TP R1.01.P2 – Semaines 50 et 01

Présentation du TP

Ce TP se déroule sur 3 X 1h30 dont 1 X 1h30 en groupe entier.

L'application développée dans ce TP simule l'utilisation d'horaires de trains pour effectuer de la planification de trajets. Cette application utilise les types abstraits : *ArrayList*, *String*, *Integer* et *Duree*.

L'application sera codée sur 2 semaines et sera rendue en fin de semaine 01 sous forme d'une archive *Nom_Prenom.zip* qui contiendra les 2 fichiers *Planification1.java* et *Planification2.java*. Archive à déposer le **dimanche 08/01 à 23h55 au + tard**. **Attention**, un **malus** sera appliqué sur la note de TP : -2 points pour tout retard de rendu (même d'une minute !) et -1 point si le nom du fichier (*Nom_Prenom.zip*, PAS de fichier *.rar* SVP) n'est pas respecté.

Dans ce TP, vous aurez besoin de la *javaDoc* des types *ArrayList*, *Integer*, *String* de l'API Java qui se trouve en *https://docs.oracle.com/javase/8/docs/api/*.

Principe général de l'application

Des horaires de train, sous une forme simplifiée, peuvent avoir la forme de tables qui ressemblent à la structure suivante (ici nous avons 2 tables) :

	gareDepart1	**gareDest1**		**gareDepart2**	**gareDest2**
trajet1:	heureDep1	heureArr1	trajet4:	heureDep4	heureArr4
trajet2:	heureDep2	heureArr2	trajet5:	heureDep5	heureArr5
trajet3:	heureDep3	heureArr3	trajet6:	heureDep6	heureArr6
…		…	…		…

L'application doit pouvoir gérer :

- L'affichage à l'écran des horaires de tous les trajets, de même que la durée de chacun d'entre eux. Ceci revient à afficher toutes les tables.
- L'affichage à l'écran des horaires limités aux trajets entre une *gareA* et une *gareB* (sans faire intervenir les correspondances). Ceci revient à afficher une seule table (celle correspondant au trajet *gareA –> gareB*).
- L'affichage de la durée de tous les trajets possibles entre une *gareA* et une *gareB*, sachant que les correspondances entre les trains sont possibles (ce problème ne sera que partiellement traité).

Les types abstraits utilisés

Le but de ce TP étant de manipuler les types abstraits (et les collections), il y en aura 4 différents :

- La classe *String* : type de la bibliothèque *java.lang* et que vous connaissez déjà (méthodes *length()*, *charAt(…)*, *equals(…)*, *split(…)* etc).
- La classe *Integer* : type de la bibliothèque *java.lang* et utilisée essentiellement dans ce TP pour transformer un *String* en type *int* avec la méthode *parseInt(…)*, mais également un *Integer* en *int* avec la méthode *intValue()*.
- La classe *ArrayList* : de type « collection » d'objets (bibliothèque *java.util*), vue en cours.

- La classe *Duree* : classe à récupérer sur Moodle. Cette classe définit une durée temporelle nécessairement positive ou nulle. Ce type doit être utilisé dès que l'application définit un instant « t » (en millisecondes ou en heures / minutes / secondes) ou un écart de temps (donc une durée entre un instant « t2 » et un instant « t1 »). Cet écart peut également s'exprimer en millisecondes ou en heures / minutes / secondes. La *javaDoc* de ce type *Duree* est donnée en annexe. Les **seules opérations** que l'on peut effectuer sur le type *Duree* sont : comparaison de 2 *Duree*, addition et soustraction de *Duree*, transformation d'une *Duree* en millisecondes (type *int*) et transformation de la *Duree* en chaîne de caractères selon différents formats.

