Fiche de Java

Quelques classes remarquables de JAVA

La class Object

Classe principale (d'ou toute les autres classes descendent)

Elle contient les methodes suivantes :*

- `protected Object clone()`
  - créée et retourne une copie de l'objet
- `boolean equals(Object obj)`
  - determine si l'objet est égal à l'objet courant
- `String toString()`
  - retourne une chaîne représentant l'objet
- `protected void finalize()`
  - appelé par le grabage collector si il n'y a plus de reference à l'objet
- `Class getClass()`
  - retourne la classe courrante de l'objet
- `int hashcode()`
  - retourne une clée pouvant être utilisée pour un tri
- `void notify()`
  - réveille un processus en attente sur l'objet
- `void notifyAll()`
  - réveille tout le processus en attente
- `void wait()`
  - met en pause le processus courant en attendant un reveil

Comparer des objets

`int compareTo(T other)` compare l'objet other à l'objet courant et retourne

- **< 0** si this < other
- **0** si this = other
- **> 0** si this > other

(la classe doit ajouter la notion de comparaison)

```
/**Ajout de la notion de comparaison à la classe Personne */
class Personne implements Cloneable, Comparable<Personne>{
    String prenom;int age;
   ...
   /**retourne <0 si objet < autre, 0 si objet == autre, >0 si objet > autre, ici tri par age croissant*/
   publicintcompareTo(Personne other) {
    intretour = 0;
    // les pointeurs null seront en fin de tableau/liste
    if(other==null) retour = -1;
    // sinon, si je suis plus age que other, j'envoie un nb positif;
    // si je suis plus jeune, j'envoie un nb négatif;
    // si on a le meme age, je retourne 0
    else retour = age - other.age;

    return retour;
    }
}
```

Types génériques

Utilisation de caractère remplaçant un type défini à l'exécution.

```
// affiche les objets d'un tableau de T
<T>void affiche(T[] tab){
    for(Object o : tab) System.out.println(o.toString());
}

//T et V sont des types génériques
<T, V>void afficheEtVal(T[] tab, V v){
    for(Object o : tab) System.out.println(o.toString());
    System.out.println("valeur = " + v.toString());
}

//T doit etre un type comparable
<T extends Comparable<T>>void compare(T a, T b){
```

```
    int comp = a.compareTo(b);
    if(comp<0) System.out.println(a + " est plus petit que " + b);
    if(comp == 0) System.out.println(a + " est égal à " + b);
    if(comp>0) System.out.println(a + "est plus grand que " + b)
}


public class Couple<T1,T2>{
    T1 v1;
    T2 v2;
    couple(){}
    couple(T1 _v1, T2 _v2){ v1 = _v1; v2 = _v2;}
    ...
}
```

La classe System

- `static PrintStream err`
  - sortie d'erreur standard
- `static InputStream in`
  - entrée standard
- `static PrintStream out`
  - sortie standard
- `static void arraycopy(...)`
  - copie de tableaux
- `static long currentTimeMillis()
  - temps courant en millisecondes
- `static long nanoTime()`
  - temps courant en nanoseconde
- `static void exit(int status)`
  - sortie de programme
- `static void gc()`
  - lance le ramasse-miettes
- `static void load(String fichier)`
  - charge le code en tant que librairie dynamique
- `static String getProperty(String key)`
  - retourne la valeur de la propriété spécifiée par la clé
- `static Properties getProperties()`
  - retourne les propriétés du système
- `static String setProperty(String key, String value)`
  - affecte une noubelle valeur à une propriété

La classe String

La chaines sont constantes, leur valeurs ne peut etre changées après leurs créations

`StringBuffer` et `StringBuilder` permettent l'utilisation de chaînes "dynamiques"

La classe `String` comporte des méthodes d'accès aux caractères, de comparaison, de recherche, d'extraction, de copie, de conversion majuscule/minuscule, ...

L'opérateur + est surchargé pour permettre la concaténation.

String[] String.split()

Permet de tronçonner la chaine.

StringBuilder

Plus rapide pour les besoin de chaine dynamiques

```
StringBuilder sb;
String sep = " ";
for(int i=0;i<taille;i++) sb.append(i).append(sep);
System.out.println(sb);
```

Contenu de java.util.*

- Interfaces :
  - Collection, Comparator, List, Set
- Classes :
  - ArrayList, Arrays, Collections, Hashtable, Observable, Random, TreeSet

