8 Héritage et liaison dynamique

Remarque : les frontières des TD à partir de la séance 7 sont flous. On peut taper dans les différentes séances et revenir en arrière sans problème
Séance 8 :
Objectif : cast, equals standard, interface
41, 42, 43, (44) (exercice fait en cours), (45) (exercice fait en cours), 46, 47, 48 (possibilité de remettre les interfaces en semaine 9 si manque de temps)
Ne pas hésiter à faire le 39 dans cette séance si on a le temps.

Exercice 41 – Chien et Mammifère (Transtypage d'objet)

Rappel de cours : Le cast (conversion de type ou transtypage) consiste à forcer un changement de type si les types sont compatibles. Pour cela, il suffit de placer le type entre parenthèses devant l'expression à convertir.

Q 41.1 La méthode main suivante est-elle correcte ? Expliquez les erreurs.

```
1 public class Mammifere { ... }
2 public class Chien extends Mammifere {
3   public void aboyer() { System.out.println("Ouaff"); }
4   public static void main(String[] args) {
5     Chien c1 = new Chien();
6     Mammifere m1 = c1;
7     c1 = (Chien) m1;
8     c1 = m1;
9     Mammifere m2 = new Mammifere();
10    Chien c2 = (Chien) m2;
11  }
12 }
```

```
Erreur de compilation pour la ligne : c1=m1;
Chien.java:7: incompatible types:
     found  : Mammifere
     required: Chien
c1=m1; => m1, variable de la classe Chien, ne peut référencer un objet de la
super-classe Mammifère. Il faut faire un cast explicite.

Pas d'erreur à la compilation pour la ligne : Chien c2=(Chien)m2;
Par contre,  erreur à l'exécution :
     Exception in thread "main" java.lang.ClassCastException: Mammifere
     cannot be cast to Chien at Chien.main(Chien.java:9)

Atttention : ce n'est pas parce que le cast passe à la compilation,
qu'il n'y a pas une erreur.
```

Exercice 42 – Redéfinition de la méthode `equals`

Soit la classe `Point` ci-dessous :

```
1 public class Point {
```

```
2   private int x, y; // coordonnees
3   public Point(int a, int b) {x=a; y=b;}
4   public Point() {x=0; y=0;}
5   public Point (Point p) { x=p.x; y=p.y;}
6
7   public static void main(String [] args) {
8     Point p1 = new Point(5,2);
9     Point p2 = new Point(5,2);
10    Point p4 = new Point(1,1);
11    Point p3 = p1;
12    System.out.println("p1=p2 : "+ p1.equals(p2));
13    System.out.println("p1=p3 : "+ p1.equals(p3));
14    System.out.println("p1=p4 : "+ p1.equals(p4));
15  }
16 }
```

Q 42.1 Qu'affiche l'exécution du main ?

```
p1=p2 : false
p1=p3 : true
p1=p4 : false
```

Q 42.2 Rédéfinir la méthode `boolean equals(Object ob)` de la classe `Object` dans la classe `Point`, de façon qu'elle teste l'égalité des coordonnées et non des références. Les instructions de test sont fournies dans la méthode `main`.

Il faut bien penser à caster o en Point sinon o.x provoque une erreur à la compilation !

```
1 public boolean equals(Object o) {
2   Point p = (Point)o;
3   return (this.x == p.x) && (this.y == p.y);
4 }
```

Affiche alors :

```
p1=p2 : true
p1=p3 : true
p1=p4 : false
```

Q 42.3 Que se passe-t-il si dans la méthode `main`, on rajoute à la suite les instructions suivantes ? Comment résoudre le problème rencontré ?

```
1 String s1=new String("Bonjour");
2 System.out.println("p1=s1 : "+ p1.equals(s1));
```