Planification version1

Dans cette première version (*Planification1.java*), les tables des horaires de train sont remplies « à la main en dur » dans 2 collections de type *ArrayList* :

- Une collection de **chaîne de caractères** *ArrayList<String> trajets = {id trajet, type train, lieu dep, lieu arr},* où :
  - *id trajet* = l'identifiant du trajet (un entier : "1", "2", …, "15", …), **c'est également la clé qui permettra de retrouver les horaires de ce trajet** dans l'autre collection de type *ArrayList<Integer>*
  - *type train* = "TER", "TGV", …
  - *lieu dep* = la gare de départ
  - *lieu arr* = la gare d'arrivée
- Une collection d'**entiers** *ArrayList<Integer> horaires = {id trajet, heure dep, min dep, heure arr, min arr},* où :
  - *id trajet* = l'identifiant du trajet (le même que dans la structure *ArrayList<String> trajets*),
  - *heure dep* = un entier qui désigne l'heure au moment du départ (entre 0 et 23)
  - *min dep* = un entier qui désigne les minutes au moment du départ (entre 0 et 59)
  - *heure arr* = un entier qui désigne l'heure au moment de l'arrivée (entre 0 et 23)
  - *min arr* = un entier qui désigne les minutes au moment de l'arrivée (entre 0 et 59)

Bien comprendre que TOUS les trajets se trouvent à la suite l'un de l'autre dans la collection *ArrayList<String> trajets* et que TOUS les horaires (correspondant à chacun des trajets) se trouvent à la suite l'un de l'autre dans la collection *ArrayList<Integer> horaires* mais pas forcément dans le même ordre. Donc a**ttention** : l'ordre de rangement des données entre la collection *trajets* et la collection *horaires* n'est pas forcément le même. Il faut donc obligatoirement faire une correspondance par identifiants « id trajet » identiques (clé en BDD) pour retrouver un *horaire* correspondant à un *trajet*.

Exemple pour une table « Vannes –> Redon » :

	Vannes	**Redon**
TER_12:	9h35	10h30
TGV_13:	8h	10h05

La collection des trajets contiendra : { "12", "TER", "Vannes", "Redon", "13", "TGV", "Vannes", "Redon" }

La collection des horaires contiendra : { 13, 8, 0, 10, 5, 12, 9, 35, 10, 30 }

La classe Planification version1 (semaine 50)

La structure de la classe *Planification1* sera la suivante :

```
import java.util.*;
class Planification1 {

      void principal() {
         testAffichages() ;
      }

      void testAffichages() {
         ArrayList<String> trajets = new ArrayList<String>();
         ArrayList<Integer> horaires = new ArrayList<Integer>();
         // Appel des méthodes codées dans la classe
      }

      // Les autres méthodes à coder
}
```

Méthodes

1. Ecrire la méthode *void remplirLesCollections (ArrayList<String> trajets, ArrayList<Integer> horaires)* qui remplit les 2 collections *trajets* et *horaires* avec les données suivantes (mais vous pouvez bien entendu en ajouter autant que vous voulez) :

	Vannes	**Redon**
TER_01:	9h35	10h30
TGV_02:	8h	10h05

	Redon	**Nantes**
TER_03:	11h	12h30
TER_04:	14h15	15h37

	Vannes	**Nantes**
TGV_05:	10h03	12h11
TGV_06:	11h25	13h38

2. Pour tester visuellement la méthode précédente, écrire la méthode :

   void afficherHorairesEtDureeTousTrajets (ArrayList<String> trajets, ArrayList<Integer> horaires)

   Il s'agit donc de parcourir correctement les 2 collections pour récupérer :

   - dans la collection *trajets*, la clé d'identification du trajet, le type de train et les gares de départ et d'arrivée,
   - dans la collection *horaires*, les heures et minutes (*heure dep, min dep, heure arr, min arr*) du départ et de l'arrivée correspondant à la clé d'identification du trajet.

Et pour chaque trajet, la durée (du trajet) doit être affichée. **Bien entendu, chaque fois que vous manipuler un horaire ou une durée, vous devez utiliser le type *Duree***.

