Conception orientée objet

Application du cours 2 : Musée

Le sujet permet de nous entrainer à utiliser :
- des interfaces notamment en utilisant le design pattern « bridge »,
- la création, la levée et le traitement d'exception.

I. Environnement du problème

Comme pour la première application de cours nous travaillons sur la vie des habitants de l'Armoricae d'après l'œuvre de René Goscinny, Albert Uderzo et Jean-Yves Ferri : « Astérix ». Nous étudierons dans cette application le stockage des trophées des Gaulois dans leur musée « Museum ».

II. Les classes vues dans le cours

1. Package musee

Interface GestionTrophee

```
public interface GestionTrophee {
  void ajouterTrophee(Gaulois proprietaire, Equipement trophee);
  String tousLesTrophees();
  String lesTrophees(Gaulois proprietaire);
}
```

Classe Musee

```
public class Musee implements ActionListener {
  private String nom;
  private int tarif;
  private GestionTrophee gestionnaireTrophee;

  public Musee(String nom, GestionTrophee gestionnaireTrophee) {
    this.nom = nom;
    this.gestionnaireTrophee = gestionnaireTrophee;
  }
  public String getNom() {
    return nom;
  }
  public void setTarif(int tarif) {
    this.tarif = tarif;
  }
  public int getTarif() {
    return tarif;
  }
  public void ajouterTrophee(Gaulois proprietaire,
                             Equipement trophee) {
    gestionnaireTrophee.ajouterTrophee(proprietaire, trophee);
  }
  public String tousLesTrophees() {
    return gestionnaireTrophee.tousLesTrophees();
  }
```

```
  public String lesTrophees(Gaulois proprietaire) {
    return gestionnaireTrophee.lesTrophees(proprietaire);
  }
}
```

Classe GoudurixGestion

```
public class GoudurixGestion implements GestionTrophee {
  private Gaulois[] proprietaires = new Gaulois[30];
  private Equipement[] trophees = new Equipement[30];
  private int nombreDeTrophee = 0;

  public void ajouterTrophee(Gaulois proprietaire, Equipement trophee){
    proprietaires[nombreDeTrophee] = proprietaire;
    trophees[nombreDeTrophee] = trophee;
    nombreDeTrophee++;
  }

  public String lesTrophees(Gaulois proprietaire) {
    String leTrophee = "Le trophée de " + proprietaire.getNom() + " est ";
    int indiceProprietaire = 0;
    for (int i = 0; i < nombreDeTrophee; i++) {
      if (proprietaire.equals(proprietaires[i])) {
        indiceProprietaire = i;
      }
    }
    leTrophee += trophees[indiceProprietaire];
    return leTrophee;
  }

  public String tousLesTrophees() {
    String tousLesTrophees = "Tous les trophées du musée sont :\n";
    for (int i = 0; i < nombreDeTrophee; i++) {
      tousLesTrophees += "- " + trophees[i] + "\n";
    }
    return tousLesTrophees;
  }
}
```

Classe RenseignementTrophee

```
public class RenseignementTrophee {
  private Gaulois proprietaire;
  private Equipement trophee;

  public RenseignementTrophee(Gaulois proprietaire,
                              Equipement trophee) {
    this.proprietaire = proprietaire;
    this.trophee = trophee;
  }

  public Gaulois getProprietaire() {
    return proprietaire;
  }

  public Equipement getTrophee() {
    return trophee;
  }
}
```

Classe KeskonrixGestion

```
public class KeskonrixGestion implements GestionTrophee {
  private RenseignementTrophee[] trophees =
    new RenseignementTrophee[30];
  private int nombreDeTrophee = 0;

  public void ajouterTrophee(Gaulois proprietaire,
        Equipement trophee) {
    trophees[nombreDeTrophee] =
      new RenseignementTrophee(proprietaire, trophee);
    nombreDeTrophee++;
  }

  public String lesTrophees(Gaulois proprietaire) {
    String tousLesTrophees = "Les trophées de " +
      proprietaire.getNom() + " sont :\n";
    for (int i = 0; i < nombreDeTrophee; i++) {
      Gaulois proprietaireTrophee = trophees[i].getProprietaire();
      if (proprietaire.equals(proprietaireTrophee)) {
        Equipement typeEquipement = trophees[i].getTrophee();
        tousLesTrophees += "- " + typeEquipement + "\n";
      }
    }
    return tousLesTrophees ;
  }

  public String tousLesTrophees() {
    String tousLesTrophees = "Tous les trophées du musée sont :\n";
    for (int i = 0; i < nombreDeTrophee; i++) {
      Equipement typeEquipement = trophees[i].getTypeTrophee();
      tousLesTrophees += "- " + typeEquipement + "\n";
    }
    return tousLesTrophees;
  }
}
```