---

La classe Arrays

- `static int binarySearch(int[] tab, int valeur)`
  - retourne l'index de la valeur, -1 si introuvable
- `static boolean equals(boolean[] tab1, boolean[] tab2)`
  - teste l'egalité de deux tableaux
- `static boolean deepEquals(Object[] tab1, Object[] tab2)`
  - teste l'egalité de deux tableau récurivement (elle effectue un appel a elle même si tab1 et tab2 contienne des tableaux)
- `static void fill(double[] tab, double valeur)`
  - remplit le tableau avec la valeur
- `static void sort(long[] tab)`
  - trie le tableau
- `static void sort(Object[] tab)`
  - trie le tableau si les objects contenus sont comparables
- `static String deepToString(Object[] tab)`
- `static boolean setAll(double[] array, IntFunction<? extends T> generator`
  - ```
    double[] tab = newdouble[6];Arrays.setAll(tab, i -> (2d*i));
    System.out.println(Arrays.toString(tab))
    ```

L'interface List

- `boolean add(int index, E e)`
- `boolean addAll(int index, COllection<? extends E> c)`
- `int indexOf(Object o)`
- `lastIndexOf(Object o)`
- `E remove(int index)`
- `E set(int index, E e)`
- `List<E> subList(int from, int to)`
- `void sort(Comparator<? super E> c)`
- `replaceAll(UnaryOperator<E> operator)`
- `static <E> List<E> copyOf(Collection<? extends E> coll)`

La classe ArrayList

La classe ArrayList est une implémentation de l'interface List Elle permet de stocker, trier, modifier des objets dans un ensemble de longueur variable. Elle est non synchronisée

---

La classe LinkedList

La classe `LinkedList` permet l'utilisation en tant que FIFO (First In First Out) ou LIFO (Last In Last Out):

- `push(object)` : ajoute un objet sur la pile
- `pollLast()` : retire l'objet du dessus de la pile
- `pollFirst()` : retire l'objet en bas de la pile
- `peek()` : renvoie une reference à l'objet du bas de la pile
- `peekLast()` : renvoie une reference à l'objet du haut de la pile
- `isEmpty(`

La classe TreeSet

La classe `TreeSet` définit un ensemble ne pouvant contenir qu'un exemplaire d'un objet

- `add(object)` ajoute un objet s'il n'est pas deja présent
- `contains(object)`
- `remove(object)`
- `ceiling(Object)` retourne le plus petit élément de l'ensemble plus grand ou egal à l'objet
- `floor(objet)`: retourne le plus grand élément e l'ensembl plus petit ou égal à l'objet

---

Interface Map

Une Map lie une clé à exactement une valeur. Une clé ne peut pas être dupliqué dans une map. toute map possède, entre autre ces methodes :

- `void clear()` : retire tout les éléments de la map
- `boolean containsKey(Object key)` retourne si une clé existe ou non dans la collection
- `boolean containsValue(Object v)` retourne si une valeur existe ou nan dans la collection
- `boolean get(Object Key)` retourne la valeur associée à la clé, ou null
- `boolean isEmpty()` retourne vrai si la map est vide
- `V put(K key, V value)` lie une valeur à une clé, retourne la valeur précédemmment associée
- `V remove(Object key)` retire une clé et son association. retourne la valeur associée
- `V replace(K key, V value)` lie une nouvelle valeur çà un clé existante, retourne la valeur précédement associée
- `int size()` retourne la taille de la collection
    ```

- `Collection<V> values()` retourne les valeurs de la map
- `set<K> keyset()` retourne les clés de la map dans un sensemble
- `void forEach(Biconsumer<K,V> action) applique une action à chaque couple clé-valeur`

HashMap Hashtable

La classe `HashMap` (non synchronisée) est un implementation de l'interface `Map`. La classe `Hashtable` est quasi identique mais synchronisée.

Gestion des d'erreurs & entrées-sortie

Gestion des exceptions

La gestion des exceptions est très importante, voire primordiale, dans tous systèmes informatiques. Elle permet d'eviter les applications qui plantent sans information.

Les exception sont des instances des classe heritant des classes

- java.lang.Error
  - Pour les erreurs graves quasi impossible a gerer : plus de mémoire, classe manquante, ...
- java.lang.Exception
  - Pour les exception attendues sinon probables pouvant etre gérée : débordement d'un tableau, erreur de calcul, ...
- Ces deux classe implémentent l'interface `Throwable`

La classe `Exception`possède deux constructeurs

- `Exception()`
- `Exception(String msg)`

Elle hérite de l'interface `Throwable`

- `getMessage` qui permet de récupérer le message de l'exception s'il existe
- `toString`, qui retourne la classe et le message sous forme de chaine
- `printStackTrace`, qui fait appel à `toString` mais qui en plus indique l'endroit du programme où a été levée l'exception.

Gerer les exceptions :

```
try{
  // endroit où pourrait apparaître une exception
}
catch(typeException e){
  // traitement de l'exeption
}
finally{
  // dans tout les cas passer par là
}
```

Classes d'exception

ClassNotFoundException, CloneNotSupportedException, IllegalAccessException, .InstantiationException, InterruptedException, NoSuchFieldException, NoSuchMethodException, RuntimeException, ArithmeticException, ArrayStoreException, ClassCastException, IllegalArgumentException, IllegalThreadStateException, NumberFormatException, IllegalMonitorStateException, IllegalStateException, IndexOutOfBoundsException, ArrayIndexOutOfBoundsException, StringIndexOutOfBoundsException, NegativeArraySizeException, NullPointerException, SecurityException, UnsupportedOperationException

Propagation d'une exception

- Si une exception n'est pas gérée dans une procedure, elle se propage hors de la procédure
- Il faut donc que cette procédure soit capable de propager l'exception (utilisation du mot clé `throws`)
- Si à la fin du programme, une exception n'a pas été attrapée, la pile des méthodes traversées par l'exception est affichée

Définir son exception

Il est possible de définir une exception en la faisant dériver de la classe Exception

```
class MonException extends Exception{

}
```

Les entrées-sortie

La classe Java.lang.System

- Utilisation des classes et interfaces du package java.io
- Il existe trois flux
  - `System.in` : flux d'entrée standart
  - `System.out` : flux de sortie standart
  - `System.err` : flux de sortie pour message d'erreur
- Les entrées sorties peuvent générer une exception de type `IOException`

Fonctionnement de l'E/S en JAVA

- Java procède par étape pour l'E/S dans un fichier, une passerelle réseau, ...
  - Il faut tout d'abord définir un flux d'entrée ou de sortie vers un objet source ou de destination
  - Puis, il faut utiliser un objet d'entrée ou de sortie vers un objet source ou de destination
  - Puis, il faut utiliser un objet d'entrée ou de sortie dédié au type de données à lire/envoyer.

Gestion de fichiers

La classe `File` permet d'accéder aux caracteristiques des fichiers et répertoires

Voici ces constructeurs :

- `File(File dir, String name)`
- `File(String path, String name)`
- `File(String path)`

Voici ces methodes :

canRead() ; canWrite() ; delete() ; equals(Object obj) ; exists() ; getAbsolutePath() ; getCanonicalPath() ; getName() ; getParent() ; getPath() ; hashCode() ; isAbsolute() ; isDirectory() ; isFile() ; lastModified() ; length() ; list(FilenameFilter filter) ; list() ; mkdir() ; mkdirs() ; renameTo(File dest) ; toString()

Flux d'entrées

- `InputStream`
- `reader` (orienté caractères)
- `Scanner`(orienté caractères) permet de lire une entrée texte et egalement d'utiliser un parseur

Flux de sortie

- `OutputStream`
- `Writer`

La sérialisation

- Signifie persistante
- Pour eviter la perte des données à la fin de l'execution
- Si un obket contien un autre objet, ce dernier est aussi sérialisé et on obtient un abre de sérialisation.
- Les objets `transients` (transitoire, éphémères)
- Ce sont des objets qui ne peuvent pas être sérialisés (processus, flux de données)
- Si ils sont féfinis en tant qu'attributs dans un objet sérialisé, il faut indiquer qu'ils sont transitoires
- idem si on ne veux pas que certains attributs d'un objet soient sérialisés
- Syntaxe :
  - La classe doit implémenter l'interface serializable (pas de méthode à implémenter)
  - Exemple

