Héritage

Fred Hémery

IUT Béthune
Département Réseaux & Télécommunications

Cours I6 — 07/08

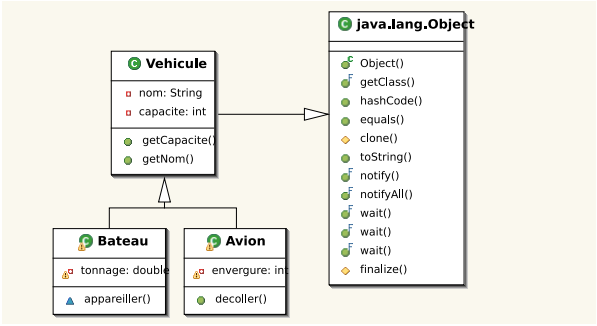
Introduction

- La notion d'héritage permet de réduire le développement de code en facilitant la réutilisation
- L'héritage va permettre de créer une nouvelle classe qui va étendre, spécialiser le code d'une classe existante
- la classe qui hérite est dite sous-classe
- la classe qui est héritée est dite super-classe

Introduction

- En java, une classe ne peut avoir qu'une super-classe. On parle d'héritage simple.
- Une classe peut être super classe de plusieurs sous-classes.
- De manière implicite une classe java hérite de la classe `Object` (à pour super classe)
- On indique de manière explicite le nom de la super classe en utilisant le mot clé "`extends`"

Introduction



Introduction

```
public class Vehicule {
private String nom;
private int capacite;

public int getCapacite() {
return capacite;
}
}
```

```
public class Bateau extends Vehicule {
private double tonnage;

public void appareiller() {
System.out.println("appareiller");
}
}
```

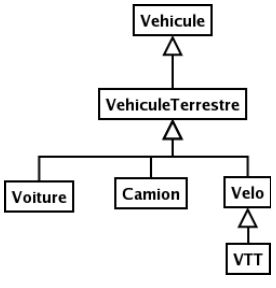
```
public class Avion extends Vehicule {
private double envergure;

public void decoller() {
System.out.println("decoller");
}
}
```


Introduction

- Le mécanisme d'héritage permet
  - d'ajouter des variables tout en conservant les variables qui ont été déclarées dans la super-classe.
  - de bénéficier et modifier les méthodes implémentées dans la super-classe.
  - d'ajouter de nouvelles méthodes (et des constructeurs).
- Le mécanisme d'héritage évite le développement de même code plusieurs fois en facilitant la réutilisation du code existant.
- Le mécanisme d'héritage est transitif

Hiérarchie de classes

- L'héritage ne se limite pas nécessairement à un seul niveau de dérivation.
- Il est possible de définir une hiérarchie de classes sur plusieurs niveaux.
- Une hiérarchie de classes s'appelle également hiérarchie d'héritage.



Ancêtres et Descendants

- Soit un chemin quelconque dans une hiérarchie de classes menant d'une classe D à une classe A.
  - la classe D est dite classe descendant de A, la classe A est dite classe ancêtre de D.
  - Par exemple, `VTT` est une classe descendant de `Vehicule`.

Héritage et création d'objets

- Lors de la création d'un objet d'une sous-classe (qui hérite d'une super-classe), le constructeur de la super-classe est appelé avant le constructeur de la sous-classe.
  - Si aucun constructeur n'a été défini dans la super classe alors un constructeur implicite est exécuté
  - Si un constructeur sans paramètre a été défini dans la super classe, il sera exécuté
  - Si un constructeur avec paramètre a été défini dans la super classe, il faudra explicitement l'exécuter dans la sous-classe (sinon il y aura une erreur de syntaxe).

