# Illustration de l'usage des interfaces
# Tri
# Sérialisation

Conception par Objets 1, HLIN406

6 mars 2017

# Sommaire

Quizz

Comparaison et tris

Sérialisation

# Interface IPassager

On représente les passagers d'une compagnie aérienne par l'interface suivante

```
public interface IPassager {
      String nomPrenom(){}
      String numeroPlace(){}
      String programmeFidelite(){}
}
```

Commentez ...

# Interface IBouteille

On représente les bouteilles par l'interface suivante

```
public interface IBouteille {
      double volumeMax;
      double volume();
}
```

Commentez ...

# Interface IBouteille

On représente les bouteilles par l'interface et la classe suivante

```
public interface IBouteille {
     double volumeMax;
     double volume();
}

public class BouteilleEau extends IBouteille{
...
}
```

Commentez ...

# Interface IProduit

On représente les produits d'un magasin par l'interface et la classe suivantes

```
public interface IProduit {
     double tauxTVA = 19.6;
     double prix();
}
public class Produit implements IProduit
{
     private double prix;
     public Produit(double prix) {this.prix = prix;}
     public double getPrix() {return prix;}
     public void setPrix(double prix) {this.prix = prix;}
}
```

Commentez ...

# Interface IProduit

On représente des jardins par l'interface et la classe suivantes

```
public interface Ijardin {
     double surface();
     String nomProprietaire();
     String coordGPS();}

public class Jardin implements Ijardin{
// ...
     public double surface(){return 0;}
     public String nomProprietaire(){return "inconnu";}
     public String coordGPS(){return "coordInconnues";}
     public static void main(String[] arg){
          Ijardin jardin1 = new IJardin();
          Ijardin jardin2 = new Jardin();
          Jardin jardin3 = new Jardin();
}}
```

Commentez ...

# Interface Ifigure

On représente des figures géométriques et des cercles par les interfaces suivantes

```
public interface IFigureGeometrique {
      int coordX(); int coordY();
      void dessiner();
      void deplacer(int x, int y);
      boolean memeCoordonneesQue(IFigureGeometrique f);
}

public interface ICercle extends IFigureGeometrique {
      double rayon();
      boolean memeCoordonneesQue(ICercle f);
}
```

Commentez ...

# Interface Ifigure

On représente des figures géométriques et des cercles par l'interface et la classe suivantes

```
public interface IFigureGeometrique {
      int coordX(); int coordY();
      void dessiner();
      void deplacer(int x, int y);
      boolean memeCoordonneesQue(IFigureGeometrique f);
}

abstract class Cercle implements IFigureGeometrique {
      public abstract double rayon();
      public boolean memeCoordonneesQue(Cercle f)
                {/* ...  */ return true;}
}
```

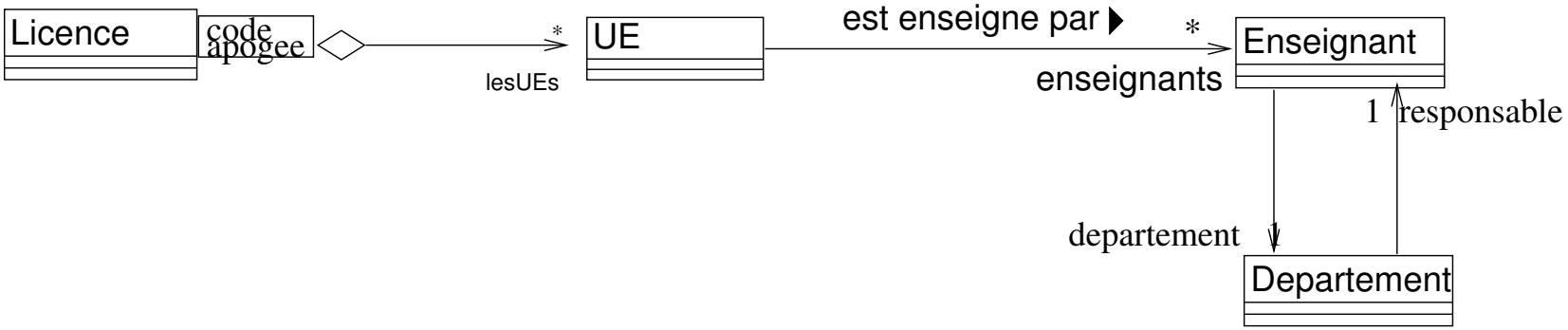
# Sommaire

Quizz

Comparaison et tris

Sérialisation

# Etude de cas Licence, UE, Enseignant, Département



# Quelques éléments sur la classe Licence

```
public class Licence{
    private String specialite;
    private int nbInscriptions;
    private Hashtable<String,UE> lesUEs
        =new Hashtable<String,UE>();
        // ...
    public UE getUEByCode(String codeApogee){
        return lesUEs.get(codeApogee);}
    public void ajoutUE(UE ue){
        lesUEs.put(ue.getCodeApogee(), ue);}
}
```