L'affichage à l'écran est du texte qui doit ressembler au format suivant :

```
Train <type_train> numéro <id_trajet> :
        Départ de <gareDep> à <HH:MM:SS>
        Arrivée à <gareDest> à <HH:MM:SS>
        Durée du trajet - <HH:MM:SS>
```

3. Ecrire la méthode (qui permettra également de tester le bon remplissage des collections) :

   void afficherHorairesEtDureeTrajets2Gares (ArrayList<String> trajets, ArrayList<Integer> horaires, String gareDep, String gareDest)

   C'est le même principe que la méthode *afficherHorairesEtDureeTousTrajets* mais on se limite cette fois aux seules gares de départ et d'arrivée passées en paramètre (trajets directs).

Planification version2

La classe Planification version2 (semaine 01)

La structure de la classe *Planification2* sera la suivante :

```
import java.util.*;
import java.io.*;
class Planification2 {
      void principal() {
         testAfficherHorairesEtDureeTousTrajets();
         testAfficherHorairesEtDureeTrajets2Gares();
         testChercherCorrespondances();
      }
      void testAfficherHorairesEtDureeTousTrajets() {
         ArrayList<String> trajets = new ArrayList<String>();
         ArrayList<Integer> horaires = new ArrayList<Integer>();
         // Appel des méthodes codées dans la classe
         remplirLesCollections ( trajets, horaires, "./TrajetsEtHoraires.txt" );
         afficherHorairesEtDureeTousTrajets ( trajets, horaires );
      }
      // Les autres méthodes à coder
}
```

Dans cette version2, il s'agit avant tout de remplacer le code de la méthode *remplirLesCollections* par une lecture de fichier texte dont le nom *TrajetsEtHoraires.txt* (à récupérer sur Moodle) est passé en paramètre. La signature de la méthode devient alors :

void remplirLesCollections (ArrayList<String> trajets, ArrayList<Integer> horaires, String nomFich)

Par convention, le format du fichier texte *TrajetsEtHoraires.txt* est le suivant :

id trajet1 / type train / lieu dep / lieu arr

id trajet1 / heure dep / min dep / heure arr / min arr

id trajet2 / type train / lieu dep / lieu arr

id trajet2 / heure dep / min dep / heure arr / min arr

id trajet3 / type train / lieu dep / lieu arr

id trajet3 / heure dep / min dep / heure arr / min arr

…

La lecture du fichier doit se baser sur l'explication vue en cours. Une fois la ligne de texte lue (par exemple *id trajet2 / type train / lieu dep / lieu arr*), la récupération séparée de chaque information *id trajet2, type train, lieu dep, lieu arr* se fera en utilisant la méthode *String[] split(...)* de la classe *String*.

Méthodes

1. Tester la méthode *void remplirLesCollections(...)* en utilisant la méthode (déjà écrite) *void afficherHorairesEtDureeTousTrajets(...)* de la version1.

2. Ecrire et tester (ou récupérer de la version1) la méthode qui affiche les informations de tous les trajets possibles entre une *gareA* et une *gareB*, sachant qu'il s'agit uniquement de trajets directs (pas de correspondances entre les trains).

   void afficherHorairesEtDureeTrajets2Gares (ArrayList<String> trajets, ArrayList<Integer> horaires, String gareDep, String gareDest)

3. Ecrire et tester une nouvelle méthode qui renvoie toutes les correspondances possibles dans une gare à partir d'une heure donnée.

   ArrayList<String> chercherCorrespondances (String gare, Duree heure, ArrayList<String> trajets, ArrayList<Integer> horaires)

   Le retour *ArrayList<String>* est une collection d'identifiants (numéros) de trajets qui doivent TOUS satisfaire les 2 conditions suivantes :

   - la gare de départ du trajet est identique au paramètre *gare*,
   - l'heure de départ de ce trajet doit être postérieure (dans le temps) au paramètre *heure*.

   De manière à pouvoir + facilement visualiser les informations, on écrira d'abord 2 méthodes :

   - *String[] obtenirInfosUnTrajet (String idTrajet, ArrayList<String> trajets)*

     A partir d'un identifiant, la méthode renvoie dans un tableau de *String* 3 informations : le type de train, la gare de départ et la gare de destination.

   - *int[] obtenirInfosUnHoraire (String idTrajet, ArrayList<Integer> horaires)*

     A partir d'un identifiant, la méthode renvoie dans un tableau d'entiers 4 informations : heure départ, minutes départ, heure arrivée, minutes arrivée.