Héritage et création d'objets

```
public class Vehicule {
  private String nom;
  private int capacite;
  Vehicule () {System.out.println("Cons Vehicule");}
  public int getCapacite() {
    return capacite;
  }
}
```

```
public class Bateau extends Vehicule {
  private double tonnage;
  Bateau () {System.out.println("Cons Bateau");}
  public void appareiller() {
    System.out.println("appareiller");
  }
}
```

```
public class Avion extends Vehicule {
    private double envergure;
    Avion () {System.out.println("Cons Avion");}
    public void decoller() {
      System.out.println("decoller");
  }
}
```


Héritage et création d'objets

```
public class TestV {
public static void main (String [] args) {
Avion a = new Avion();
Bateau b = new Bateau();
}
```

```
[---] java TestV
Cons Vehicule
Cons Avion
Cons Vehicule
Cons Bateau
[---]
```

Si on modifie le constructeur de Vehicule :

```
public class Vehicule {
private String nom;
private int capacite;

Vehicule (int i) {System.out.println("Cons Vehicule");}
public int getCapacite() {
return capacite;
}
}
```

Il y a une erreur de compilation.

Héritage et création d'objets

Pour exécuter de manière explicite un constructeur de la super-classe on utilise le mot clé `super`. L'appel doit être la première instruction.

```
public class Bateau extends Vehicule {
  private double tonnage;
  Bateau () {
    super(10);
    System.out.println("Cons Bateau");
  }
  public void appareiller() {
    System.out.println("appareiller");
  }
}
```

```
public class Avion extends Vehicule {
  private double envergure;
  Avion () {
    super(10);
    System.out.println("Cons Avion");
  }
  public void decoller() {
    System.out.println("decoller");
  }
}
```


Le transtypage

- Le transtypage c'est la possibilité pour une variable (référence) v d'un type de classe C1 d'accepter une assignation avec un objet d'une autre classe C2.
- Il y a deux types de transtypage
  - transtypage ascendant : la classe C1 est un ascendant de la classe C2.
  - transtypage descendant : la classe C1 est un descendant de la classe C2. Dans ce cas le transtypage sera explicite.
- Toutes autres situations seront refusées par le compilateur

Le transtypage

Exemple de transtypage ascendant

```
public class TestVV {
  public static void main (String [] args) {
    Vehicule v1 = new Avion();  // Transtypage ascendant implicite
    Vehicule v2 = (Vehicule)new Avion(); // Transtypage ascendant explicite
}
```

```
public class TestVV {
  public static void main (String [] args) {
    Vehicule v = new Avion();
    Avion a = v; // interdit
    a = (Avion)v // autoris , Transtypage descendant explicite
    if (v instanceof Avion)
      a = (Avion)v
}
```


Le transtypage

- le mot clé `instanceof` permet de vérifier si une référence r est un descendant d'une classe C.

```
Vehicule v = new Vehicule();
System.out.println( v instanceof Vehicule); // affiche true
System.out.println( v instanceof Avion);    // affiche false
v = new Avion();
System.out.println( v instanceof Vehicule); // affiche true
System.out.println( v instanceof Avion);    // affiche true
Avion a = null;
System.out.println( a instanceof Avion);    // affiche false
```


Masquage et redéfinition

- Lorsqu'une classe C2 hérite d'une classe C1, elle hérite de tous ces membres (variables et méthodes).
- Il est parfois intéressant de modifier les membres hérités (pour les adapter à la spécialisation) en plus d'en ajouter d'autres.
- Il existe deux mécanismes pour modifier les membres
  - le masquage
  - la redéfinition

Masquage et redéfinition

- le masquage : consiste à avoir deux champs de même nom, ou deux méthodes de même profil (nom et paramètres identiques). Le premier vient de la super-classe, le second est défini dans la sous-classe.

Masquage et redéfinition

- le masquage : consiste à avoir deux champs de même nom, ou deux méthodes de même profil (nom et paramètres identiques). Le premier vient de la super-classe, le second est défini dans la sous-classe.
- la redéfinition (sur les méthodes) : la sous-classe qui redéfinit une méthode ne peut plus la voir telle qu'elle avait été définie dans la super-classe.