Le programme affiche :

```
p1=p2 : true
p1=p3 : true
p1=p4 : false
Exception in thread "main" java.lang.ClassCastException: String cannot be cast to Point
        at Point.equals(TestMethodeEquals.java:8)
        at TestMethodeEquals.main(TestMethodeEquals.java:29)
```

Le problème est qu'on ne peut pas transtyper un String en un Point. Pour résoudre le problème, on doit d'abord vérifier que l'objet passé en paramètre est bien un objet de type Point, pour cela, on utilise l'opérateur `instanceof`.

```
1  public boolean equals(Object o) {
2     if (o instanceof Point) {
3        Point p = (Point)o;
4        return (this.x == p.x) && (this.y == p.y);
5     }
6     return false;
7  }
```

NB : la correction ci dessus est (un peu) fausse car si `o` est une instance d'une sous classe de Point, on peut répondre `true` alors que c'est faux... Bonne correction :

```
1  public boolean equals(Object o) {
2     if (getClass() == o.getClass()) {
3        Point p = (Point)o;
4        return (this.x == p.x) && (this.y == p.y);
5     }
6     return false;
7  }
```

Exercice 43 – Véhicules à moteurs

On considère un parc de véhicules. Chacun a un numéro d'identification (attribué automatiquement) et une distance parcourue (initialisée à 0). Parmi eux on distingue les véhicules à moteurs qui ont une capacité de réservoir et un niveau d'essence (initialisé à 0) et les véhicules sans moteur qui n'ont pas de caractéristique supplémentaire. Les vélos ont un nombre de vitesses, les voitures ont un nombre de places, et les camions ont un volume transporté.

Q 43.1 : Construire le graphe hiérarchique des classes décrites ci-dessus.

```
Vehicule(id, distParcourue)
  |
  |_______AMoteur (capacitéReservoir, niveauEssence)
             |_____Voiture (nbPlaces)
             |_____Camion (Volume)
  |
  |______ SansMoteur ()
             |_____Vélo (nbVitesses)
```

Q 43.2 Ecrire le code java des classes `Vehicule`, `AMoteur`, et `SansMoteur` avec tous les constructeurs nécessaires et les méthodes `toString()`.
Rappel de cours : Tout constructeur d'une sous-classe `a` implicitement comme première instruction un appel au constructeur sans paramètre de la super classe (s'il n'appelle pas lui-même un constructeur de la super classe explicitement).

```
1 public abstract class Vehicule {
2  private static int nbVehicules=0;
3  private int id;
4  private double distParcourue=0;
```

```
5   public Vehicule() {
6     nbVehicules++;
7     id=nbVehicules;
8   }
9   public String toString() {
10    return "" + id +"␣";
11  }
12 }
13 public abstract class AMoteur extends Vehicule {
14   private double capaciteReservoir;
15   private double niveauEssence=0;
16   public AMoteur(double capa) {
17     super();
18     capaciteReservoir=capa;
19   }
20   public String toString() {
21     return super.toString() + "\n␣Vehicule␣a␣moteur,␣reservoir␣:␣"  + capaciteReservoir
           +"␣litres.␣\n␣Niveau␣actuel␣a␣:␣" + niveauEssence + "␣litres";
22   }
23 }
24 public abstract class SansMoteur extends Vehicule {
25   public SansMoteur() {
26     super();
27   }
28   public String toString() {
29     return super.toString() + "␣\n␣Vehicule␣sans␣moteur";
30   }
31 }
```

Q 43.3 Ecrire une méthode `rouler(double distance)` qui fait avancer un véhicule. A quel niveau de la hiérarchie faut-il l'écrire ?

Il faut l'écrire dans la classe Vehicule.
Note : dans le `println()`, l'appel à `toString` est implicite.

```
1 public void rouler(double distance) {
2     distParcourue+=distance;
3     System.out.println("vehicule␣"+ this + "a␣fait"+ distance+ "km\n");
4 }
```

Q 43.4 Ecrire les méthodes `void approvisionner(double nbLitres)`, et `boolean enPanne()` (en panne s'il n'y a plus d'essence). A quel niveau de la hiérarchie faut-il les écrire ?