```
import java.io.*

class MaClass implements Serializable {
  public transient Thread mathread;
  private String nom;
  private int total;
}


// sauvegarde d'un objet
public static void main(String args[]){
  Personne p1 = new Personne("Mérovée");
  Personne p2 = new Personne("Childeric");
  File f = new File("/home/fredo/fichier","Essai.obj");
  try{
    FileOutputStream fos = new FileOutputStream(f);
    ObjectOutputStream oos = new ObjectOutputStream(fos);
    oos.writeObject(p1);
    oos.writeObject(p2);
  }
  catch(Exception e){
    System.out.println("erreur dans l'écriture'" + e);
  }
}


//Lecture d'un objet
public static void main(String args[]){
  File f = new File("/home/fredo/fichier","Essai.obj");
```

```
    try{
      Personne p1 = (Personne) ois.readObject();
      Personne p2 = (Personne) ois.readObject();
    }
    catch(Exception e){
      System.out.println("erreur dans la lectutre" + e);
    }
  }
```

Gestion de processus

Un processus c'est un code qui s'execute "en parallèle" c'est à dire dont l'execution n'est pas bloquante pour le reste du programme

Exemple d'utilisation :

- longs calculs lancés en tâche de fond, permettant de lancer d'autres calculs.
- L'interaction avec l'utilisateurs (les fenetres sont des processus)
- Le grabage collector est un processus

Le programme principale est le processus principal

Creation de processus

- Etendre la classe Thread permet de créer un processus
- Le comportement principal du processus est à définir dans la méthode `public void run()`
- La méthode `run()` est constituée généralement d'une boucle longue
    - longs calculs
    - attente de données
- Elle est appelée par la méthode `start()`

```
class UneTache extends Thread{
  int nbRuns = 0;
  UneTache(String nom){
    super(nom);
  }
  public void run(){
    while(true){
      ...
      Thread.yield();
    }
  }
}


public class MaTache {
  public static void main(String args[]){
    UneTache tache1 = new UneTache("tache 1");
    UneTache tache2 = new UneTache("Tache 2");
    tache1.start();
    tache2.start();
    int i=0;
    while(i<100){
      ...
      Thread.yield();
    }
    System.exit(0);
  }
}
```

Interface `Runnable`

Permet de lancer le processus dans un thread a part

```
class ObjetComptant implements Runnable {
  String name;
  ObjetComptant(String nom){
    name = nom;
  }
  public void run(){
    while(true){
      ...
      Thread.yield();
    }
  }
}



public class MonRunable{
  public static void main(Strings args[]){
    new Thread(new ObjetComptant("c1")).start();
    new Thread(new ObjetComptant("c2")).start();
    int i=0;
    while(i<100){
      ...
      i++;
      thread.yield();
    }
    System.exit(0);
  }
}
```

Groupement de processus

- Possibilité d'associer des processus à des groupes (`ThreadGroup`)
  - Possibilité d'associer des groupes a des groupes

Quelques methodes :

- `activeCount()`: nombre de processus actifs dans le groupe
- `enumerate(Thread[] tab)`:place dans tab la liste des processus actifs
- `interrupt()`: interrompt tous les processus du groupe
- `join()`: appel bloquant, attends que les processus meure
- `join(long delai)` attends au plus `delais` ms que le processus meure

```
ThreadGroup groupe = new ThreadGroup("mon groupe");
Thread p1 = new Thread(groupe, ...);
Thread p2 = new Thread(groupe, ...);
p1.start();
p2.start();
while(groupe.activeCount() != 0){
  thread.yield();
}
```

Relation processus-groupe

- Création de processus
  - `Thread(Runnable cible)`: Crée un processus sur l'objet cible
  - `Thread(Runnable clible, String nom)`: crée un processus sur l'objet cible et donne un nom
  - `Thread(ThreadGroup groupe, Runnable cible, String nom)`: crée un processus sur l'objet cible, lui donne un nom et l'affecte à un groupe
  - `Thread(ThreadGroup groupe, RUnnable cible)`: Crée un processus sur l'objet cible et l'affecte à un groupe, le nom est donné automatiquement
- retrouver le groupe, le processus
  - `getThreadGroup()`: dans la classe `Thread` retourne le groupe auquel appartient le processus
  - `Thread.currentThread()`: retourne le processus exécutant la methode

Planifier l'éxécution de processus

Une tache planifiée est de type `TimerTask`, elle possède les méthodes :

- `cancel()`: annule la tache
- `run()`: action exécutée par la tache
- `scheduledExecutionTime()`: retourne la prochaine date en ms à laquelle run() va etre executée

Un `Timer` (`java.util.Timer`) est un objet qui peut temporiser l'execution de la tâche `java.util.TimerTask`.

La classe `Timer` propose ces methodes :

- `schedule(TimerTask task, long delay)`: planifie l'éxecution de la tâche après un délais
- `schedule(TimerTask task, long delay, long period)`: planifie l'exécution de la tâche après un delai, et cycliquement selon une période
- `schedule(TimerTask task, Date moment)` : planigie l'execution de la tache à une date donnée

Synchronisation

- Pour bloquer l'accès mutuel à une fonction, celle-ci doit être précédée du mot clé `synchronized`
- Si un processus appelle une fonction synchronisée dans lequal se trouve deja un processus, il est bloqué aux portes de la fonction.
    - Il entrera dans la fonction si :
        - le processus l'utilisant en sort et qu'aucun autre processus n'etais mis en attente avant lui
        - le processus est mis en attente (wait()) et qu'aucun autre processus n'etais mis en attente avant lui

---

Mise en veille et reveil de processus

- placée dans une fonction d'un objet, l'appel à la fonction `wait()` met en veille le processus l'appelant
- Placée dans une fonction d'un objet, `notify()` réveille un processus mis en veille dans une des procédures de l'objet
- Placée dans une fonction d'un objet, `notifyAll()` réveille tous les processus mis en veille dans une des procédures de l'objet

---

Volatilité

- La JVM de java est composée d'une mémoire centrale et d'une mémoire cache
- Un changement de valeur d'une variable s'effectue en mémoire cache
- Si un processus souhaite accéder à la variable, il reçoit la valeur de la mémoire centrale
- Certains attributs doivent donc être rafraîchis en permanence

on utilise alors le mot clé `volatile`

- Les attributs volatiles :
    - sont chargés de la mémoire centrale avant chaque utilisation
    - sont stocké en mémoire centrale après chaque accès/ecriture
- A utiliser si une variable est partagée, hors d'un bloc synchronisé

Créer un bloc synchronisé