Masquage des variables

```
public class Disque {
  double x, y, rayon;
  int refCouleur = 5;
  Disque(double x1, double y1, double r) {
    x = x1;
    y = y1;
    rayon = r;
  }
  Disque() { this(0.0, 0.0, 1.0);}
  double surface() { return Math.PI * rayon * rayon; }
}
```

```
public class Anneau extends Disque {
  double rayonInt;
  String refCouleur = "bleu";
  Anneau (double x1, double y1, double rayon, double rayonInterne){
    super(x1,y1,rayon);
    rayonInt = rayonInterne;
  }
  Anneau() { this(0.0, 0.0,1.0,0.0);}
  String refCouleur() { return refCouleur; }
  int refCouleurMasque() { return super.refCouleur; }
  double surface() {
    return Math.PI * rayon * rayon - Math.PI * rayonInt * rayonInt;
  }
}
```


Masquage des variables

```
public class TestD {
  public static void main (String [] args) {
    Disque d = new Disque(10.0, 10.0, 2.0);
    Anneau a = new Anneau(6.0, 3.0);
    System.out.println(d.refCouleur); // affiche 5
    System.out.println(a.refCouleur); // affiche bleu
    System.out.println(a.refCouleurMasque()); // affiche 5

    d = a; // transtypage ascendant
    System.out.println(a.refCouleur); // affiche bleu
    System.out.println(d.refCouleur); // affiche 5
    System.out.println(((Disque)a).refCouleur); // affiche 5
  }
}
```

- La résolution des noms de variables atteints par une référence est effectuée statiquement, en fonction du type déclaré de la référence.

Redéfinition de méthode

- La redéfinition d'une méthode permet d'affiner son comportement à la sous-classe.

```
double surface() { //red finition de surface pour la classe Anneau
  return super.surface() - Math.PI * rayonInt * rayonInt;
}
```

```
public class TestDD {
  public static void main (String [] args) {
    Disque d = new Disque(10.0, 10.0, 1.0);
    Anneau a = new Anneau(3.0, 2.0);
    System.out.println(d.surface()); // affiche 3.141592653589793
    System.out.println(a.surface()); // affiche 15.707963267948966

    d = a; // transtypage ascendant
    System.out.println(d.surface()); // affiche 15.707963267948966
    System.out.println(a.surface()); // affiche 15.707963267948966
  }
}
```

- La recherche de la méthode est effectuée dynamiquement. En fonction de l'objet associé à la référence, quelque soit le type déclaré.

Redéfinition de méthode

- Une méthode en redéfinissant une autre doit :
  - avoir le même nom que la méthode qu'elle redéfinit;
  - avoir le même nombre de paramètres et que ceux-ci soient du même type (sinon surcharge);
  - avoir le même type de retour;
  - ne pas être static (sinon masquage);
  - avoir une accessibilité au moins aussi grande (protected ⟶ public ok, public ⟶ protected pas ok)

Surcharge de méthode et héritage

- Différence entre surcharge et redéfinition
  - le type des paramètres doit être exactement le même dans le cas d'une redéfinition; sinon, il s'agit d'une surcharge.

```
public class Disque {
  ...
  void intersection (Disque autreDisque){
    System.out.pritnln("Intersection dans disque"); }
}
```

```
public class Anneau extends Disque {
  ...
  void intersection (Anneau autreAnneau){
    System.out.pritnln("Intersection dans Anneau(surcharge)"); }
}
```

```
...
Disque d = new Disque();
Anneau a = new Anneau();
d.intersection(d); // affiche Intersection dans disque
a.intersection(a); // affiche Intersection dans Anneau(surcharge)
a.intersection(d); // affiche Intersection dans disque
...
```


Surcharge de méthode et héritage

- Différence entre surcharge et redéfinition
  - le type des paramètres doit être exactement le même dans le cas d'une redéfinition; sinon, il s'agit d'une surcharge.

```
public class Disque {
  ...
  void intersection (Disque autreDisque){
    System.out.pritnln("Intersection dans disque"); }
}
```

```
public class Anneau extends Disque {
  ...
  void intersection (Disque autreDisque){
    System.out.pritnln("Intersection dans Anneau(red finition)"); }
}
```

```
...
Disque d = new Disque();
Anneau a = new Anneau();
d.intersection(d); // affiche Intersection dans disque
d.intersection(a); // affiche Intersection dans disque
a.intersection(a); // affiche Intersection dans Anneau(red finition)
a.intersection(d); // affiche Intersection dans Anneau(red finition)
...
```