Il faut les écrire dans la classe `AMoteur` :

```
1   public void approvisionner(double nbLitres) {
2     if (niveauEssence+nbLitres > capaciteReservoir)
3        System.out.println("ca␣deborde..!");
4     else {
5        niveauEssence +=nbLitres;
6        System.out.println("ajoute␣"+ nbLitres + "␣litres␣dans␣" + this);
7     }
8   }
9   public boolean enPanne() {
```

```
10      if (niveauEssence==0) System.out.println("plus d'essence!\n");
11      return (niveauEssence==0);
12    }
```

Q 43.5 Ecrire la classe `Velo` avec consructeur et méthode `toString()` et une méthode `void transporter(String depart, String arrivee)` qui affiche par exemple “le vélo n°2 a roulé de Dijon à Châlon”.

```
1   public class Velo extends SansMoteur {
2     private int nbVitesses;
3     public Velo(int n) {
4       super();
5       nbVitesses=n;
6     }
7     public String toString() {
8       return " Velo "+super.toString();
9     }
10    public void transporter(String depart, String arrivee) {
11      System.out.println(this + " roule de "+ depart + " a "+ arrivee +"\n");
12    }
13  }
```

Q 43.6 Ecrire la classe `Voiture` avec constructeur et méthode `toString()` et une méthode `void transporter(int n, int km)` qui affiche par exemple ”la voiture n°3 a transporté 5 personnes sur 200 km” ou bien ”plus d’essence!” suivant les cas.

```
1 public class Voiture extends AMoteur {
2     private int nbPlaces;
3     public Voiture(double capa,int n) {
4       super(capa);
5       nbPlaces=n;
6     }
7     public String toString() {
8       return " Voiture "+super.toString();
9     }
10    public void transporter(int n, int km) {
11      if (enPanne()) {
12        return;
13      }
14      System.out.println(this + " transporte "+ n +" personnes sur "+ km + " km");
15    }
16  }
```

Q 43.7 : Ecrire la classe `Camion` avec constructeur, la méthode `toString()` et une méthode `void transporter(String materiau, int km)` qui affiche par exemple ”plus d’essence!” ou bien ”le camion n°4 a transporté des tuiles sur 500 km”.

```
1 public class Camion extends AMoteur {
2     private double volume;
3     public Camion(double capa,double vol) {
```

```
4       super(capa);
5       volume=vol;
6     }
7     public String toString() {
8       return "␣Camion␣"+super.toString();
9     }
10    public void transporter(String materiau, int km) {
11      if (enPanne()) {
12        return;
13      }
14      System.out.println(this + "transporte␣des" + materiau +"␣sur␣"+ km + "km");
15    }
16  }
```

Q 43.8 Peut-on factoriser la déclaration de la méthode transporter, et si oui, à quel niveau ?

Non, les signatures sont toutes différentes

Q 43.9 On considère le main ci-dessous. Ce programme est-il correct ? Le corriger si nécessaire. Qu'affiche-t-il ?

```
1 public static void main(String[] args) {
2   Vehicule v1=new Velo(17); // nb de vitesses
3   Vehicule v2=new Voiture(40.5,5); // capacite reservoir, nb de places
4   Vehicule v3=new Camion(100.0,100.0); // capacite reservoir, volume
5   System.out.println("Vehicules␣:␣"+v1+v2+v3);
6   System.out.println();
7   v2.approvisionner(35.0); // litres d'essence
8   v3.approvisionner(70.0);
9   System.out.println();
10  v1.transporter("Dijon","Valence");
11  v2.transporter(5,300);
12  v3.transporter("tuiles",1000);
13 }
```