La méthode *chercherCorrespondances* devra faire appel à 1 autre méthode (qui devra être testée indépendamment) :

ArrayList<String> trouverTousLesTrajets (String gareDep, ArrayList<String> trajets)

Par une recherche dans la collection des *trajets*, cette méthode renvoie **tous** les trajets (identifiants des trajets) dont la gare de départ est identique au paramètre *gareDep* (peu importe l'heure de départ car ceci sera vérifié dans un second temps).

Dans un second temps, utiliser la méthode *int[] obtenirInfosUnHoraire (...)* ci-dessus pour ne sélectionner dans la collection des *trajets* **que** ceux dont l'heure de départ est postérieure à *heure* (paramètre de la méthode *chercherCorrespondances* ci-dessus).

ANNEXE : JavaDoc de la classe Duree

Class Duree

java.lang.Object
    Duree

---

public class **Duree**
extends java.lang.Object

Cette classe définit une durée temporelle. Elle permet la manipulation d'intervalles de temps. Une durée s'exprime en millisecondes.

Author:

Kamp J-F.

Constructor Summary

Constructors

Constructor and Description
Duree(int millisec) Constructeur avec initialisation en millisecondes.
Duree(int heures, int minutes, int secondes) Constructeur à partir des données heures, minutes, secondes.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
void	**ajouter**(**Duree** autreDuree) Modificateur qui ajoute une durée à la durée courante.
int	**compareA**(**Duree** autreDuree) Accesseur qui effectue une comparaison entre la durée courante et une autre durée.
java.lang.String	**enTexte**(char mode) Accesseur qui renvoie sous la forme d'une chaîne de caractères la durée courante.
int	**getLeTemps**() Accesseur qui retourne la valeur de la durée courante en millisecondes.
void	**soustraire**(**Duree** autreDuree) Modificateur qui soustrait une durée à la durée courante.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Duree

public Duree(int millisec)

Constructeur avec initialisation en millisecondes.

Parameters:

millisec - la durée exprimée en millisecondes

Duree

public Duree(int heures,
             int minutes,

int secondes)

Constructeur à partir des données heures, minutes, secondes.

Parameters:

heures - nbre d'heures

minutes - nbre de minutes

secondes - nbre de secondes

Method Detail

getLeTemps

```
public int getLeTemps()
```

Accesseur qui retourne la valeur de la durée courante en millisecondes.

Returns:

la durée en millisecondes

compareA

```
public int compareA(Duree autreDuree)
```

Accesseur qui effectue une comparaison entre la durée courante et une autre durée.

Parameters:

autreDuree - durée à comparer à la durée courante

Returns:

un entier qui prend les valeurs suivantes :

- -1 : si la durée courante est + petite que autreDuree
- 0 : si la durée courante est égale à autreDuree
- 1 : si la durée courante est + grande que autreDuree
- -2 : si autreDuree est null (cas d'erreur)

enTexte

```
public java.lang.String enTexte(char mode)
```

Accesseur qui renvoie sous la forme d'une chaîne de caractères la durée courante.

Parameters:

mode - décide de la forme donnée à la chaîne de caractères
La forme de la chaîne de caractères dépend du "mode" (caractère passé en paramètre) choisi :

- si mode == 'J' => chaîne de caractères de la forme "JJJ jours HH h"
- si mode == 'H' => chaîne de caractères de la forme "HHH:MM:SS"
- si mode == 'S' => chaîne de caractères de la forme "SSS.MMM sec"
- si mode == 'M' => chaîne de caractères de la forme "MMMMMM millisec"

Returns:

la durée sous la forme d'une chaîne de caractères

ajouter

```
public void ajouter(Duree autreDuree)
```

Modificateur qui ajoute une durée à la durée courante.

Parameters:

autreDuree - durée à rajouter

soustraire

```
public void soustraire(Duree autreDuree)
```

Modificateur qui soustrait une durée à la durée courante.

Parameters:

autreDuree - durée à soustraire