# Quelques éléments sur la classe UE

```
public class UE{
    private String codeApogee;
    private String nom;
    private Vector<Enseignant> enseignants
           = new Vector<Enseignant>();

...
}
```

# Quelques éléments sur la classe Enseignant

```
public class Enseignant{
    private String nom;
    private String prenom;
    private int anneeRecrutement;
    private Departement departement;
//.....
```

# Quelques éléments sur la classe Département

```
public class Departement
{
private String nom;
private Enseignant responsable;

//....
}
```

# Questions de tris

- Comparer les départements d'enseignement selon :
  - leur nom
- Trier les enseignants d'une UE selon :
  - leur nom
  - leur ancienneté
  - leur département

# Trier les enseignants d'une UE selon un critère

- Savoir comparer les enseignants selon ce critère
- Trier le vecteur `enseignants` de la classe `UE`

# Trier Collections.sort

- `public static void sort(List<T> list)`
- 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.

```
public Interface Comparable<T>{
int compareTo(T o) }
// ...
e1.compareTo(e2)
// Returns a negative integer, zero, or a positive integer
// as this object is
// less than, equal to, or greater than the specified object.
```

# Trier les enseignants par leur nom

- Donner un ordre naturel aux enseignants d'après l'ordre lexicographique de leur nom
- Utilise le fait qu'il existe un ordre naturel sur les String (par implémentation de `Comparable` et de `compareTo`)

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

   public int compareTo(Enseignant e){
      return this.getNom().compareTo(e.getNom());}
}
```

# Trier les enseignants par leur nom

- Appeler la méthode `sort` qui utilise cet ordre naturel
- Le vecteur `enseignants` est modifié par le tri

```
public class UE{
//....
   public void trieEnseignantParNom()
      {Collections.sort(enseignants);}
}
```

# Trier les enseignants par leur ancienneté

- Ecrire un nouvel ordre naturel ?
- Mais on ne peut le faire qu'en réécrivant la méthode `compareTo` de `Enseignant` (on perd l'autre)
- solution : écrire une classe `Comparator` spécialisée dans la comparaison des enseignants suivant l'ancienneté

# Trier avec un comparateur Collections.sort

- public static void sort(List<T> list, Comparator<T> c)
- Sorts the specified list according to the order induced by the specified comparator.

```
public Interface Comparator<T>{
   int compare(T o1, T o2)
   // Compares its two arguments for order.
   boolean  equals(Object obj)
   // Indicates whether some other object is "equal to"
   // this comparator.
}
```

# Comparateur d'enseignants par ancienneté

```
public class ComparatorEnseignantParAnciennete
      implements Comparator<Enseignant>
{
      // retourne un nb > 0 si e1 est plus ancien que e2
      // 0 si même ancienneté
      // un nb < 0 si e1 est moins ancien que e2
      public int compare(Enseignant e1,Enseignant e2){
            return e1.getAnneeRecrutement()
                  - e2.getAnneeRecrutement();}
}
```

# Tri des enseignants par ancienneté

```
public class UE{
//....
public void trieEnseignantParAnciennete(){
   Collections.sort(enseignants,
       new ComparatorEnseignantParAnciennete());}
}
```

# Tri des enseignants par département

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

public int compareTo(Departement d)
       // sur le nom du département
{
   return this.getNom().compareTo(d.getNom());
}

}
```

# Tri des enseignants par département

```
public class CompareEnseignantParDepartement
     implements Comparator<Enseignant>
{
    public int compare(Enseignant e1,Enseignant e2){
        int resultat=e1.getDepartement()
             .compareTo(e2.getDepartement());
        if (resultat==0){ // même département
            resultat=e1.compareTo(e2);
        }
        return resultat;
    }
}
```

# Tri des enseignants par département

```
public class UE{
//....
public void trieEnseignantParDepartement(){
   Collections.sort(enseignants,
      new CompareEnseignantParDepartement());
}
```