Non, erreur de compilation. Pour pouvoir envoyer les messages approvisionner et transporter à un objet d'une sous-classe `de` Véhicule, il faut caster explicitement ce véhicule dans sa sous-classe réelle.
Cela donne, après correction :

```
1 public class MainVehicule {
2     public static void main(String[] args) {
3       Vehicule v1=new Velo(17); // nb de vitesses
4       Vehicule v2=new Voiture(40.5,5); // km,nb de Places
5       Vehicule v3=new Camion(100.0,100.0); // km,volume
6       System.out.println("Vehicules␣:␣"+v1+v2+v3);
7       System.out.println();
8       ((AMoteur)v2).approvisionner(35.0); // litres d'essence
9       ((AMoteur)v3).approvisionner(70.0);
10      System.out.println();
11      ((Velo)v1).transporter("Dijon","Valence");
12      ((Voiture)v2).transporter(5,300);
13      ((Camion)v3).transporter("tuiles",1000);
14    }
15  }
```

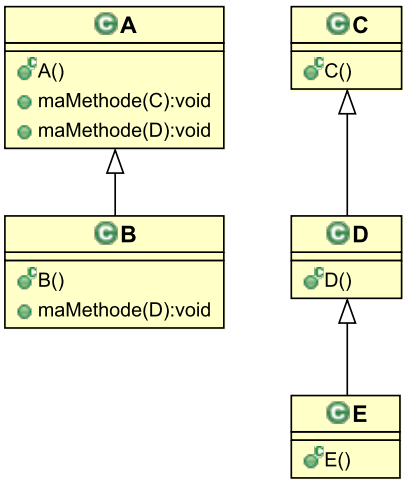
L'exécution donne :

```
Vehicules :  Velo 1  Voiture 2  Camion 3
ajoute 35.0 dans  Voiture 2
ca deborde..!
ajoute 70.0 dans  Camion 3
Velo 1  roule de Dijon a Valence
Voiture 2  transporte 5 personnes sur 300 km
Camion 3 transporte tuiles sur 1000km
Press any key to continue...
```

Exercice 44 – Sélection de méthode

Soit une hiérarchie de classes (figure ci-dessous).

Q 44.1 Pour chaque ligne de code appelant `maMethode`, dire quelle méthode est effectivement appelée.




```
A a = new A();
B b = new B();
A ab = new B();

C c = new C();
D d = new D();
E e = new E();
C cd = new D();
```

```
a.maMethode(c);
a.maMethode(d);
a.maMethode(cd);
a.maMethode((D) cd);
a.maMethode(e);

b.maMethode(c);
b.maMethode(d);
b.maMethode(cd);
b.maMethode((D) cd);
b.maMethode(e);

ab.maMethode(c);
ab.maMethode(d);
ab.maMethode(cd);
ab.maMethode((D) cd);
ab.maMethode(e);
```

Je numérote les 3 solutions de haut en bas
sur l'instance `a` :

- 1 : pas de piège
- 2 : pas de piège
- 1 : PIEGE : on est dynamique sur les instances qui appellent les méthodes mais pas sur les arguments ! `cd` est de type C : sélection de la méthode correspondante
- 2 : pas de piège (mais gare à l'exécution si on fait mal le cast)
- 2 : pas de signature exacte correspondant à l'appel ⇒ recherche de la signature compatible la plus proche (E est un D), D est plus proche de E (par rapport à C→E)

```
je suis dans A (arg C)
je suis dans A (arg D)
je suis dans A (arg C) // PIEGE: on est dynamique sur les instances qui appellent les methodes mais pas sur les arguments ! cd est de type C: selection de la methode correspondante
je suis dans A (arg D)
je suis dans A (arg D) // pas de signature exacte correspondant a l'appel ⇒ recherche de la signature compatible la plus proche (E est un D), D est plus proche de E (par rapport a C→E)

je suis dans A (arg C)
je suis dans B (arg D)
```

```
9 je suis dans A (arg C)
10 je suis dans B (arg D)
11 je suis dans B (arg D)
12
13 je suis dans A (arg C)
14 je suis dans B (arg D)
15 je suis dans A (arg C)
16 je suis dans B (arg D)
17 je suis dans B (arg D)
```

