Interfaces Comparable<T> et Comparator<T>

I Ordre naturel: interface prédéfinie Comparable<T>

- Une classe peut spécifier un ordre naturel en implantant l'interface prédéfinie Comparable<T> du package java.lang.
- T doit être la classe spécifiant l'ordre
- Valeur de retour de la méthode d'interface **int compareTo(T t)** :

  <0 si this est **inférieur** à t

  ==0 si this est **égal** à t

  >0 si this est **supérieur** à t

L'implantation de **compareTo** doit être compatible avec celle d'**equals** !!

Si o1.equals(o2) == true alors o1.compareTo(o2) == 0 (et vice versa)

Utilisation:

- la méthode statique **public static void sort(Object[] a)** demande à ce que le tableau contienne des éléments mutuellement comparables

*Sorts the specified array of objects into **ascending order,** according to the natural ordering of its elements. All elements in the array must implement the **`Comparable`** interface. Furthermore, all elements in the array must be **mutually comparable** (that is, `e1.compareTo(e2)` must not throw a **`ClassCastException`** for any elements `e1` and `e2` in the array).*

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort.

***The sorting algorithm is a modified mergesort** (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n*log(n) performance.*

Parameters:

`a` *- the array to be sorted*

Throws:

`ClassCastException` *- if the array contains elements that are not mutually comparable (for example, strings and integers).*

- la méthode statique **void java.util..sort(Collections<Animal> list)** demande à ce que le liste contienne des éléments mutuellement comparables

Sorts the specified list into ***ascending order****, according to the* ***natural ordering*** *of its elements. All elements in the list must implement the* **`Comparable`** *interface. Furthermore, all elements in the list must be mutually comparable (that is,* `e1.compareTo(e2)` *must not throw a* **`ClassCastException`** *for any elements* `e1` *and* `e2` *in the list).*

This sort is guaranteed to be stable*: equal elements will not be reordered as a result of the sort.*

The specified list must be modifiable, but need not be resizable.

The sorting algorithm is a modified mergesort *(in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n log(n) performance. This implementation dumps the specified list into an array, sorts the array, and iterates over the list resetting each element from the corresponding position in the array. This avoids the* n^2 *log(n) performance that would result from attempting to sort a linked list in place.*

Parameters:

`list` *- the list to be sorted.*

Throws:

`ClassCastException` *- if the list contains elements that are not mutually comparable (for example, strings and integers).*

`UnsupportedOperationException` *- if the specified list's list-iterator does not support the* `set` *operation.*

See Also:

`Comparable`

Exemple:

```
package image;
import java.util.List;
import java.util.Collections;

public class Utilisation{
    public static void main(String args[]) {
      Utilisation.test();
    }

    public static void test()
    {
        Image image = new Image();


        image.addItem(new Rectangle(4,5));

        image.addItem(new Rectangle(6,7));

        image.addItem(new Square(6));

        image.addItem(new Triangle(12,8));

        System.out.println("TOUT LE MONDE AFFICHE SA TAILLE!");
        image.sizeAllItems();

        System.out.println("\n \n ON COMPARE SA TAILLE 2 A 2!");
        image.compareToAllItems();

        List <Forme> items = image.getItems();
        Collections.sort(items);
        for(Forme item : items)
            System.out.println(item);
    }
}
```

```
package image;

// LA CLASSE FORME EST ABSTRAITE CAR ELLE N'A PAS DEFINIE LA METHODE size()
// DE L'INTERFACE MEASURABLE
public  abstract class Forme implements Comparable<Forme>{
        /** @return -1, 0, 1 if this is <, = or > than x */
    public int compareTo(Forme x)  {
        if (this.size() < x.size()) {
            return -1 ;
        }
        if (this.size() > x.size()) {
            return 1 ;
        }
        return 0 ;
    }
}
```

```
package image;

// Comparables par leur taille

// ON RAJOUTE UNE CLASSE TRIANGLE SANS DEVOIR MODIFIER LA CLASSE IMAGE
public class Triangle extends Forme {
      private double base;
      private double hauteur ;

      public Triangle(double base, double hauteur) {
            this.base = base;
            this.hauteur= hauteur;
      }

      public double size() {
            return .5 * (base*hauteur);
      }

      @Override
      public String toString() {
            return "Triangle [taille =" + this.size() + "]";
      }
}
```

```
package image;

// Comparables par leur taille
public class Rectangle extends Forme {
    private double largeur ;
     private double longueur ;

    public Rectangle(double largeur, double longueur) {
        this.largeur = largeur;
        this.longueur = longueur;
    }

    public double size() {
        return largeur*longueur;
    }

    @Override
      public String toString() {
            return "Rectangle [taille =" + this.size() + "]";
      }
}
```

```
package image;

public class Square extends Rectangle {

    public Square(double largeur) {
        super(largeur, largeur);
    }

    @Override
      public String toString() {
            return "Carre [taille =" + this.size() + "]";
      }
}
```

```
package image;
import java.util.List;
import java.util.ArrayList;
import java.lang.Comparable;

public class Image{
    List<Forme> items = new ArrayList<Forme>();

    public List <Forme> getItems() {
            return items;
    }

    public void addItem(Forme x){
        items.add(x);
    }

    public void sizeAllItems() {

        Forme item;

        if (items.size() == 0)
            System.out.println("Rien dans l'image");
        else {
            for (int i = 0;i < items.size();i++) {
                  item = items.get(i);
                  // on affiche la taille représenté par un double
                System.out.println( item.size());
            }
        }
    }

    // ON COMPARE LES TAILLES 2 à 2
    public void compareToAllItems() {

        Forme item1, item2;

        if (items.size() == 0)
            System.out.println("Rien dans l'image");
        else {
            for (int i = 0;i < items.size() -1;i++) {
                  item1 = items.get(i);
                  item2 = items.get(i+1);

                  // ON COMPARE LES TAILLES 2 à 2
                   int resultat = item1.compareTo(item2);
                switch(resultat)
                {
                case -1: System.out.println(item1.toString()  + "<" +
item2.toString() ); break;
                case 0: System.out.println(item1.toString()  + "=" +
item2.toString() ); break;
                case 1: System.out.println(item1.toString()  + ">" +
item2.toString() ); break;
                default: System.out.println("LE PIRE EST ARRIVE!"); break;
                }
            }
        }
    }
}
```