# Tri des enseignants par tout critère passé en paramètre sous forme d'un comparateur

```
public class UE{
//....
public void trieEnseignantParCritere
       (Comparator<Enseignant> critere){
              Collections.sort(enseignants, critere);
}
```

# Appel des méthodes de tri

```
public static void main(String[] args){
    Departement info = new Departement("Informatique");
    Departement math = new Departement("Mathematique");
    Licence l=new Licence();
    l.setSpecialite("info");
    UE ue=new UE();
    ue.setCodeApogee("ULIN407");
    ue.setNom("Conception et programmation par objets");
    Enseignant mh=new Enseignant("Huchard", "M",info, 1992);
    Enseignant cn=new Enseignant("Nebut", "C",info, 2005);
    Enseignant mc=new Enseignant("Cuer", "M",math, 2000);
    ue.addEnseignant(mh);
    ue.addEnseignant(mc);
    ue.addEnseignant(cn);
    l.ajoutUE(ue);
// ....
 }
```

# Appel des méthodes de tri

```
 public static void main(String[] args){
     // ....

System.out.println("Liste triee par dept ------");
ue.trieEnseignantParDepartement();

System.out.println("Liste triee par anciennete ------");
ue.trieEnseignantParAnciennete();

System.out.println("Liste triee par nom ------");
ue.trieEnseignantParNom();

System.out.println("Liste triee par anciennete
          en passant le critère ------");
ue.trieEnseignantParCritere
         (new ComparatorEnseignantParAnciennete());
  }
```

# Sommaire

Quizz

Comparaison et tris

Sérialisation

# Flot

- Flot = objet capable de transférer une suite de données
  - d'une source externe (ex. fichier, clavier) vers le programme
  - du programme vers une cible externe (ex. fichier, écran)
- Flots en Java - plusieurs dizaines de classes
  - flots binaires (octets)
  - flots de caractères (texte)
    - avec transformation
    - UNICODE -> format texte de la machine hôte

# Fichier texte, exemple

```
public static void main(String[] arg) throws IOException{
  BufferedReader fc = new BufferedReader
    (new InputStreamReader (System.in));

  BufferedWriter ff = new BufferedWriter
    (new FileWriter ("essai2015.txt"));

  System.out.println
              ("Entrez des lignes (Return pour terminer)");
  String s = fc.readLine();
  while (s.length() != 0)
    {
    ff.write(s);
     ff.newLine();
    s = fc.readLine();
  }
  ff.close();
}
```

# Fichier d'objets

- la classe de l'objet à sauvegarder et de ses sous-objets doit implémenter l'interface `java.io.Serializable`
- l'objet est décomposé selon ses attributs jusqu'au niveau où les données sont de type primitif ou String

Par exemple on veut sauver les informations d'une licence

```
public class Licence implements Serializable{ ...
public class UE implements Serializable{ ...
public class Enseignant
       implements Comparable<Enseignant>, Serializable{ ...
public class Departement
       implements Comparable<Departement>, Serializable{ ...
```

# Fichier d'objets

Le fait d'implémenter `Serializable` autorise à utiliser :

- java.io.ObjectOutputStream
- public final void writeObject(Object obj) throws IOException

- java.io.ObjectInputStream
- public final Object readObject() throws OptionalDataException, ClassNotFoundException, IOException

# Fichier d'objets - sauvegarde

```
 public static void main(String[] args)
      throws FileNotFoundException, IOException{
{
// ....
  ObjectOutputStream flotS =
              new ObjectOutputStream
                    (new FileOutputStream ("essai2015.txt"));
  flotS.writeObject(l);
  flotS.close();
}
```

# Fichier d'objets - récupération

```
 public static void main(String[] args)
      throws FileNotFoundException, IOException,
           ClassNotFoundException
{
// ....
  ObjectInputStream flotE =
              new ObjectInputStream
                    (new FileInputStream ("essai2015.txt"));
  Object lic = flotE.readObject();
  Licence licRecuperee = (Licence) lic;
  flotE.close();
}
```

# Fichiers d'objets

- automatique (il suffit de déclarer que les classes implémentent `Serializable`)
- permet de gérer facilement des objets complexes
- les fonctions readObject et writeObject peuvent être redéfinies dans la classe (Licence par exemple) pour modifier ce qui est stocké
- inconvénient : les fichiers ne sont pas lisibles (binaires)