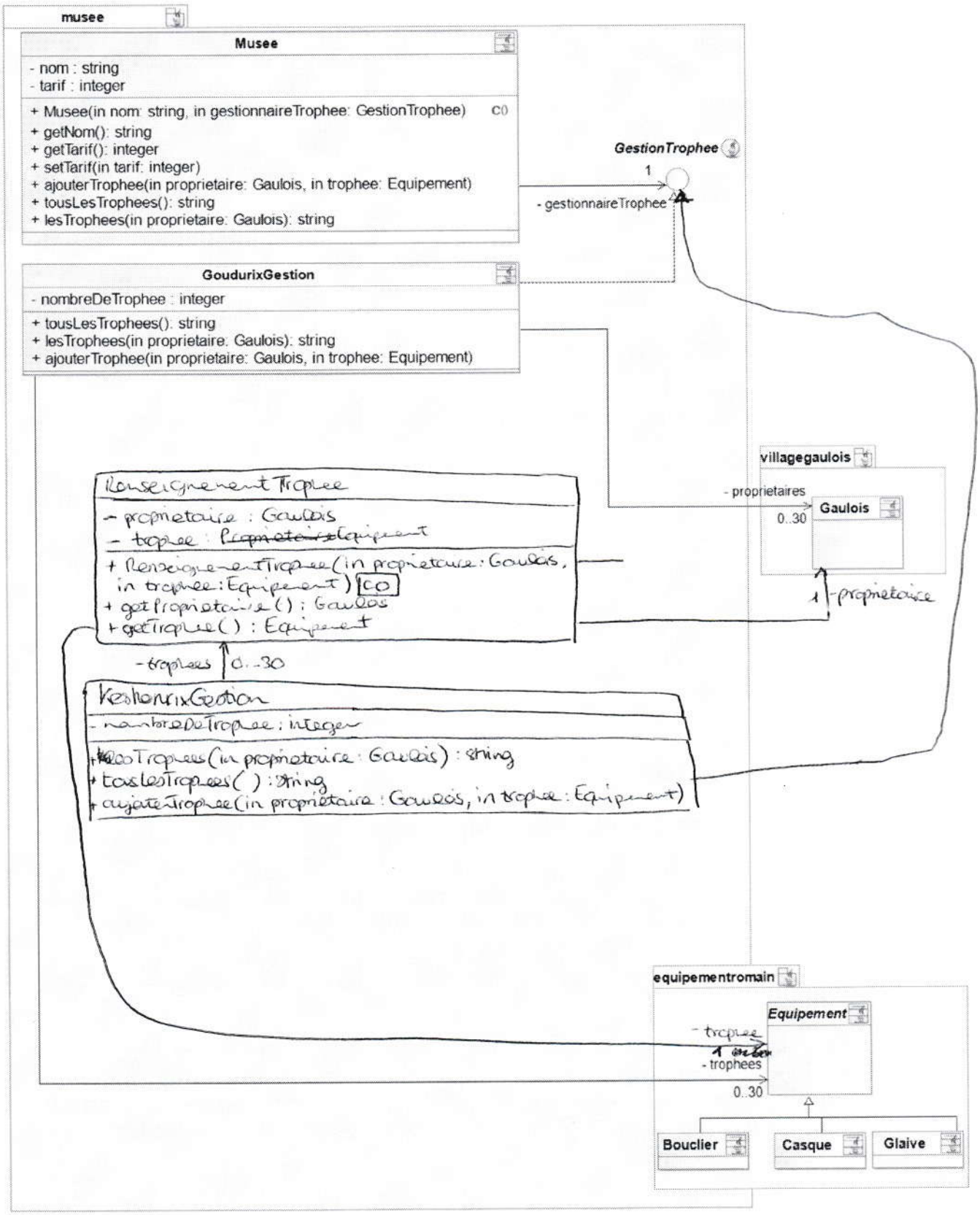
III. Enoncé du TD

1. Diagramme de classes de la solution de Keskonrix

Les gaulois souhaitent pouvoir consulter leurs trophées (équipement d'un soldat romain). Il faut donc qu'une entité rende les services suivants :
- ajouter un trophée,
- récupérer l'ensemble des trophées du village,
- récupérer l'ensemble des trophées apportés par un gaulois en particulier.