Même comportements sur les instances `b` et `ab`
Remarque sur les priorités : entre une signature exacte dans la classe mère et une signature approchée dans la classe de l'instance, on prend la signature exacte (l9 et l15)

Q 44.2 Conversions implicites (ou pas)

On envisage 3 ajouts de méthodes :

Cas 1 :

```
1 // dans A
2 public void meth(double d)
3 // rien dans B
```

Cas 2 :

```
1 // dans A
2 public void meth(int i)
3 // dans B
4 public void meth(double d)
```

Cas 3 :

```
1 // dans A
2 public void meth(int i)
3 // rien dans B
```

Le code à exécuter est maintenant le suivant :

```
1 a.meth(2);
2 a.meth(2.);
3 b.meth(2);
4 b.meth(2.);
5 ab.meth(2);
6 ab.meth(2.);
```

En envisageant les 3 cas ci-dessus :
- Quelles sont les lignes posant des problèmes de compilation ?
- Quelles sont les méthodes sélectionnées (pour le cas 2) ?

Cas 1 :
Tout est OK à la compilation + exécution : conversion implicite int⇒double

Cas 2 :

```
1 a.meth(2);   // OK 1
2 a.meth(2.);   // KO compil
3 b.meth(2); // OK 1
4 b.meth(2.); // OK 2
5 ab.meth(2); // OK 1
6 ab.meth(2.); // KO compil
```

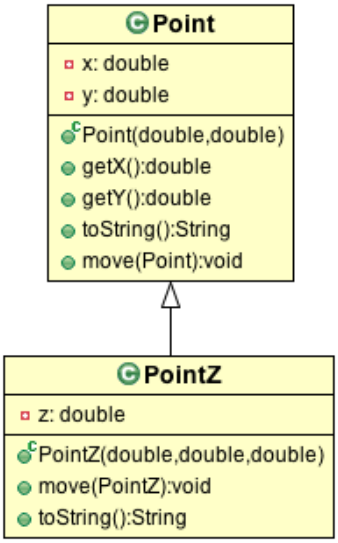
Pas de conversion double ⇒ int (trop dangereux)

Cas 3 :

```
1 a.meth(2);
2 a.meth(2.);   // KO compil
3 b.meth(2);
4 b.meth(2.); // KO compil
5 ab.meth(2);
6 ab.meth(2.); // KO compil
```

Exercice 45 – Redéfinition piégeuse

Soit la structure hiérarchique décrite dans le schéma UML ci-dessous :




```
1 public class Point {
2     private double x,y;
3     public Point(double x, double y) {
4         this.x = x;     this.y = y;
5     }
6     public double getX() { return x; }
7     public double getY() { return y; }
8     public String toString() {return "["+ x + "␣" + y +"]";}
9     public void move(Point p){ x+=p.x; y+=p.y; }
10 }
11 ///
12 public class PointZ extends Point {
13     private double z;
14     public PointZ(double x, double y, double z) {
15         super(x, y);
16         this.z = z;
17     }
18     public void move(PointZ p){
19         super.move(p);
20         z += p.z;
21     }
22     public String toString(){
23         return "["+getX()+"␣"+getY()+"␣"+z+"]";
24     }
25 }
```

Q 45.1 Pourquoi cette hiérarchie de classe est-elle discutable ?

Il faut savoir si un PointZ est une spécialisation (cas particulier) de Point ou l'inverse.
- Si on considère qu'il s'agit d'un cas particulier (c'est un Point qui a pour particularité d'avoir une composante en Z) ⇒ OK
  un PointZ **EST UN** Point
- Si on considère que PointZ = point 3D... Il y a un soucis. En effet, du point de vue logique, un Point2D EST UN Point3D et pas l'inverse...

