée de 2
J'affecte au Joueur 4, la stratégie 'class StrategieTypeDeux', pour une durée de 2
Joueur 1 position: -6
Joueur 2 position: -1
Joueur 3 position: 3
Joueur 4 position: 1
Joueur 1 position: -5
Joueur 2 position: 6
Joueur 3 position: 4
Joueur 4 position: 4
Joueur 1 position: -11
Joueur 2 position: 11
J'affecte au Joueur 3, la stratégie 'class StrategieTypeTrois', pour une durée de 4
Joueur 3 position: -6
J'affecte au Joueur 4, la stratégie 'class StrategieTypeDeux', pour une durée de 4
Joueur 4 position: 22
Joueur 1 position: -16
J'affecte au Joueur 2, la stratégie 'class StrategieTypeDeux', pour une durée de 2
Joueur 2 position: 16
Joueur 3 position: 1
Joueur 4 position: 34
Joueur 1 position: -15
..............
J'affecte au Joueur 4, la stratégie 'class StrategieTypeUne', pour une durée de 4
Joueur 4 position: 261
J'affecte au Joueur 1, la stratégie 'class StrategieTypeDeux', pour une durée de 3
Joueur 1 position: -25
Joueur 2 position: 483
Joueur 3 position: 319
Joueur 4 position: 149
Joueur 1 position: -20
J'affecte au Joueur 2, la stratégie 'class StrategieTypeTrois', pour une durée de 2
Joueur 2 position: 736
      Joueur 2 gagne le jeu.
```

Exercice 6 : Simulation d'afficheurs lumineux

Le but de cet exercice est de simuler en Java les afficheurs lumineux qu'on voit un peu partout et qui font circuler un texte en boucle.

1 Le décaleur

Intéressons-nous d'abord au décaleur. C'est un objet qui stocke une suite de L caractères avec L > 0 (et pas plus). L est constant et est appelé largeur du décaleur. Initialement, un décaleur contient L espaces.

Les quatre fonctionnalités d'un décaleur sont :

- `getLargeur` renvoie la largeur du décaleur,
- `raz` force tous les caractères à espace,
- `decale` décalage d'une position vers la gauche de sa suite de caractères : le caractère le plus à gauche est supprimé de la suite et est renvoyé par la méthode. Le nouveau caractère le plus à droite est fixé à la valeur du paramètre de la méthode.
- `toString` renvoie sous forme de `String` une copie du contenu de la suite de caractères du décaleur.

Q 1. Définir la classe `Decaleur`.

Information

La classe prédéfinie `String` possède le constructeur `String(char[] t)` qui permet d'obtenir une instance de `String` ayant le même contenu que le tableau de caractères `t`.

2 Les afficheurs lumineux

Les caractéristiques d'un afficheur lumineux sont les suivantes : il ne peut visualiser simultanément qu'un nombre N fixe et entier de caractères avec N > 0. Le message qui se déroule en boucle sur l'afficheur ne doit pas avoir une longueur nulle, en revanche, celle-ci peut être plus petite, égale ou supérieure à N .

Le message défile dans l'afficheur en se décalant d'une position vers la gauche à chaque top d'une horloge. Quand le dernier caractère du message vient juste d'entrer dans la partie visualisée, au prochain top horloge, c'est le premier caractère du message qui y entre à son tour.

Le code de la classe `Afficheur` pourrait ressembler à :

```
public class Afficheur {
...
  // fixe un nouveau message a afficher
  public void setMessage(char[] message)
  // un top d'horloge
  public void top() {...}
  // renvoie ce qui doit etre affiche
  public String toString(){...}
}
```

Pour fixer les idées, soit la classe ci-dessous, l'invocation `new Test().tester(new Afficheur(6)) ;` produit alors la trace de droite.

```
public class Test {

   public void tester(Afficheur afficheur) {
      char[] message = { 'D', 'e', 's', 'p', 'r', 'é' , 's'};
      afficheur.setMessage(message);
      for (int i = 0; i < 10; i++) {
         afficheur.top();
         System.out.println("<<" + afficheur + ">>");
      }
   }
}
```

```
<<      D>>
<<     De>>
<<    Des>>
<<   Desp>>
<<  Despr>>
<<Despré>>
<<esprés>>
<<sprésD>>
<<présDe>>
<<résDes>>
```

Q 2. Complétez le code de la classe `Afficheur`.

3 Les afficheurs avec latence

On remarque que pour les afficheurs de l'exercice précédent il est difficile de voir où se termine le message. Pour éviter ce problème, on veut une nouvelle classe d'afficheurs pour lesquels on pourra spécifier lors de leur création un "temps de latence" entre l'entrée du dernier et celle du premier caractère. Ce temps de latence sera exprimé par un nombre positif ou nul d'espaces à insérer entre ces deux caractères.

Appelons `Latence` cette nouvelle classe d'afficheurs.

Avec l'invocation `new Test().tester(new Latence(6,3));` on crée un afficheur avec une latence de trois espaces et on le teste :	`<<  D>>`
	`<<  De>>`
	`<<  Des>>`
	`<<  Desp>>`
	`<< Despr>>`
	`<<Despré>>`
	`<<esprés>>`
	`<<sprés >>`
	`<<prés  >>`
	`<<rés  >>`

Q 3. Définissez la classe `Latence`.

4 Les afficheurs avec latence et vitesse paramétrable

A chaque top d'horloge, les afficheurs précédents font un seul décalage. On voudrait une nouvelle sorte d'afficheur dont on pourrait fixer le nombre de décalages effectués à chaque top. Ce nombre sera un entier positif ou nul.

Appelons `Vitesse` cette nouvelle classe d'afficheurs.

Q 4. Définissez la classe `Vitesse`.

Avec l'invocation `new Test().tester(new Vitesse(6,3,2));` on crée un afficheur avec une latence de trois espaces et une vitesse de 2 et on le teste :	`<<  De>>`
	`<<  Desp>>`
	`<<Despré>>`
	`<<sprés >>`
	`<<rés  >>`
	`<<s  De>>`
	`<<  Desp>>`
	`<<Despré>>`
	`<<sprés >>`
	`<<rés  >>`