```
synchronized (compteur) {compteur.add(); compteur.mult();}
```

- Synchronise l'accès à l'objet compteur
- lorsqu'une méthode synchronisée d'un objet est appelée, le verrou est mis, aucune autre méthode synchronisée de cet objet peut être exécutée
    - tant que le processus n'est pas sorti
        - sauf s'il a été mis en attente
- Acquerir le verrou d'un objet est forcément coûteux, Il faut donc savoir gérer et diminuer au maximum les section critiques.

Communication entre processus

- SI un processus veut envoyer une donnée à un autre processus il peut appeler une fonction, ou communiquer par envoi de message
- Pour cela, il faut créer un tube de communication
    - L'emmeteur reçoit l'entrée du tube
    - le destinateure deçoit la sortie

Travaux dirigés

TD1

Créer une classe représentant un Bidule

Un objet Bidule est composé :

- d'un nom, chaine de caractères
- d'un volume, valeur entière
- Créer un constructeur par paramètres initialisant le nom et le volume.
- Surcharger la méthode toString() pour retourner une représentation d'un bidule par son nom, son volume.

```
public  class Bidule {
  String nom;
  int volume;
  Bidule() { nom="";   }
  Bidule(String _nom, int _volume) {
    nom = _nom;
    volume = _volume;
```

```
  }

  @Override
  public String toString(){
    return("bidule  "+ no + ", nom="+nom.toUpperCase() + ", volume = " + volume );
  }

}
```

Dans une classe TD1 créer la methode static void testCreation bidule(int nb) qui crée nb objet Bidule

```
static void testCreationBidules(long nb) {
    for (long i = 0; i < nb; i++) {
        Bidule b = new Bidule();
    }
}
```

Créer une classe Chose qui étend Bidule.

- Un objet Chose possède un attribut de poids de type réel
- Créer le contructeur de Chose par défaut qui initialise le volume à 10 et le poid à 3.4

Redéfinir la methode toString() pour Chose.

```
/**classe Chose, qui herite de Bidule*/
public class Chose   extends Bidule {
  double poids;

  /** constructeur par défaut*/
  Chose() {
    volume = 10;
    poids=3.4;
  }

  Chose(String _nom, int _volume, double _poids) {
    super(_nom, _volume);
    poids = _poids;
  }

  public String toString()  {
    return("chose "+ no+ ", nom=" + nom.toUpperCase() + ", volume =  " + volume + ", poids = " + poids);
  }
}
```

Dans la classe TD1, créer une méthode testHéritage qui crée 2 objets Bidules et 2 objets Choses.

- Placer ces objets dans un tableau de Bidule.
- Balayer cette liste et afficher les objets.

```
static void testHeritage() {
   Bidule[] tab = {new Bidule("b", 1),
      new Chose("c", 2, 1.2),
      new Bidule("bb", 10),
      new Chose()};
   for(Bidule b:tab) System.out.println(b);
 }
```

Redéfinir la méthode equals pour les classes Bidule et Chose.

Définir la fonction TestEgalite dans la classe TD1 pour tester l'egalité entre objets bidule

```
//dans la classe  Bidule :
public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return false;

    Bidule bidule = (Bidule) o;

    if (volume != bidule.volume) return false;
    return Objects.equals(nom, bidule.nom);
}

//dans la classe  Chose :
public boolean equals(Object o) {
    if (this == o) return true;
```

```
    if (o == null || getClass() != o.getClass()) return false;
    if (!super.equals(o)) return false;

    Chose chose = (Chose) o;

    return Double.compare(chose.poids, poids) == 0;
}
```

Créer la méthode testTriPrimitive dans la classe TD1. Cette méthode définit et trie un tableau d'entiers. Le nombre d'entiers étant passé en paramètre. Calculer la durée du tri.

```
static void testTriPrimitive(int taille) {
// creer un tableau de taille entiers
  int[] tab = new int[taille];
// initialiser le tableau avec des valeurs aléatoires entre 0 et taille
  Random hasard = new Random()
  Arrays.setAll(tab, i -> hasard.nextInt(taille));

// afficher le contenu du tableau s'il n'est pas trop grand
  if(taille<101) System.out.println(" tab=" + Arrays.toString(tab));

  long tpsDebut = System.currentTimeMillis();
  java.util.Arrays.sort(tab);
  long tpsFin = System.currentTimeMillis();

  System.out.println("tri en : " + (tpsFin - tpsDebut));

  if(taille<101) System.out.println(" tab=" + Arrays.toString(tab));
}
```

La classe Bidule implémente maintenant l'interface Comparable. Ajouter la méthode nécessaire à la classe Bidule

```
public  class Bidule implements Comparable<Bidule>

//on ajoute la fonction de comparaison :
public int compareTo(Bidule autre) {
  int retour = 0;
  retour = nom.compareTo(autre.nom);
  if(retour==0) retour = volume-autre.volume;
  return retour;
}
```

Dans TD1, créer la maéthode triCollection qui définit une collection de Bidule par nom puis par volume

```
static void triCollection() {
int nb = 10;
Random hasard = new Random()
ArrayList<Bidule> listeB = new ArrayList<>();
  for (int i = 0; i < nb; i++)
    listeB.add(new Bidule("b" + hasard.nextInt(nb), i));

  listeB.forEach(b-> System.out.println(b));

  Collections.sort(listeB);

  listeB.forEach(b-> System.out.println(b));
}
```

Toujours dans la méthode triCollection, effectuer un tri uniquement par valeur décroissante à l'aide de la méthode sort(Comparator) de la classe ArrayList et d'un comparateur passé en notation lambda.