Ces services sont repris dans l'interface « *GestionTrophee* ».
Keskonrix, un aspirant à devenir informaticien (non confirmé du tout), a apporté l'implémentation «KeskonrixGestion» à l'interface. Il a créé la classe « RenseignementTrophee » lui permettant d'associer un propriétaire à un trophée.

Modifier le diagramme de classe ci-dessous pour prendre en compte la solution de Keskonrix.



Modifier si besoin la classe « Test » ci-dessous.

```
public class Test {
  public static void main(String[] args) {
    Gaulois asterix = new Gaulois("Astérix");
    Gaulois obelix = new Gaulois("Obélix");
    Gaulois abraracourcix = new Gaulois("Abraracourcix");

    Bouclier bouclierMordicus = new Bouclier("bon état");
    Casque casqueAerobus = new Casque("cabossé", "fer");
    Glaive glaiveCornedurus = new Glaive("cassé");
    Glaive glaiveAerobus = new Glaive("tordu");
    Casque casqueHumerus = new Casque("bon état", "fer");

    GestionTrophee gestionnaireTrophees = new GoudurixGestion();
    Musee musee = new Musee("Museum", gestionnaireTrophees);
    musee.ajouterTrophee(asterix, bouclierMordicus);
    musee.ajouterTrophee(asterix, casqueAerobus);
    musee.ajouterTrophee(asterix, glaiveCornedurus);
    musee.ajouterTrophee(obelix, glaiveAerobus);
    musee.ajouterTrophee(abraracourcix, casqueHumerus);
    System.out.println("Le musé " + musee.getNom()
      + " est ouvert !\n");
    System.out.println(musee.tousLesTrophees());
    System.out.println(musee.lesTrophees(asterix));
  }
}
```

Modifier si besoin l'extrait de la classe « Musee » ci-dessous.

```
public class Musee {
  private String nom;
  private int tarif;
  private GestionTrophee gestionnaireTrophee;

  public Musee(String nom, GestionTrophee gestionnaireTrophee) {
    this.nom = nom;
    this.gestionnaireTrophee = gestionnaireTrophee;
  }

  ...
}
```

2. *Les exceptions*

Dans le village gaulois, on organise régulièrement des festins. Pour les financer, on propose des activités payantes.
Créer l'interface « ActivitePayante » ayant deux méthodes : *getTarif* et *setTarif*.

Interface ActivitePayante

```
public interface ActivitePayante {
    int getTarif();
    void setTarif(int tarif);
}
```

Reprendre la classe « Musee » afin qu'elle implémente cette interface.

D'autres activités sont proposées : lancer de menhir, bataille de poissons (pas frais), chasse aux sangliers ...
Chacune de ses activités est représentée par une classe qui implémente l'interface « ActivitePayante ».

Nous proposons la classe « Billetterie » qui permet d'acheter des places à une activité donnée.

```
public class Billeterie {
  private int buttin = 0;
  private ActivitePayante[] activites = new ActivitePayante[4];
  private int nbActivite = 0;

  public void ajouterActivite(ActivitePayante activitePayante,
      int prixBillet) {
    activites[nbActivite] = activitePayante;
    activitePayante.setTarif(prixBillet);
  }

  public void acheterBillet(int numeroActivite){
    ActivitePayante activite = activites[numeroActivite];
    buttin += activite.getTarif();
  }

  public int getButtin() {
    return buttin;
  }
}
```

Dans la classe « Billetterie », lorsque l'on achète une place, l'utilisateur fournit le gaulois et le numéro de l'activité, celui-ci ne peut malheureusement correspondre à aucune activité.
Admettons maintenant que l'utilisateur ne puisse acheter des billets que pour activité existante.