Le mot clé `super`

- Il y a trois utilisations possibles du mot clé super :
  - appel du constructeur de la super-classe `super()`. Il n'est pas possible d'aller voir plus loin (super.super() est interdit);

Le mot clé `super`

- Il y a trois utilisations possibles du mot clé super :
  - appel du constructeur de la super-classe `super()`. Il n'est pas possible d'aller voir plus loin (super.super() est interdit);
  - appel d'une méthode dans la super-classe `super.m()`

Le mot clé `super`

- Il y a trois utilisations possibles du mot clé super :
  - appel du constructeur de la super-classe `super()`. Il n'est pas possible d'aller voir plus loin (super.super() est interdit);
  - appel d'une méthode dans la super-classe `super.m()`
  - appel d'une variable masquée dans la super-classe `super.refCouleur`

Le mot clé `final`

- Le mot clé final peut être utilisé:
  - pour la déclaration d'une classe
- Il signifie que l'élément ne peut pas évoluer.

Le mot clé `final`

- Le mot clé final peut être utilisé:
  - pour la déclaration d'une classe
  - pour la déclaration d'une méthode
- Il signifie que l'élément ne peut pas évoluer.

Le mot clé `final`

- Le mot clé final peut être utilisé:
  - pour la déclaration d'une classe
  - pour la déclaration d'une méthode
  - pour la déclaration d'une variable
- Il signifie que l'élément ne peut pas évoluer.

Le mot clé `final`

- Le mot clé final peut être utilisé:
  - pour la déclaration d'une classe
  - pour la déclaration d'une méthode
  - pour la déclaration d'une variable
  - pour la déclaration d'un paramètre
- Il signifie que l'élément ne peut pas évoluer.

Le mot clé `final`

- On ne peut pas étendre une classe qui a été déclarée `final`.
- On ne peut pas redéfinir une méthode qui a été déclarée `final`.
  - On augmente l'efficacité à l'exécution
  - Toute méthode déclarée `private` est implicitement `final`.
- On ne peut pas modifier la valeur d'une variable (primitive) déclarée final après son initialisation :
  - l'initialisation peut se faire à la compilation, ou encore
  - à l'exécution

Le mot clé `final`

- On ne peut pas modifier la valeur d'une variable (référence) déclarée `final` après son initialisation
  - après initialisation, la valeur de la référence ne pourra pas être modifiée, cependant
  - les valeurs associées à la référence pourront elles, être modifiées.

Exemple d'utilisation

```
final class toto {
  // les    lments    de cette classe ne pourront pas   tre    tendus
}

public class BlankFinal {
  private final int i = 0; // Var final initialis e
  private final int j; // Var final non initialis e
  private final Disque d; // Var ref final non initialis e
    // Les var non initialis es doivent l' tre
    //dans le constructeur
  public BlankFinal() {
    j = 1; // Initialise Var final non initialis e
    d = new Disque(); // Initialise Var ref final non initialis e
  }
  final void f() {
    System.out.println("BlankFinal.f()");
  }

  public static void main(String[] args) {
    new BlankFinal();
  }
}
```


Les interfaces

- En java une classe ne peut hériter directement que d'une seule super-classe.
- La notion d'interface est un mécanisme permettant de simuler l'héritage multiple en adoptant un style de programmation par abstraction
- Une interface définit le **"quoi"** mais pas le **"comment"**.
  - Les méthodes sont déclarées mais ne sont pas définies (codées).
  - Les variables sont implicitement `static final`.

Les interfaces

- Une interface peut être considérée comme une déclaration d'intentions.
- Pour créer une interface on utilise le mot clé `interface` à la place du mot clé `class`.

```
interface figure {

  int NORD = 1;          // implicitement static final
  int EST =  1 << 1;     //
  int SUD =  1 << 2;     //
  int OUEST = 1 << 3;    //

  void dessine();        // implicitement public
  void deplace(Point p);
  void zoom(double facteur);
}
```

- On ne peut pas instancier une interface.