Q 45.2 Syntaxe : les lignes 15, 19 et 23 sont-elles correctes ? Sinon, proposez des modifications. En ligne 23, peut-on utiliser directement `x` et `y` sans passer par les accesseurs ? Pourquoi ?

Le prog est correct.
Le super.get... en ligne 23 est évidemment optionnel.
On ne peut pas utiliser directement `x` et `y` car ce sont des attributs privés de la classe mère... Il faudrait qu'ils soient `protected`.

Q 45.3 Que pensez-vous du programme suivant ?

```
1 Point p = new Point(1,2);
2 Point p3d = new PointZ(1,2,3);
3 PointZ depl = new PointZ(1,1,1);  // deplacement a effectuer
4
5 System.out.println(p); // affichage avant modif
6 System.out.println(p3d);
7 p.move(depl); // modif
8 p3d.move(depl); // modif
9 System.out.println(p); // affichage apres modif
10 System.out.println(p3d);
```

Q 45.3.1 Qu'est-ce qui s'affiche ?

Q 45.3.2 Est-ce que ça vous semble logique ?

Q 45.3.3 Expliquer en détail ce qui s'est passé au niveau de la compilation et de l'exécution.

Affichage :

```
[1.0, 2.0]
[1.0 2.0 3.0]
[2.0, 3.0]
[2.0 3.0 3.0] // composante en Z non modifiee!!
```

1. Compilation : présélection sur le type des variables

   ```
   p3d.move(depl); // -> move(Point p)
   ```

2. Execution :
   (a) Recherche de `move(Point p)`
   (b) La méthode existe dans la super-classe, elle est exécutée
   (c) PointZ peut toujours remplacer un Point, il rentre dans la méthode

Exercice 46 – Interfaces véhicules

Nous souhaitons gérer une grande liste de véhicule à moteur, chacun d'eux ayant comme propriété de pouvoir : `demarrer` et s'`arreter`. Pour clarifier l'organisation des véhicules, nous introduisons une hiérarchie incluant les `Roulant` (possédant une méthode `void rouler()`), les `Volant` (méthode `voler()`) et les `Flottant` (méthode `naviguer()`).

Q 46.1 Donner la hiérarchie d'interface à créer.

```
Vehicule
  + void demarrer()
  + void arreter()

|__ Roulant extends Vehicule
        + void rouler()
|__ Volant extends Vehicule
        + void voler()
|__ Flottant extends Vehicule
        + void naviguer()
```

Q 46.2 Donner la signature de la classe `Voiture` et les méthodes à coder impérativement.

```
public class Voiture implements Roulant{
  public void demarrer(){..}
  public void arreter(){..}
  public void rouler(){..}
}
```

Q 46.3 Donner la signature de la classe `Hydravion` et les méthodes à coder impérativement.

```
1 public class Hydravion implements Volant,Roulant{
2   public void demarrer(){..}
3   public void arreter(){..}
4   public void voler(){..}
5   public void naviguer(){..}
6 }
```

Exercice 47 – Interface réversible

Une interface correspond à une propriété. Nous envisageons dans cet exercice la propriété de reversibilité. Pour une chaine de caractères, il s'agit de pouvoir la lire à l'envers lorsqu'on le souhaite, pour un tableau, de prendre les éléments dans l'ordre opposé. Nous choisissons arbitrairement de définir cette propriété sans argument et sans retour : il s'agit juste de modifier l'élément invoquant la méthode.