Il clique sur un personnage puis sur une activité (l'affichage étant réalisé qu'à partir des activités existantes dans le tableau activites de la classe « Billeterie »). **Il est donc impossible à l'utilisateur de se tromper.**

Donc lorsqu'on appelle la méthode *acheterBillet* le numéro de l'activité est forcément valide par rapport au fonctionnement normal de l'application. Le contraire serait donc une exception.

Laissons apparaitre l'exception.

```
public static void main(String[] args) {
   Billeterie billeterie = new Billeterie();
   billeterie.acheterBillet(1);
}
```

L'exception levée est : java.lang.NullPointerException

Traiter l'exception :

```
public void acheterBillet(int numeroActivite) {
try {
   ActivitePayante activite = activites[numeroActivite];
   bilan += activite.getTarif();
} catch(NullPointerException e) {
   System.out.println("Exception : Tentative d'acheter un
   billet pour une activité inexistante");
} catch(TarifNonInitialiseException e){
   System.out.println("Exception : Tarif non initialisé");
}

}
```

Normalement, dans la classe « Billetterie », le tarif de l'activité est affecté lors de son ajout dans le tableau activites (méthode *ajouterActivite*).
Donc lorsqu'on appelle la méthode *acheterBillet* le tarif de l'activité a forcément été initialisé par rapport au fonctionnement normal de l'application. Le contraire serait donc une exception.

Ecrire l'exception personnalisée « TarifNonInitialiserException ».

```
public class TarifNonInitialiserException extends RuntimeException{
    private static final long serialVersionUID = 1L;
    public TarifNonInitialiserException(){
        super();
    }
    public TarifNonInitialiserException(String message){
        super(message);
    }
    public TarifNonInitialiserException(Throwable cause){
        super(cause);
    }
    public TarifNonInitialiserException(String message, Throwable cause){
        super(message, cause);
    }
}
```

Lever l'exception personnalisée dans la méthode *getTarif* de la classe « Musee ». Rappel : un entier primitif est initialisé à 0 par défaut.

```
public int getTarif() throws TarifNonInitialiserException
    if(tarif!=0){
        return tarif;
    }
    else throw new TarifNonInitialiserException();
}
```

Modifier la méthode *acheterBillet* de la classe « Billetterie » en conséquence.

Conception orientée objet

Application du cours 5 : Marché aux poissons et comparaison de gaulois

---

Cet énoncé vient en appui des diapositives du Cours.

I. Comparaison d'objets

1. Ordre naturel

a) Implémenter l'interface « *Comparable* » afin que les poissons puissent être triés selon leur date de pêche puis s'ils ont la même date, selon leur poids. Deux poissons seront considérés identiques si leurs deux attributs sont identiques.

```
public class Poisson implements Comparable {
  private Date datePeche;
  private float poids;

  public Poisson(Date datePeche, float poids) {
    this.datePeche = datePeche;
    this.poids = poids;
  }
  public int compareTo(Poisson poisson) {
      int comparaison = datePeche.compareTo(poisson.datePeche);
      if(comparaison !=0){return comparaison;}
      float difference = poids - poisson.poids;
      if(difference<0){return -1;}
      if difference>0 {return 1;}
      return 0;
  }
  public boolean equals(Object objet){
     if(objet instanceof Poisson){
        Poisson poisson = (Poisson) objet;
        return datePeche.equals(poisson.datePeche) && poids ==
              poisson.poids;
     }
  }
}
```

b) Ecrire les méthodes nécessaires à votre implémentation précédente.

```
public class Date implements Comparable<Date> {
   int annee;
   int mois;
   int jour;

   public Date(int jour, int mois, int annee) {
      this.jour = jour;
      this.mois = mois;
      this.annee = annee;
   }

   public boolean equals( Object objet ) {
      if(objet instanceof Date){
         Date date = (Date) objet;
         return jour == date.jour
            && mois == date.mois
            && annee == date.annee;
         }
      return false
   }

   public int compareTo( Date dateToCompare ) {
    int difference = jour - date.jour;
      if (difference == 0)
         int difference mois = date.mois;
         if(difference == 0) { int difference = annee - date.annee;
      return difference;
   }
}
```