Les interfaces

- Une classe déclare qu'elle souscrit au contrat définit par l'interface en utilisant le mot clé `implements`.
  Dans ce cas la classe doit implanter (coder) chaque méthode définie dans l'interface (ou alors se déclarer abstraite, voir plus loin).

```
class rectangle implements figure {

  public void dessine(){ ...};
  public void deplace(Point p){ ... };
  public void zoom(double facteur) { ... };
}
```

- Une classe peut instancier plusieurs interfaces

```
class A implements I1, I2, I3 {
  ...
}
```


Les classes abstraites

- Dans le paradigme objet, il peut être utile de modéliser son application en spécifiant des concepts trop générique pour être instanciés.
  - par exemple véhicule peut être vu comme un concept (objet dont la fonction est de déplacer des charges et/ou des personnes).
- En java, une classe abstraite permet de déclarer un concept, sans le définir (complètement)
- Une classe abstraite est une classe qui n'est pas (qui ne peut pas être) entièrement définie.

Les classes abstraites

- En java, on définit une classe abstraite en utilisant le mot clé `abstract` avant le mot clé `class`.

```
public abstract class instrument {

  /* d claration de variables ... */

  public abstract void joue();
  public String description () { ... };
  public abstract void volume(double facteur);
}
```

- Toute classe qui contient au moins une méthode abstraite doit être déclarée comme abstraite.
- Une classe abstraite ne peut pas être instanciée.

Les classes abstraites

- Les constructeurs et méthodes statiques d'une classe abstraite ne peuvent pas être abstraits.
- Une classe abstraite ne contient pas forcement de méthodes abstraites.
- Une classe qui hérite d'une classe abstraite sera elle même abstraite sauf si elle implante les méthodes abstraites de la super-classe.

```
public class saxo extends instrument {

  /* d claration de variables ... */

  public void joue(){ ... };
  public void volume(double facteur) { ... };
}
```

Les classes internes

- Il est possible en java de déclarer une classe (interface) à l'intérieur d'une classe (interface, méthode).

```
public class ClasseExt {
  int i;
  public class ClasseInt {
    public void m() {}
  }
}
```

- La classe interne a accès aux variables de la classe englobante.

```
public class ClasseExt {
  int i;
  static int j;
  public static class ClasseInt {// acc s     j mais pas     i
    public void m() {}
  }
}
```

- La classe interne statique a accès aux variables statiques de la classe englobante.

Les classes internes

```
public class Table {
  private Object [] table;
  private int index;
  private class Iter implements Iterator {
    private int indexIt;
    public Iter () {
      index = 0;
    }
    public void remove() {
      throw new UnsupportedOperationException();
    }
    public boolean hasNext() { return indexIt < index; }
    public Object next() {    return table[indexIt++]; }
  }

  public Table (int taille) {
    table = new Object [taille];
    index = 0;
  }
  public void add (Object o) {  table[index++] = o; }

  public Iterator iterator() {   return new Iter();  }
}
```


Les classes internes locales

```
public class Table {
  private Object [] table;
  private int index;
  public Table (int taille) {
    table = new Object [taille];
    index = 0;
  }
  public void add (Object o) {
    table[index++] = o;
  }
  public Iterator iterator() {
    class Iter implements Iterator { \\ classe locale     la  m thode
      private int indexIt;
      public Iter () { index = 0; }
      public void remove() {
        throw new UnsupportedOperationException();
      }
      public boolean hasNext() { return indexIt < index; }
      public Object next() { return table[indexIt++]; }
    }
    return new Iter();
  }
}
```


Les classes anonymes

```
public class Table {
  private Object [] table;
  private int index;
  public Table (int taille) {
    table = new Object [taille];
    index = 0;
  }
  public void add (Object o) { table[index++] = o; }
  public Iterator iterator() {
    return new Iterator () { //cr ation d'une classe anonyme
      private int indexIt;
      public Iter () {  index = 0;  }
      public void remove() {
        throw new UnsupportedOperationException();
      }
      public boolean hasNext() { return indexIt < index; }
      public Object next() { return table[indexIt++]; }
    }; //ne pas oublier le point virgule qui termine l'instruction
  }
}
```