Q 47.1 Donner le code de l'Interface `Reversible`

```
1 public interface Reversible {
2 public void reverse();
3 }
```

Q 47.2 Donner le code de la classe `StringReversible`

```
1 public class StringReversible implements Reversible{
2     private String str;
3
4     public StringReversible(String str) {
5         this.str = str;
6     }
7
8     public void reverse() { // stocker la chaine renversee simplifie le toString
9         String str2 = "";
10        for(int i=str.length()-1; i>=0; i--)
11            str2 += str.charAt(i);
12        str = str2;
13    }
14
15    public String toString(){
16        return str;
17    }
18 }
```

Q 47.3 Nous souhaitons maintenant créer une structure de type `ArrayList` réversible. Donner le code étendant l'`ArrayList<Object>`, ajoutant les méthodes nécessaire (dont `toString()` et surchargeant la méthode `get`.
NB : ajouter un attribut booléen indiquant si la structure est renversée ou pas.

```
1 import java.util.ArrayList;
2
```

```
3 public class ArrayRev extends ArrayList<Object> implements Reversible{
4     private boolean rev;
5
6     public ArrayRev() {
7         super();
8         rev = false;
9     }
10
11    public void reverse() {
12       rev = !rev;
13    }
14
15    public Object get(int i){
16        return rev?super.get(size()-1-i):super.get(i);
17    }
18
19    public String toString(){
20        String str="";
21        for(int i=0; i<size(); i++){
22            str += get(i).toString();
23        }
24        return str;
25    }
26 }
```

Q 47.4 Ajouter quelques lignes de code pour rendre la réversibilité récursive quand c'est possible dans la structure précédente. Par exemple, quand la liste contient des `StringReversible`, nous souhaitons renverser la liste ET renverser les éléments de la liste si c'est possible.

```
1     public void reverse() {
2         for(Object o: this)
3             if(o instanceof Reversible)
4                 ((Reversible) o).reverse();
5         rev = !rev;
6     }
```

Q 47.5 (Option) Proposer une seconde implémentation de la structure de données récursive basée sur la composition et non plus sur l'héritage (attribut `ArrayList` au lieu de `extends ArrayList`)

```
1 public class ArrayRev2 implements Reversible{
2     private ArrayList<Object> li;
3
4     public ArrayRev2() {
5         li = new ArrayList<Object>();
6     }
7
8     public void add(Object o){
9         li.add(o);
10    }
11
12    public void reverse() {// recursif
13        ArrayList<Object> tmp = new ArrayList<Object>();
```

```
14          for(int i=li.size()-1; i>=0; i--){
15              if(li.get(i) instanceof Reversible)
16                  ((Reversible) li.get(i)).reverse();
17              tmp.add(li.get(i));
18          }
19          li = tmp;
20      }
21
22      public String toString(){
23          String str="";
24          for(int i=0; i<li.size(); i++){
25              str += li.get(i).toString();
26          }
27          return str;
28      }
29  }
```

Exercice 48 – Interface comparable

Nous nous plaçons maintenant comme utilisateur d'un cadre défini pour les interfaces. Nous avons besoin de trier une liste de vecteurs en fonction de leur norme. Nous disposons de la classe de base :

```
1 public class Vecteur {
2     private double x,y;
3     public Vecteur(double x, double y){
4         this.x = x;
5         this.y = y;
6     }
7     public double norme(){return Math.sqrt(x*x+y*y);}
8 }
```

La Javadoc nous indique : (1) dans la classe `Collections` :

```
1 static <T extends Comparable<? super T>> void  sort(List<T> list)
2 // Sorts the specified list into ascending order, according to the natural ordering of its
      elements.
3 static <T> void sort(List<T> list , Comparator<? super T> c)
4 // Sorts the specified list according to the order induced by the specified comparator.
```

(2) Interface Comparable

```
1 int      compareTo(T o) // Compares this object with the specified object for order.
2 // si x<y alors, x.compareTo(y) < 0
3 // si x.equals(y) alors  x.compareTo(y) == 0
4 // sinon x.compareTo(y) > 0
```

(3) Interface Comparator

Q 48.1 Indiquer les modifications à effectuer dans la classe `Vecteur` pour utiliser `Comparable`

Q 48.2 Donner le code d'un main effectuant le tri d'une liste de `Vecteur` générée aléatoirement par rapport à leurs normes.