2. *Classe « Gaulois »*

```
public class Gaulois implements Comparable<Gaulois> {

   private String nom;
   private int age;

   public Gaulois(String nom, int age) {
      this.nom = nom;
      this.age = age;
   }

   public boolean equals(Object obj) {
      if (obj instanceof Gaulois) {
         Gaulois gaulois = (Gaulois) obj;
         return nom.equals(gaulois.nom);
      }
      return false;
   }

   public String toString() {
      return nom + " à " + age + " ans";
   }

   public int compareTo(Gaulois gauloisToCompare) {
      return nom.compareTo(gauloisToCompare.nom);
   }
}
```

3. *TreeSet*

Dans le main d'une classe « Test » nous avons créé les objets suivants :

```
Gaulois asterix = new Gaulois("Astérix", 35);
Gaulois obelix = new Gaulois("Obélix", 30);
Gaulois abraracourcix = new Gaulois("Abraracourcix", 40);
Gaulois bonemine = new Gaulois("Bonemine", 36);
Gaulois panoramix = new Gaulois("Panoramix", 90);
```

Puis nous les avons ajoutés dans un **TreeSet** ensemble :

```
NavigableSet<Gaulois> ensemble = new TreeSet<>();
ensemble.add(asterix);
ensemble.add(obelix);
ensemble.add(abraracourcix);
ensemble.add(bonemine);
ensemble.add(panoramix);
```

Le main est complété par :

System.***out***.println("Tri par _____");
System.***out***.println(ensemble);

Par quelle fin de phrase doit-on compléter la sortie écran (« Tri par ... ») pour donner l'ordre de l'affichage de l'ensemble des gaulois ensemble

Ecrire la méthode *afficherGaulois* dans la classe **Test** qui affiche chacun des gaulois contenu dans le **TreeSet** :
- en retournant à la ligne entre chaque gaulois,
- en utilisant un itérateur.

public static void afficherGaulois(NavigableSet<Gaulois> ensemble) {

```
for(Iterator<Gaulois> it=ensemble.iterator();
    it.hasNext();){
Gaulois gaulois=it.next();
System.out.println(gaulois);

}
```

4. Ordre impose

Créer un ensemble ensembleGaulois qui sera ordonné du plus ancien au plus jeune et s'ils ont le même âge selon l'ordre alphabétique.

```
public TreeSet(Comparator<? super E> c){
    NavigableSet<Gaulois> ensembleGaulois =
    new TreeSet<>(
        new Comparator<Gaulois>(){
            public int compare(Gaulois gaulois1, Gaulois gaulois
            int comparaison = gaulois2.getAge()-gaulois1.getAge();
            if(comparaison == 0) comparaison = gaulois1.compareTo
            (gaulois2);
            return comparaison;
    });
```

Ajouter l'ensemble des Gaulois de la collection ensemble, ainsi que deux nouveaux gaulois : Ordralfabétix et Cétautomatix qui ont tous deux 41 ans.

```
ensembleGaulois.addAll(ensemble);
ensembleGaulois.add(new Gaulois("Ordralfabétix", 41));
ensembleGaulois.add(new Gaulois("Cétautomatix", 41));
System.out.println("Affichage des villageois");
afficherGaulois(ensembleGaulois);
```

ceiling ⇒ (>=) plus grand ou égal
floor ⇒ (>) plus grand strict
lower ⇒ (<) plus petit strict

II. Les ensembles triés

1. *NavigableSet*

Dans cette partie seules les méthodes de l'interface **NavigableSet** devront être utilisées. Avec ensembleGaulois = {Bonnemine à 36 ans, Astérix à 35 ans, Obélix à 30 ans}

a) Donner le premier gaulois qui a plus de 35 ans.

~~ensemble.lower(new Gaulois("",35));~~

System.Out.println(ensemble.lower(new Gaulois("AA",35)));

b) Donner le premier gaulois qui a 35 ans au moins.

System.out.println(ensemble.floor(newGaulois("ZZ",35)));

c) Que changer dans la méthode *afficherGaulois* écrite précédemment pour afficher les gaulois en ordre inverse ?

Modifier : ensemble.iterator();

Par : ensemble.descendingIterator();

2. *Les vues*

Créer et afficher la vue selection correspondant aux gaulois ayant plus de 40 ans (40 ans exclus).