II Comparaison externe: interface Comparator<T>

L'interface java.util.Comparator permet de spécifier un ordre externe

```
public interface Comparator<T> {
        public abstract int compare(T o1, T o2);
        public abstract boolean equals(Object o);
}
```

- Un ordre externe est un ordre valable juste à un moment donné (rien de naturel et d'évident)
- La valeur de retour de **compare** suit les mêmes règles que **compareTo**
- **en général equals est rédéfini (héritage de la classe Object)**

L'implantation de compare doit être compatible avec celle d'**equals** !!

Si o1.equals(o2)==true alors compare(o1,o2)==0 (et vice versa)

Exemple:

```
package comparator;

/*
Java Comparator example.
This Java Comparator example describes how java.util.Comparator
interface is implemented to compare Java custom class objects.

This Java Comparator is passed to Collection's sorting method
(for example Collections.sort method) to perform sorting of Java custom class
objects.
 */

import java.util.*;

/*
java.util.Comparator interface declares two methods,
1) public int compare(Object object1, Object object2) and
2) boolean equals(Object object)
 */

/*
We will compare objects of the Employee class using custom comparators
on the basis of employee age and name.
 */

public class Employee{
      private int age;
      private String name;

      public void setAge(int age){
            this.age=age;
      }

      public int getAge(){
            return this.age;
      }
      public void setName(String name){
            this.name=name;
      }

      public String getName(){
            return this.name;
      }
}

/*
User defined Java comparator.
To create custom java comparator, implement Comparator interface and
define compare method.
The below given comparator compares employees on the basis of their age.
*/
class AgeComparator implements Comparator<Employee>{
      public int compare(Employee emp1, Employee emp2){
            /*
             * parameter are of type Object, so we have to downcast it
             * to Employee objects
             */
            int emp1Age = emp1.getAge();
            int emp2Age = emp2.getAge();

            if(emp1Age > emp2Age)
                  return 1;
            else if(emp1Age < emp2Age)
                  return -1;
            else
                  return 0;
      }
}

/*
The below given comparator compares employees on the basis of their name.
*/
class NameComparator implements Comparator<Employee>{
      public int compare(Employee emp1, Employee emp2){
            //parameter are of type Object, so we have to downcast it to
Employee objects
            String emp1Name = ((Employee)emp1).getName();
            String emp2Name = ((Employee)emp2).getName();

            //uses compareTo method of String class to compare names of the
employee
            return emp1Name.compareTo(emp2Name);
      }
}
```

```
/*
This Java comparator example compares employees on the basis of
their age and name and sort them in that order.
*/

public class JavaComparatorExample{

      public static void main(String args[]){
            //Employee array which will hold employees
            Employee employee[] = new Employee[2];

            //set different attributes of the individual employee.
            employee[0] = new Employee();
            employee[0].setAge(40);
            employee[0].setName("Joe");

            employee[1] = new Employee();
            employee[1].setAge(20);
            employee[1].setName("Mark");

            System.out.println("Order of employee before sorting is");
            //print array as is.
            for(int i=0; i < employee.length; i++){
                  System.out.println( "Employee " + (i+1) + " name :: " +
employee[i].getName()
                                                           + ", Age :: " +
employee[i].getAge());
            }

            /*
        Sort method of the Arrays class sorts the given array.
        Signature of the sort method is,
        static void sort(Object[] object, Comparator comparator)

        IMPORTANT: All methods defined by Arrays class are static. Arrays class
        serves as a utility class.
             */

            //Sorting array on the basis of employee age by passing
AgeComparator
            Arrays.sort(employee, new AgeComparator());

            System.out.println("\n\nOrder of employee after sorting by employee
age is");
            for(int i=0; i < employee.length; i++){
                  System.out.println( "Employee " + (i+1) + " name :: " +
employee[i].getName() + ", Age :: " + employee[i].getAge());
            }

            //Sorting array on the basis of employee Name by passing
NameComparator
            Arrays.sort(employee, new NameComparator());

            System.out.println("\n\nOrder of employee after sorting by employee
name is");
            for(int i=0; i < employee.length; i++){
                  System.out.println( "Employee " + (i+1) + " name :: " +
employee[i].getName() + ", Age :: " + employee[i].getAge());
            }
      }
}

/*
```

```
OUTPUT of the above given Java Comparable Example would be :
Order of employee before sorting is
Employee 1 name :: Joe, Age :: 40
Employee 2 name :: Mark, Age :: 20
Order of employee after sorting by employee age is
Employee 1 name :: Mark, Age :: 20
Employee 2 name :: Joe, Age :: 40

Order of employee after sorting by employee name is
Employee 1 name :: Joe, Age :: 40
Employee 2 name :: Mark, Age :: 20
*/
```