Illustrations

- L'interface `Iterator` du paquetage `java.util`

```
public interface Iterator {
  boolean hasNext();
  Object next();
  void remove();
}
```

- Parcours d'un `ArrayList` avec un `Iterator`

```
ArrayList a ...
  ...
Iterator it = a.iterator();
while (it.hasnext()) {
  // traitement de l'  lment    courant
}
```

- La méthode `iterator` de la classe `ArrayList`

```
package java.util;
public class ArrayList {
  ...
  public Iterator iterator() {
    return new ArrayIterator (this);
  }
```


Illustrations

- La class `ArrayIterator` du paquetage `java.util`

```
package java.util;
public class ArrayIterator implements Iterator {
  private ArrayList a;
  private int index;
  public ArrayIterator (ArrayList a) {
    rthis.a = a;
    index = 0;
  }
  public boolean hasNext() { ... }
  public Object next() { ... }
   void remove () { ... }
}
```


Illustrations

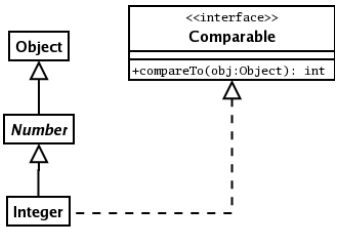
- L'interface `Comparable` du paquetage `java.lang`

```
package java.lang;
public interface Comparable {
  public int compareTo(Object obj);
}
```

- La méthode `compareTo` fonctionne de la manière suivante: soit `n = obj1.compareTo(obj2)` ⇒
  - `n < 0` si `obj1` précède `obj2`
  - `n == 0` si `obj1` est égal à `obj2`
  - `n > 0` si `obj1` suit `obj2`
- L'interface `Comparable` permet de définir un ordre naturel pour les objets `obj`. Certaines méthodes statiques de la classe `java.util.Arrays`, l'utilise pour faire un tri.

Illustrations

- La classe `Integer` et l'interface `Comparable`
  - La classe `Integer` hérite de la classe abstraite `Number` et implante l'interface `Comparable`



Illustrations

- Définir un ordre naturel sur les objets d'une classe suppose que l'on puisse la modifier.
- Il est parfois utilise de définir plusieurs ordres différents pour la même classe d'objets.
- On utilise alors l'interface `java.util.Comparator`

```
package java.util;
public interface Comparator {
  public int compare(Object obj1, Object obj2);
  public boolean equals(Object obj);
}
```

- La méthode `compare` fonctionne de la manière suivante: soit `n = compare(obj1,obj2)` ⇒
  - `n < 0` si `obj1` précède `obj2`
  - `n == 0` si `obj1` est égal à `obj2`
  - `n > 0` si `obj1` suit `obj2`

Illustrations

- La classe `Arrays` du paquetage `java.util` contient des méthodes qui permettent
  - de trier un tableau
  - d'effectuer une recherche dichotomique (dans un tableau trié)
- les méthodes de tri sont:

```
public static void sort (T [] a); // T un type primitif
public static void sort ( T [] a, int iDeb, int iFin);
public static void sort (Object [] a);
public static void sort (Object [] a, int iDeb, int iFin);
public static void sort (Object [] a, Comparator c);
public static void sort (Object [] a, int iDeb, int iFin, Comparator c);
```

- Le tableau d'objets est trié avec
  - la méthode `compareTo`
  - la méthode `compare` de la classe `Comparator`