```
...
  listeB.sort((b1,b2)-> (b2.volume-b1.volume));

  listeB.forEach(b-> System.out.println(b));
}
```

TD2

Enumeration

Créer une énumération Moyen qui liste les moyens de transport Train, Tram et Bus, en initialisant leurs coût et vitesse.

```
public enum Moyen {BUS(0.1, 40), TRAM(0.3, 50), TRAIN(1, 70);
  /**cout en euro au km*/
  final double cout;
  /**vitesse en km/h*/
  final double v;
  Moyen(double _cout, double _v){cout = _cout; v = _v;}
}
```

Créer une énumération Ville comme la suivante :

enum Ville{A,B,C,D,E,F;}

Mais en l'étendant pour qu'elle contienne

- le tableau des distances,
- la fonction static double getDist(Ville start, Ville end) qui retourne la distance entre les deux villes start et end.
- la fonction double getDist(Ville end) qui retourne la distance entre la ville courante et la ville end.

Remarque :

- Si x est un objet de type énuméré, x.ordinal() retourne le no de déclaration de x.* Si NomType est un type énuméré, NomType.values() retourne un tableau de tous ses éléments.

```
public enum Ville {
  A,B,C,D,E,F;
  private final static int [][] tabDist = {
    {0,  10, -1, 20, -1, 35},
    {10,  0, 10, 15, -1, -1},
    {-1, 10,  0, 10, 15, 20},
    {20, 15, 10,  0, 15, 20},
    {-1, -1, 15, 15,  0, 10},
    {35, -1, 20, 20, 10,  0}};

  static double getDist(Ville start, Ville end)
    { return tabDist[start.ordinal()][end.ordinal()]; }

  double getDist( Ville end)
    { return tabDist[this.ordinal()][end.ordinal()]; }
}
```

Créer une classe Tests qui comprend :

- une méthode `static void testEnumMoyens()` qui affiche la liste des moyens, leurs vitesses et coût au km associés.
- static void testVilles() qui affiche les distances entre chaque ville.

```
public class Tests {

static void testEnumMoyens(){
  for(Moyen m: Moyen.values())
    System.out.printf("%s :: cout = %.2f €/km, vitesse = %.2f km/h %n", m, m.cout, m.v);
  System.out.println("---");
  System.out.println(Arrays.toString(Moyen.values()));
}

static void testVilles() {
 for (Ville v1 : Ville.values()) {
  for (Ville v2 : Ville.values()) {
   double d = Ville.getDist(v1, v2);
   if (d > 0)
    System.out.printf("distance entre %s et %s = %.2f km%n", v1, v2, d);
   }
   System.out.println("------");
 }
}

  public static void  main(String[] args)
  {
    testEnumMoyens();
    testVilles();
  }
}
```

Trajet simple

Créer une classe trajet simple qui représente un trajet direct existant entre deux villes

- Depart, Arrivee (Ville)
- distance : double
- moyen : Moyen
- coût : double
- dateDepart : LocalTime
- durée : int (minutes)
- dateArrivee : LocalTime

Définir le constructeur d'un trajet qui prend en entrée le départ, l'arrivée, la date de départ, le moyen ; mais sous forme d'énumération et de date.

Les valeurs des variables coût, durée et date d'arrivée sont calculées automatiquement par la méthode privée `private void calcule()`.

Définir le constructeur d'un trajet qui prend en entrée le départ, l'arrivée, le moyen sous forme de chaîne et la date de départ sous forme d'entier sous la forme hhmm. Exemple int d=1045 signifie un depart à 10h45.

Définir la méthode `public void setMoyen(Moyen _moyen)` qui permet de changer la moyen de locomotion d'un voyage. Cette fonction appelle le recalcul de la date d'arrivée, de la durée et du coût.

```
import java.time.LocalTime;

public class TrajetSimple implements  Cloneable {
  /** origine */
  Ville depart;
  /** destination */
  Ville arrivee;
  /** moyen de transport*/
  Moyen moyen;



  /** duress du voyage en minutes */
  int duree;
  /** longueur du parcours en km */
  double distance;
  /** date de depart */
  LocalTime dateDepart;
  /** date d'arrivee */
  LocalTime dateArrivee;
  /** cout */
  double cout;

  TrajetSimple() { }

  /**constructeur où les arrivée, depart, moyen sont en chaine de caractères
  * et où la date de départ est une chaîne de type hhmm*/
  TrajetSimple(String _depart, String _arrivee,  int _dateDepart, String _moyen) {
    depart = Ville.valueOf(_depart.toUpperCase());
    arrivee = Ville.valueOf(_arrivee.toUpperCase());
    moyen = Moyen.valueOf(_moyen.toUpperCase());
    distance  = Ville.getDist(depart,  arrivee);
    int hh= _dateDepart / 100;
    int mm = _dateDepart - hh*100;
    dateDepart = LocalTime.of(hh, mm);
    calcule();
  }

  /**constructeur de TrajetSimple*/
  TrajetSimple(Ville _depart, Ville _arrivee,  LocalTime _dateDepart, Moyen _moyen) {
    depart = _depart;
    arrivee = _arrivee;
    moyen = _moyen;
    dateDepart = _dateDepart;
    distance  = Ville.getDist(depart,  arrivee);
    calcule();
  }