Remarque: Ce qui se passe dans les coulisses du tri

Lors de l'appel **Arrays.*sort*(employee, new AgeComparator()),** la classe **AgeComparator** est instanciée.
C'est sur cette instance anonyme (pas de nom de variable) que l'algorihme de tri-fusion mis en oeuvre dans la méthode de tri **Sort** appelle la fonction de comparaison de 2 employés du tableau employee: **public int compare(Employee emp1, Employee emp2).**

Cette méthode est une méthode d'instance de la classe **AgeComparator** et a 2 paramètres de type **Employee**.

Tout ce passe comme si l'instance anonyme de la classe **AgeComparator**, que l'on pourrait appelée tmp, appelait la méthode **compare** dans l'algorithme de la méthode Sort:

```
          AgeComparator tmp = new AgeComparator();
          tmp.compare(emp1, emp2);
```

III Classe imbriquée et méthodes statiques (exemple de chez Sun)

```
package comparator;



/*
Java Comparator example.
This Java Comparator example describes how java.util.Comparator
interface is implemented to compare Java custom class objects.

This Java Comparator is passed to Collection's sorting method
(for example Collections.sort method) to perform sorting of Java custom class
objects.
 */

import java.util.*;

/*
java.util.Comparator interface declares two methods,
1) public int compare(Object object1, Object object2) and
2) boolean equals(Object object)
 */

/*
We will compare objects of the Employee class using custom comparators
on the basis of employee age and name.
 */


public class Employee{
      private int age;
      private String name;

      public void setAge(int age){ this.age=age; }

      public int getAge(){ return this.age; }

      public void setName(String name){ this.name=name; }

      public String getName(){ return this.name; }

      // 2 METHODES STATIQUES ==> sucre syntaxique
      /*
        Sort method of the Arrays class sorts the given array.
        Signature of the sort method is,
        static void sort(Object[] object, Comparator comparator)

        IMPORTANT: All methods defined by Arrays class are static. Arrays class
        serves as a utility class.
       */

      public static void sortAge(Employee [] employees)
      {
            Employee x = new Employee();
            Arrays.sort(employees, x.new AgeComparator());
      }
```

```
    public static void sortName(Employee [] employees)
    {
        Employee x = new Employee();
        Arrays.sort(employees, x.new NameComparator());
    }


    // 2 CLASSES IMBRIQUEEES

    /*
    User defined Java comparator.
    To create custom java comparator, implement Comparator interface and
    define compare method.
    The below given comparator compares employees on the basis of their age.
    */
    class AgeComparator implements Comparator<Employee>{
        public int compare(Employee emp1, Employee emp2){
            /*
             * parameter are of type Object, so we have to downcast it
             * to Employee objects
             */
            int emp1Age = emp1.getAge();
            int emp2Age = emp2.getAge();

            if(emp1Age > emp2Age)
                return 1;
            else if(emp1Age < emp2Age)
                return -1;
            else
                return 0;
        }
    }


    /*
    The below given comparator compares employees on the basis of their name.
    */
    class NameComparator implements Comparator<Employee>{
        public int compare(Employee emp1, Employee emp2){
            //parameter are of type Object, so we have to downcast it to
Employee objects
            String emp1Name = ((Employee)emp1).getName();
            String emp2Name = ((Employee)emp2).getName();

            //uses compareTo method of String class to compare names of
the employee
            return emp1Name.compareTo(emp2Name);
        }
    }
}
```

```
// CLASSE DE TESTS
/*
This Java comparator example compares employees on the basis of