Illustrations

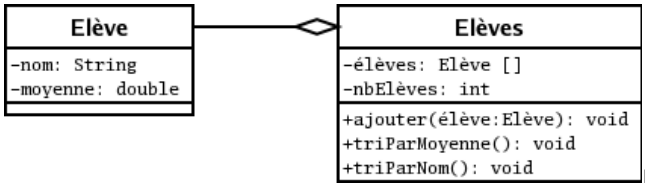
- Tri d'un tableau d'entiers

Le programme suivant donne un exemple d'utilisation de la méthode `sort` de la classe `Arrays`

```
import java.util.Arrays;
public class TriEntiers {

  public static void main(String[] args) {
    int [] tabEntiers = {10, 5, 3, 2, 50, 20};
    Arrays.sort(tabEntiers);
    for (int i = 0; i < tabEntiers.length; i++)
      System.out.print(tabEntiers[i] + " ");
    System.out.println();
  }
}
```


Illustrations

- Tri d'un tableau d'élèves



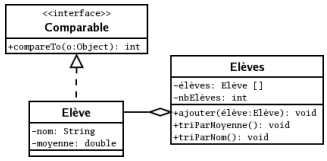
Po

utiliser la méthode `Arrays.sort`, il faut que la classe `Eleve` implante l'interface `Comparable` ou encore `Comparator`

```
import java.util.Arrays;
public class Eleves {
  private Eleve [] eleves;
  private int nbEleves;
    ...
  public void triParMoyenne() {
    Arrays.sort(eleves, 0, nbEleves);
  }
}
```


Illustrations

- Tri d'un tableau d'élèves




```
public class Eleve implements Comparable {
  private String nom;
  private double moyenne;

  public int compareTo(Object o) {
    Eleve eleve = (Eleve)o;
    if (moyenne > eleve.moyenne)
      return 1;
    else if (eleve.moyenne > moyenne)
      return -1;
    else return nom.compareTo(eleve.nom);
  }
}
```


Illustrations

Si on veut faire des tris avec des critères différents, il est préférable d'utiliser des classes internes statiques qui implantent la classe `Comparator`.

```
public class Eleve implements Comparable {
  ...
  public static TRI_MOYENNE implemens Comparator {
    public int compare(Object o1, Object o2) {
      Eleve eleve1 = (Eleve) o1;
      Eleve eleve2 = (Eleve) o2;
      if (eleve1.moyenne > eleve2.moyenne)
        return 1;
      else if (eleve2.moyenne > eleve1.moyenne)
        return -1;
      else
        return eleve1.nom.compareTo(eleve2.nom);
    }
  }
  public static TRI_NOM implements Comparator {
    public int compare(Object o1, Object o2) {
      Eleve eleve1 = (Eleve) o1;
      Eleve eleve2 = (Eleve) o2;
      return eleve1.nom.compareTo(eleve2.nom);
    }
  }
```


Illustrations

```
public class Eleves {
  private Eleve [] eleves;
  private int nbEleves;
  public Eleves () {
    eleves = new Eleve[10];
    nbEleves = 0;
  }
  public void ajouterEleve(Eleve eleve) {
    ...
  }
  public void triParMoyenne() {
    Arrays.sort(eleves,0, nbEleves, new Eleve.TRI_MOYENNE());
  }
  public void triParNom() {
    Arrays.sort(eleves,0, nbEleves, new Eleve.TRI_NOM());
  }
  public void liste() {
    for (int i = 0; i < nbEleves; i++)
      System.out.println(eleves[i]);
  }
```


Illustrations

```
Eleves eleves = new Eleves();
  eleves.ajouterEleve(new Eleve("Dupont", 10.5d));
  eleves.ajouterEleve(new Eleve("Duval", 11d));
  eleves.ajouterEleve(new Eleve("Dupond", 10.5d));
  eleves.ajouterEleve(new Eleve("Martin", 9.5d));
  System.out.println("Liste tri par nom");
  eleves.triParNom();
  eleves.liste();
  System.out.println("Liste tri par moyenne");
  eleves.triParMoyenne();
  eleves.liste();
```

Résultat

```
Liste tri par nom
[Dupond, 10.5]
[Dupont, 10.5]
[Duval, 11.0]
[Martin, 9.5]
Liste tri par moyenne
[Martin, 9.5]
[Dupond, 10.5]
[Dupont, 10.5]
[Duval, 11.0]
```