  /**calcule la durée, la date d'arrivée, et le coût en fonction des villes
  * et du moyen de transport (si distance==-1 (car aucun trajet direct possible
  * entre les villes), cout=duree=-1 et date d'arrivee = null))*/
  private void calcule() {
    duree = (distance==-1?-1:(int) (60d*distance / moyen.v));
    dateArrivee = (distance==-1?null:dateDepart.plusMinutes(duree));
    cout = (distance==-1?-1:distance*moyen.cout);
  }


  @Override
  public String toString() {
```

```
    StringBuilder sb = new StringBuilder("trajet de ");
    sb.append(depart).append(" à ").append(arrivee);
    sb.append(" par ").append(moyen);
    sb.append(", depart: ").append(dateDepart);
    sb.append(", arrivee: ").append(dateArrivee);
    sb.append(", cout = ").append(cout);
    sb.append(", distance = ").append(distance);
    return sb.toString();
  }

  @Override
  protected TrajetSimple clone()  {
    TrajetSimple clone=null;
    try { clone = (TrajetSimple) super.clone(); }
    catch (CloneNotSupportedException e) { e.printStackTrace();}
    return clone;
  }

  public void setMoyen(Moyen _moyen) { moyen = _moyen; calcule(); }
  public void setCout(double cout) { this.cout = cout;}

  public Ville getDepart() { return depart; }
  public Ville getArrivee() { return arrivee; }
  public Moyen getMoyen() { return moyen; }
  public int getDuree() { return duree; }
  public double getDistance() { return distance; }
  public LocalTime getDateDepart() { return dateDepart; }
  public LocalTime getDateArrivee() { return dateArrivee; }
  public double getCout() { return cout; }
}
```

Définissez dans la classe Test une méthode static void testTrajetSimple1() :

- créer une liste de TrajetSimple
- ajouter des trajets simples
- utilise la classe Collections et ses méthodes min et max pour déterminer le trajet le plus court, le trajets le plus cher et le trajets le plus rapide.

```
static void testTrajetsSimples() {
  TrajetSimple tj1 = new TrajetSimple("A", "F", 824, "train");
  TrajetSimple tj2 = new TrajetSimple("B", "D", 831, "bus");
  TrajetSimple tj3 = new TrajetSimple("F", "D", 844, "tram");

  var liste = List.of(tj1, tj2, tj3);

  liste.forEach(System.out::println);

  System.out.println("-".repeat(40));

  System.out.print("-trajet le moins cher " );
  var pasCher = Collections.min(liste, Comparator.comparingDouble(TrajetSimple::getCout));
  System.out.println(pasCher);

  System.out.print("-trajet le plus long " );
  var plusLong = Collections.max(liste, Comparator.comparingInt(TrajetSimple::getDuree));
  System.out.println(plusLong);

  var coutTotal = liste.stream().mapToDouble(TrajetSimple::getCout).sum();
  System.out.printf("-cout total des voyages : %.2f %n",coutTotal);
}
```

Trajet compose

Créer une classe TrajetCompose.

Cette classe possède une liste de trajets consécutifs ; c'est à dire que l'arrivée d'un trajet est le départ du trajet suivant dans la liste.

Une trajet composé possède un départ, une arrivée, une durée totale, un coût total, une date de départ et une date d'arrivée.

- Créer la fonction private void calcule() qui calcule la durée totale et le coût total à partir des trajets contenus dans le trajet composé. La fonction initialise également les dates de départ et d'arrivée du trajet composé.
    - La méthode ChronoUnit.MINUTES.between(dateDepart, date) qui retourne le nombre de minutes séparant les dates (nombre pouvant être négatif) sera être utilisé.
- Créer la fonction void add(TrajetSimple trajet) qui ajoute le trajet à la liste des trajets s'il est bien une suite du voyage déjà constitué.
- Créer la fonction void add(List trajets) qui ajoute les trajets passés en paramètres à la liste des trajets. On suppose que les trajets se suivent logiquement. Cette fonction appelle la fonction calcule()

- Surcharger la fonction String toString() pour afficher le départ et l'arrivée finale d'un trajet composé, ainsi que le coût total, les date de départ et d'arrivée, et la durée totale.

Dans la classe Test, rédigez la procédure static void testTrajetCompose() qui

- crée une trajet composé pour y ajouter les trajets simples de A à B par Bus, de B à C par Bus, et de C à F en Train.
- crée une trajet composé pour y ajouter les trajets simples de A à B par Bus, et de C à F en Train. Cet ajout ne doit pas être permis.

```
import java.time.LocalTime;
import java.time.temporal.ChronoUnit;
import java.util.ArrayList;
import java.util.List;

public class TrajetCompose  {
  /** origine */
  Ville depart;
  /** destination */
  Ville arrivee;
  /** duress du voyage en minutes */
  int duree;
  /** longueur du parcours en km */
  double distance;
  /** date de depart */
  LocalTime dateDepart;
  /** date d'arrivee */
  LocalTime dateArrivee;
  /** cout */
  double cout;

  /**liste de trajets simples*/
  List<TrajetSimple> listeTrajets;

  TrajetCompose() {
  listeTrajets = new ArrayList<>();
  }


  /**calcule la durée, la distance et le cout total*/
  private void calcule() {
  cout = listeTrajets.stream().mapToDouble(TrajetSimple::getCout).sum();
  distance = listeTrajets.stream().mapToDouble(TrajetSimple::getDistance).sum();
  duree = (int) ChronoUnit.MINUTES.between(dateDepart, dateArrivee);
  }

  /**ajoute le trajet à la liste des trajets s'il est bien une suite du voyage déjà constitué.
  * @param tj trajet simple à ajouter
  * @return true si le trajet a pu être ajouté
  */
  public boolean addTrajetSimple(TrajetSimple tj){
  boolean ok = false;
  if (listeTrajets.size()==0)
  {
    listeTrajets.add(tj);
    depart = tj.getDepart();
    dateDepart = tj.getDateDepart();
    ok = true;
  }
  else {
    if (tj.getDepart() == arrivee) {
    if (tj.getDateDepart().compareTo(dateArrivee) > 0) {
      listeTrajets.add(tj);
      ok = true;
    }
    }
  }
  if (ok){
    arrivee = tj.getArrivee();
    dateArrivee = tj.getDateArrivee();
    calcule();
  }
  return ok;
  }