NavigableSet<~~Integer~~ Gaulois> selection = ensembleGaulois.headSet(new Gaulois("",40), false);

afficherGaulois(selection);

Créer et afficher l'ensemble ensembleSelection aux gaulois ayant plus de 40 ans (40 ans exclus).

NavigableSet<Gaulois> ensembleSelection = new TreeSet<>(selection);

afficherGaulois(~~selection~~ ensembleSelection);

L'affichage de la vue selection et l'affichage de l'ensemble ensembleSelection est le même :

```
[Panoramix à 90 ans, Cétautomatix à 41 ans, Ordralfabétix à 41 ans]
```

On ajoute le père d'Obélix « Obélodalix » à l'ensemble des gaulois trié par âge :

```
ensembleGaulois.add(new Gaulois("Obélodalix",62));
```

Quel est l'affichage correspondant aux instructions suivantes :
System.*out*.println("Affichage Selection :");
System.*out*.println(selection);

Affichage Selection :
[Panoramix à 90 ans, Obélodalix à 62 ans, Cétautomatix à 41 ans, Ordralfabétix à 41 ans]

Quel est l'affichage correspondant aux instructions suivantes :
System.*out*.println("Affichage Ensemble :");
System.*out*.println(ensembleSelection);

Affichage Ensemble :
[Panoramix à 90 ans, Cétautomatix à 41 ans, Ordralfabétix à 41 ans]

III. TreeSet (extrait)

Constructor and Description
TreeSet() Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
TreeSet(Collection<? extends E> c) Constructs a new tree set containing the elements in the specified collection, sorted according to the *natural ordering* of its elements.
TreeSet(Comparator<? super E> comparator) Constructs a new, empty tree set, sorted according to the specified comparator.

Modifier and Type	Method and Description
boolean	add(E **e**) Adds the specified element to this set if it is not already present.
boolean	addAll(Collection<? extends E> **c**) Adds all of the elements in the specified collection to this set.
E	ceiling(E **e**) Returns the least element in this set greater than or equal to the given element, or **null** if there is no such element.
void	clear() Removes all of the elements from this set.
boolean	contains(Object **o**) Returns **true** if this set contains the specified element.
Iterator<E>	descendingIterator() Returns an iterator over the elements in this set in descending order.
E	first() Returns the first (lowest) element currently in this set.
E	floor(E **e**) Returns the greatest element in this set less than or equal to the given element, or **null** if there is no such element.
NavigableSet<E>	headSet(E **toElement, boolean inclusive**) Returns a view of the portion of this set whose elements are less than (or equal to, if **inclusive** is true) **toElement.**
E	*higher*(*E* **e**) Returns the least element in this set strictly greater than the given element, or **null** if there is no such element.
boolean	*isEmpty*() Returns **true** if this set contains no elements.
Iterator<E>	*iterator*() Returns an iterator over the elements in this set in ascending order.
E	*last*() Returns the last (highest) element currently in this set.
E	*lower*(*E* **e**) Returns the greatest element in this set strictly less than the given element, or **null** if there is no such element.
E	*pollFirst*() Retrieves and removes the first (lowest) element, or returns **null** if this set is empty.
E	*pollLast*() Retrieves and removes the last (highest) element, or returns **null** if this set is empty.
int	*size*() Returns the number of elements in this set (its cardinality).
NavigableSet<E>	*subSet*(*E* **fromElement, boolean fromInclusive,** *E* **toElement, boolean toInclusive**) Returns a view of the portion of this set whose elements range from **fromElement** to **toElement.**
NavigableSet<E>	*tailSet*(*E* **fromElement, boolean inclusive**) Returns a view of the portion of this set whose elements are greater than (or equal to, if **inclusive** is true) **fromElement.**

1. *Methods inherited from class java.util.*AbstractSet
equals, hashCode, removeAll
2. ***Methods inherited from class java.util.***AbstractCollection
containsAll, retainAll, toArray, toArray, toString
3. ***Methods inherited from class java.lang.***Object
finalize, getClass, notify, notifyAll, wait, wait, wait
4. ***Methods inherited from interface java.util.***Set
containsAll, equals, hashCode, removeAll, retainAll, toArray, toArray