their age and name and sort them in that order.
*/

public class JavaComparatorExample{

      public static void main(String args[]){
            //Employee array which will hold employees
            Employee employee[] = new Employee[2];

            //set different attributes of the individual employee.
            employee[0] = new Employee();
            employee[0].setAge(40);
            employee[0].setName("Joe");

            employee[1] = new Employee();
            employee[1].setAge(20);
            employee[1].setName("Mark");

            System.out.println("Order of employee before sorting is");
            //print array as is.
            for(int i=0; i < employee.length; i++){
                  System.out.println( "Employee " + (i+1) + " name :: " +
employee[i].getName()
                                                            + ", Age :: " +
employee[i].getAge());
            }

      //Sorting array on the basis of employee age by passing AgeComparator
      // approche utilisant la classe AgeComparator
      // le code suivant est jugé trop lourd pour un utilisateur
      // Employee x = new Employee();
      // Arrays.sort(employee, x.new AgeComparator());
      // on  lui offre le sucre syntaxique suivant
            Employee.sortAge(employee);


            System.out.println("\n\nOrder of employee after sorting by employee
age is");
            for(int i=0; i < employee.length; i++){
                  System.out.println( "Employee " + (i+1) + " name :: " +
employee[i].getName() + ", Age :: " + employee[i].getAge());
            }

      //Sorting array on the basis of employee Name by passing NameComparator
      // approche utilisant la classe NameComparator
            Employee.sortName(employee);


            System.out.println("\n\nOrder of employee after sorting by employee
name is");
            for(int i=0; i < employee.length; i++){
                  System.out.println( "Employee " + (i+1) + " name :: " +
employee[i].getName() + ", Age :: " + employee[i].getAge());
            }
      }
}
```

```
OUTPUT of the above given Java Comparable Example would be :

Order of employee before sorting is
Employee 1 name :: Joe, Age :: 40
Employee 2 name :: Mark, Age :: 20

Order of employee after sorting by employee age is
Employee 1 name :: Mark, Age :: 20
Employee 2 name :: Joe, Age :: 40

Order of employee after sorting by employee name is
Employee 1 name :: Joe, Age :: 40
Employee 2 name :: Mark, Age :: 20
```

IV Classe imbriquée anonyme (exemple de chez Sun)

Les classes imbriquées "ordinaires" sont lourdes à utiliser d'où le recours à un sucre syntaxique via des méthodes statiques.

De plus elles nécessitent leur instanciation à chaque appel à la méthode Arrays.sort.

Une autre approche bien plus agréable est celle des classes imbriquées anonymes définies à l'intérieur d'une interface de constantes.

Le sucre syntaxique est à forte dose et conduit à un confort de programmation très appréciable.

```
-
package comparator;
import java.util.Comparator;

// OBSERVER BIEN LA CLASSE ANONYME IMBRIQUEE
// static final ==> constante ==> une et une seule instanciation de ces 
comparateurs

public interface ComparatorEmployee {

static final Comparator<Employee> AGE_ORDER = new Comparator<Employee>() {
            public int compare(Employee emp1, Employee emp2) {
                  …....
            }
      };

static final Comparator<Employee> NAME_ORDER = new Comparator<Employee>() {
            public int compare(Employee emp1, Employee emp2){
                  …....
            }
      };
}
```

Le code utilisateur s'écrit de la manière simple suivante:

```
// approche utilisant un comparateur statique unique
Arrays.sort(employee, ComparatorEmployee.AGE_ORDER);

// approche utilisant un comparateur statique unique
Arrays.sort(employee, ComparatorEmployee.NAME_ORDER);
```