 /**ajoute une liste de trajets simple autant que possible
   * @param trajets liste de trajets simples à concaténer
   * @return true si les trajets ont pu être tous ajoutés
   */
  boolean addTrajets(List<TrajetSimple> trajets){
```

```
    boolean ok = true;
    int nb = trajets.size();
    for(int i=0; i<nb && ok; i++)
    {
      var tj = trajets.get(i);
      ok = addTrajetSimple(tj);
    }
    return ok;
    }

    @Override
    public String toString() {
    StringBuilder sb = new StringBuilder("trajet de ");
    sb.append(depart).append(" à ").append(arrivee);
    sb.append(", depart: ").append(dateDepart);
    sb.append(", arrivee: ").append(dateArrivee);
    sb.append(", cout = ").append(cout);
    sb.append(", distance = ").append(distance);
    sb.append("\n");
    listeTrajets.forEach(tj->sb.append("\t- ").append(tj).append("\n"));
    return sb.toString();
    }

    public Ville getDepart() { return depart; }
    public void setDepart(Ville depart) { this.depart = depart; }

    public Ville getArrivee() { return arrivee; }
    public void setArrivee(Ville arrivee) { this.arrivee = arrivee; }

    public int getDuree() { return duree; }
    public double getDistance() { return distance; }
    public double getCout() { return cout; }

    public LocalTime getDateDepart() { return dateDepart; }
    public void setDateDepart(LocalTime dateDepart) { this.dateDepart = dateDepart; }

    public LocalTime getDateArrivee() { return dateArrivee; }
  }
```

Catalogue

Créer une classe Catalogue.

Cette classe possède une table de hashage tableDepart de type HashMap qui possède en clé la ville de départ et en valeur l'ensemble des trajets partant de cette ville.

- Créer la fonction void addTrajetSimple(TrajetSimple trajet) qui ajoute le trajet à sa bonne clé dans la table.
- Créer la fonction creerCatalogue() qui crée la liste des trajets simples du réseau et les ajoute à la table.
    - Les trajets ont lieu toutes les 30mn, de 6h00 à 20h00
- Créer la fonction trouveCheminsDirects(Ville depart,Ville arrivee, LocalTime dateDepart, int delaiMax)
    - qui retourne la liste des trajets simples entre départ et arrivée partant entre dateDepart et dateDepart + delaiMax.
        - il faut utiliser les récentes méthodes sur les listes (liste.removeIf(.....))
    - Exemple de méthodes utiles : si date et dateDepart sont de type LocalTime
- Créer la fonction public boolean trouverCheminIndirect(Ville depart, Ville arrivee, LocalTime date, int delai, `ArrayList<TrajetSimple> compositionEnTest`, `List<Ville> via`, `List<TrajetCompose> resultats`) qui calcule les chemins directs et indirects possibles entre 2 villes à partir d'une date donnée avec un retard et un delai entre voyages autorisé.

compositionEnTest est une suite de trajets simples qui tentent d'aller de départ à arrivée. Si la composition réussit (arrivée de la composition = arrivée), la composition est ajoutée à la liste des résultats.

Via est une liste des ville visitée par la composition en test (pour éviter de repasser 2 Foix par la même ville)

Exemple d'utilisation : Recherche des voyages entre A et E partant entre 8h30 et 8h45

```
  import java.io.File;
  import java.io.IOException;
  import java.nio.file.Files;
  import java.time.LocalTime;
  import java.util.ArrayList;
  import java.util.HashMap;
  import java.util.List;
  import java.util.Map;
  import java.util.stream.Collectors;

  public class Catalogue {
    /**map (ville de depart, liste de trajets directs à partir de cette ville)*/
```

```
Map<Ville, List<TrajetSimple>> tableDepart;

Catalogue(){tableDepart = new HashMap<>();}

/**
* cree le catalogue correspondant à l'énoncé
* */
public void creerCatalogue()
{
List<TrajetSimple> trajets = new ArrayList<>();
for(Ville x:Ville.values())
  for(Ville y:Ville.values())
  {
  if (x!=y && Ville.getDist(x, y)>0)
  {
    TrajetSimple ts = new TrajetSimple(x, y, LocalTime.of(6, 0), Moyen.BUS);
    if( (x==Ville.A && y==Ville.F) || (x==Ville.F && y==Ville.A) ) ts.setMoyen(Moyen.TRAIN);
    if( (x==Ville.A && y==Ville.D) || (x==Ville.D && y==Ville.A) ) ts.setMoyen(Moyen.TRAM);
    if( (x==Ville.D && y==Ville.F) || (x==Ville.F && y==Ville.D) ) ts.setMoyen(Moyen.TRAM);
    trajets.add(ts);
    //repete le trajet jusqu'au soir
    for(int i=1; i<28;i++)
    {
    TrajetSimple tsSuite = ts.clone();
    tsSuite.dateDepart = ts.dateDepart.plusMinutes(i*30);
    tsSuite.dateArrivee = ts.dateArrivee.plusMinutes(i*30);
    trajets.add(tsSuite);
    }
  }
  }

//à partir de la liste des trajets, crée automatiquement la map grâce à un collector
tableDepart = trajets.stream().collect(Collectors.groupingBy(TrajetSimple::getDepart));
}