Le but n'est pas d'insister sur la syntaxe générique... Mais on est obligé de l'effleurer.

```
1 public class Vecteur implements Comparable<Vecteur> {
2 ...
3     public int compareTo(Vecteur o) {
4         double d1 = this.norme(); double d2 = o.norme();
```

```
        if(d1 < d2) return -1;
        if(d1 == d2) return 0;
        return 1;
    }

public class Test {
    public static void main(String[] args) {
        ArrayList<Vecteur> arr = new ArrayList<Vecteur>();
        for(int i = 0; i<3; i++){
            arr.add(new Vecteur(Math.random()*10, Math.random()*10));
            System.out.println(arr.get(arr.size()-1));
        }
        Collections.sort(arr);
        for(Vecteur v:arr){
            System.out.println(v);
        }
    }
}
```

Q 48.3 Donner la procédure et le code pour utiliser un `Comparator`. Quel est l'avantage de cette approche ?

Avantage : pas besoin de modifier les classes existantes !

```
import java.util.Comparator;

public class VComp implements Comparator<Vecteur>{
  // constructeur implicite

    public int compare(Vecteur o1, Vecteur o2) {
        double d1 = o1.norme(); double d2 = o2.norme();
        if(d1 < d2) return -1;
        if(d1 == d2) return 0;
        return 1;
    }
}
// Dans le main

public class Test {
    public static void main(String[] args) {
        ...
        Collections.sort(arr, new VComp());
        ...
    }
}
```

Quizz 12 – Héritage et liaison dynamique

Soient les 4 classes suivantes :

```
public class Animal {
        public void f() { }
        public String toString() {return "Animal";}
}
public class Poisson extends Animal {
        public void g() { }
        public String toString() {return "Poisson";}
}
```

```
9 public class Cheval extends Animal { }
10 public class Zoo { }
```

et les déclarations suivantes :

```
1 Animal a1=new Animal();
2 Poisson p1=new Poisson();
3 Cheval c1=new Cheval();
4 Zoo z1=new Zoo();
```

QZ 12.1 Parmi les instructions suivantes, lesquelles provoquent une erreur à la compilation ? Expliquez.

- `a1.f();`
- `p1.f();`
- `a1.g();`
- `p1.g();`

a1.f() ;p1.f() ;p1.g() ; sont correctes.
a1.g() ; est incorrecte car la méthode g() n'existe pas dans la classe Animal.
Pour vous en convaincre, changez le nom de la méthode f() par vieillir() et le nom de la méthode g() par nager().
Les animaux et les poissons vieillisent, mais seuls les poissons nagent

QZ 12.2 Que retournent les instructions suivantes ?

- `a1.toString()`
- `p1.toString()`
- `c1.toString()`
- `z1.toString()`

- `a1.toString()` retourne "Animal"
- `p1.toString()` retourne "Poisson"
- `c1.toString()` retourne "Animal" (comme toString() n'est pas définie dans la classe Cheval, c'est la méthode toString() de Animal qui est utilisée)
- `z1.toString()` retourne Zoo@1bc4459 (comme toString() n'est pas définie dans la classe Zoo, c'est la méthode toString() de Object qui est utilisée)

QZ 12.3 Parmi les instructions suivantes, lesquelles provoquent une erreur à la compilation ? Expliquez.

- `Animal a2=p1;`
- `Animal a3=(Animal)p1;`
- `Poisson p2=a1;`
- `Poisson p3=(Poisson)a1;`

Animal a2=p1 ;
Animal a3=(Animal)p1 ;
Poisson p3=(Poisson)a1 ;
=> pas d'erreurs à la compilation,
mais Poisson p3=(Poisson)a1 ; provoque une erreur à l'exécution car l'objet référencé par a1 est réellement un animal pas un poisson
Poisson p2=a1 ; provoque une erreur à la compilation, car on ne peut pas affecter un Animal dans un Poisson