TP R1.01.P2 – Semaines 50 et 51

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 51 sous forme d'une archive *Nom_Prenom.zip* qui contiendra les 2 fichiers *TestDuree.java* et *Planification.java*. Archive à déposer le **samedi 21/12 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 !). Le nommage du fichier sera impérativement *Nom_Prenom.zip* (PAS de fichier .rar).

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.

Le test de la classe Duree

Le test de la classe *Duree* , nommée *TestDuree*, doit effectuer un test systématique de chaque méthode de cette classe dont la *javaDoc* est fournie en annexe. Le fichier *Duree.class* (déjà écrite et compilée et à récupérer sur Moodle) doit être placé dans le répertoire */class*. Voici le squelette de la classe de test à écrire :

```
class TestDuree {

   void principal() {
      testConstructeur1EtGetLeTemps() ;
      testConstructeur2EtGetLeTemps() ;
      testAjouter() ;
      testSoustraire() ;
      testCompareA() ;
      testEnTexte() ;
   }

   void testConstructeur1EtGetLeTemps () {…}
   …
}
```

Chaque méthode doit être testée dans les 3 cas habituels : cas normaux, cas limites et cas d'erreurs.

Classe pour la Planification de trajets

Les tables des trajets en train sont mémorisées dans un fichier texte *TrajetsEtHoraires.txt* dont le format est le suivant :

id trajet3 / type train / lieu dep / lieu arr

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

id trajet1 / type train / lieu dep / lieu arr

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

id trajet2 / type train / lieu dep / lieu arr

id trajet1 / 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*.

Les informations (textuelles) lues dans le fichier seront mémorisées dans 2 collections *ArrayList* distinctes :

- 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* mais de type entier),
  - *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 se trouvent à la suite l'un de l'autre dans la collection *ArrayList<Integer> horaires* mais pas forcément dans le même ordre que dans *trajets*. Donc **attention** : 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

Si la collection des trajets contient : { "12", "TER", "Vannes", "Redon", "13", "TGV", "Vannes", "Redon" }

La collection des horaires peut contenir les horaires dans cet ordre : { 13, 8, 0, 10, 5, 12, 9, 35, 10, 30 }

La classe Planification

La structure de la classe *Planification* qui met en œuvre la planification de trajets sera la suivante (pour les tests, on se contentera des cas normaux) :

```
import java.util.*;
import java.io.*;
class Planification {

      // Variables globales accessibles par toutes les méthodes
      ArrayList<String> trajets = new ArrayList<String>();
      ArrayList<Integer> horaires = new ArrayList<Integer>();

      void principal() {
         testRemplirLesCollections();
         testAfficherHorairesEtDureeTrajets2Gares();
         testChercherCorrespondances();
         // A COMPLETER
      }

      /*** Les méthodes de test ***/

      void testRemplirLesCollections() {
         // Appel de la méthode à tester "cas normal"
         remplirLesCollections ( "./TrajetsEtHoraires.txt" );
         // Test VISUEL de la bonne lecture du fichier
         afficherHorairesEtDureeTousTrajets();
      }

      void testAfficherHorairesEtDureeTrajets2Gares() {
         // D'abord remplir les collections
         remplirLesCollections ( "./TrajetsEtHoraires.txt" );
         // Appel de la méthode à tester "cas normal"
         // Test VISUEL
         …
         afficherHorairesEtDureeTrajets2Gares ( "Vannes", "Redon" );
         …
      }

      // A COMPLETER
```

```
    /*** Le code des méthodes demandées ***/

    void remplirLesCollections ( String nomFich ) {…}
    void afficherHorairesEtDureeTousTrajets() {…}
    void afficherHorairesEtDureeTrajets2Gares ( String gareDep, String gareDest ) {…}
    ArrayList<String> chercherCorrespondances ( String gare, Duree heure ) {…}
    String[] obtenirInfosUnTrajet ( String idTrajet ) {…}
    int[] obtenirInfosUnHoraire ( String idTrajet ) {…}
    ArrayList<String> trouverTousLesTrajets ( String gareDep ) {…}

}
```

Méthodes

1. Ecrire la méthode *void remplirLesCollections (String nomFich)* qui remplit les 2 collections *trajets* et *horaires* (variables globales !) à partir d'une lecture de fichier texte. Afin de vider systématiquement les collections avant de les remplir, commencer par les 2 lignes de code *this.trajets.clear()* et *this.horaires.clear()*.
2. Pour tester visuellement la méthode précédente, écrire la méthode :

   void afficherHorairesEtDureeTousTrajets ()

   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 (la classe) *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 :

   void afficherHorairesEtDureeTrajets2Gares (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).

4. 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)

   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)*

  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)*

  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)

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:

J-F. Kamp - septembre 2024

Constructor Summary

Constructors

Constructor	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	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.
Cas d'erreurs : si millisec est strictement négatif.

Parameters:

`millisec` - la durée exprimée en millisecondes

Duree

```
public Duree(int heures,
             int minutes,
             int secondes)
```

Constructeur à partir des données heures, minutes, secondes.
Cas d'erreurs : si au moins une donnée parmi heures, minutes, secondes est strictement négative.

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.
Cas d'erreurs : si autreDuree est null.

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.
Cas d'erreurs : si le mode n'est pas soit 'J', soit 'H', soit 'S', soit 'M'.

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 "HH: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.
Cas d'erreurs : si autreDuree est null.

Parameters:

`autreDuree` - durée à rajouter

soustraire

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

Modificateur qui soustrait une durée à la durée courante.
Deux cas d'erreurs :

- si autreDuree est null
- si la durée courante devient strictement négative après soustraction

Parameters:

`autreDuree` - durée à soustraire