/**
* trouve tous les trajets directs existants entre les villes de depart et d'arrivée à partir d'une certaine
* et avant un certain délai d'attente
* @param depart
*   ville de depart
* @param arrivee
*   ville d'arrivee
* @param dateDepart
*         date de depart
* @param delaiMax
*         minutes de retard autorisées
* @return les trajets directs correspondant à la demande
*/
List<TrajetSimple> trouveCheminsDirects(Ville depart, Ville arrivee, LocalTime dateDepart, int delaiMax) {
List<TrajetSimple> cheminsDirects = null;
List<TrajetSimple> trajets = tableDepart.get(depart);
if (trajets != null)
{
  cheminsDirects = new ArrayList<>(List.copyOf(trajets));
  LocalTime dateDepartMax = dateDepart.plusMinutes(delaiMax);
  cheminsDirects.removeIf(t->(t.arrivee != arrivee  || t.dateDepart.isBefore(dateDepart) || t.dateDepart.isA
  if (cheminsDirects.isEmpty()) cheminsDirects = null;
}
return cheminsDirects ;
}

/**
* calcule les chemins directs et indirects possibles entre 2 villes
* à partir d'une date donne avec un retard et un delai entre voyage autorise<br>
* Exemple d'utilisation ci-dessous :
* <ul>
*  <li>Recherche des voyages entre A et E partant entre 8h30 et 8h45
*  <ul>
*   <li>ArrayList<TrajetCompose> voyages = new ArrayList<>();</li>
*   <li>boolean result = catalogs.trouverCheminIndirect(Ville.A, Ville.E, LocalTime.of(8,30), 15, new ArrayL
*  </ul>
*  </li>
* </ul>
*
* @param depart
*   vile de depart
* @param arrivee
*   ville d'arrivee
```

```
* @param momentDepart
*   date de depart souhaitee
* @param delai
*   delai maximal autorise avant de partir, ou entre 2 voyages
* @param voyageEnCours
*   voyage en train d'etre construit
* @param via
*   liste des villes visitees par le voyage
* @param results
*   liste de tous les chemins indirects possibles
* @return true si au moins un chemin a ete trouve
*/
public boolean trouverCheminIndirect(Ville depart, Ville arrivee, LocalTime momentDepart, int delai, List<Tr
          List<Ville> via, List<TrajetCompose> results) {
boolean result;
via.add(depart);
var list = tableDepart.get(depart);
if (list == null) return false;
for (TrajetSimple j : list) {
  if (j.dateDepart.compareTo(momentDepart)>0
    && j.dateDepart.compareTo(momentDepart.plusMinutes(delai))<0)
  if (j.arrivee == arrivee) {
    voyageEnCours.add(j);
    var compo = new TrajetCompose();
    compo.addTrajets(voyageEnCours);
    results.add(compo);
    voyageEnCours.remove(voyageEnCours.size() - 1);
  } else {
    if (!via.contains(j.arrivee)) {
    voyageEnCours.add(j);
    trouverCheminIndirect(j.arrivee, arrivee, j.dateArrivee, delai, voyageEnCours, via, results);
    via.remove(j.arrivee);
    voyageEnCours.remove(j);
    }
  }
}
result = !results.isEmpty();
return result;
}

/**lit un fichier CSV constitué des colonnes : <br>
* Ville de depart, Ville d'arrivée, Moyen de locomotion, Date de 1er départ, nb de répétitions, période entr
* Ex : A, F, Train, 600, 14, 60<br>
* et crée le catalogue des trajets associés*/
public void lireCSV(String fichier)
{
var liste = new ArrayList<TrajetSimple>();
try ( var lines = Files.lines(new File(fichier).toPath()))
{
  var listeLignes = lines.toList();
  for(String ligne:listeLignes) {
  var details = ligne.split(",");
  int date = Integer.parseInt(details[3]);
  TrajetSimple tj = new TrajetSimple(details[0], details[1], date, details[2]);
  liste.add(tj);
  int nb = Integer.parseInt(details[4]);
  long delai = Integer.parseInt(details[5]);
  for (int i = 1; i < nb; i++) {
    var tjClone = tj.clone();
    tjClone.dateDepart = tj.dateDepart.plusMinutes(delai * i);
    tjClone.dateArrivee = tj.dateArrivee.plusMinutes(delai * i);
    liste.add(tjClone);
  }
  }
}
catch (IOException e) {
  System.err.println("Fichier source introuvable !! : "+ fichier );}
if(liste.size()>0) tableDepart  = liste.stream().collect(Collectors.groupingBy(TrajetSimple::getDepart));
}

@Override
public String toString() {
var sb = new StringBuilder();
tableDepart.forEach((v, liste)->{
  sb.append("voyages à partir de ").append(v).append(" : \n");
  liste.forEach(t->sb.append("\t").append(t).append("\n"));
  sb.append("\n");
});
```

```
  return sb.toString();
  }

  public Map<Ville, List<TrajetSimple>> getTableDepart() {
  return tableDepart;
  }

  public void setTableDepart(Map<Ville, List<TrajetSimple>> tableDepart) {
  this.tableDepart = tableDepart;
  }

}
```

Gestion de fichier

```
/**lit un fichier CSV constitué des colonnes : <br>
* Ville de depart, Ville d'arrivée, Moyen de locomotion, Date de 1er départ, nb de répétitions, période entre
* Ex : A, F, Train, 600, 14, 60<br>
* et crée le catalogue des trajets associés*/
public void lireCSV(String fichier)
{
var liste = new ArrayList<TrajetSimple>();
try ( var lines = Files.lines(new File(fichier).toPath()))
{
  var listeLignes = lines.toList();
  for(String ligne:listeLignes) {
  var details = ligne.split(",");
  int date = Integer.parseInt(details[3]);
  TrajetSimple tj = new TrajetSimple(details[0], details[1], date, details[2]);
  liste.add(tj);
  int nb = Integer.parseInt(details[4]);
  long delai = Integer.parseInt(details[5]);
  for (int i = 1; i < nb; i++) {
    var tjClone = tj.clone();
    tjClone.dateDepart = tj.dateDepart.plusMinutes(delai * i);
    tjClone.dateArrivee = tj.dateArrivee.plusMinutes(delai * i);
    liste.add(tjClone);
  }
  }
}
catch (IOException e) {
  System.err.println("Fichier source introuvable !! : "+ fichier );}
if(liste.size()>0) tableDepart  = liste.stream().collect(Collectors.groupingBy(TrajetSimple::getDepart));
}
```