1. Préalable

Vous devez implémenter les méthodes d'une classe et répondre à 2 questions posées à même le fichier de la classe.

1.1. Préparation

- Le projet est un projet Java standard, SANS BESOIN D'UTILISER MAVEN.
- Créez un projet Java de base sous votre IDE.
- Téléchargez et décompressez le package fourni sur umtice.
- Rappatriez le package présent sur umtice dans votre projet, à la racine de ce dernier (dossier src/) : ne le mettez pas dans un autre package, pour éviter d'avoir des problèmes de compilation.
- Le package contient a minima :
    - une interface exprimant les méthodes à implémenter, avec leur javadoc ;
    - une classe principale implémentant cette interface, et présentant les éléments suivant déjà implémenté :
        - tous les attributs nécessaires ;
        - les constructeurs nécessaire ;
        - possiblement certaines méthodes privées utilitaires ;
        - une méthode main et des méthodes de test pour tester vos implémentation (à partir de la méthode main, vous n'avez pas besoin de lire ou comprendre tout le code) ;
        - **un cartouche de commentaire contenant 2 questions auxquelles vous devrez répondre.**
    - Selon le sujet, le package peut contenir d'autres classes manipulées par la classe principale.

1.2. Travail à Faire

- Lire le sujet du projet ci-dessous
- Implémenter les méthodes dans la classe qui n'ont le sont pas déjà. **ATTENTION** : la javadoc fournie dans l'interface (recopiée dans la classe) est la source primaire de règle à respecter, veillez à bien la suivre.
- Vous pouvez exécuter à tout moment la classe pour déclencher une série de test. **ATTENTION** : les tests ne couvrent pas forcément tous les cas possibles.
- Répondre directement dans le fichier de la classe aux questions posées en commentaire dans le cartouche en amont de la définition de la classe.

VOUS NE DEVEZ IMPLEMENTER QUE LES MÉTHODES DE L'INTERFACE DANS LA CLASSE. Ces méthodes sont déclarées dans la classe mais ne sont pas encore implémentée (elle lèvent l'exception de type *UnsupportedOperationException*)

1.3. Rendu :

- Déposez sur umtice **uniquement** le fichier .java de la classe dont vous avez implémenté les méthodes et répondu aux questions.
- Ne compressez pas ce fichier dans une archive, n'ajoutez aucun autre fichier.
- Le non respect de ces règles entrainera automatiquement la note de 0.

2. Projet : le gestionnaire de rendez-vous

Vous devez implémenter un gestionnaire de calendrier de rendez-vous. Ce calendrier utilise une structure d'arbre binaire de recherche (ABR) rouge-noir déjà implémentée en Java : le `TreeSet`.

Pour rappel :

- Un Arbre binaire est un arbre qui a au plus 2 fils : un sous-arbre gauche (SAG) et un sous-arbre droit (SAD).
- Un ABR est un arbre binaire dont les valeurs des nœuds sont ordonnées : pour un nœud N de valeur V, toutes les valeurs des nœuds du SAG de N sont inférieures ou égales à V, et toutes les valeurs des nœuds du SAD de N sont strictement supérieures à V.
- Un ABR à valeur unique est un ABR dont une valeur est présente au plus une fois dans l'arbre. Les propriétés sont donc plus strictes : pour un nœud N de valeur V, toutes les valeurs des nœuds du SAG de N sont strictement inférieures à V, et toutes les valeurs des nœuds du SAD de N sont strictement supérieures à V.
- Un ABR rouge-noir est un ABR à valeurs uniques auto-équilibré : à chaque ajout/retrait de nœuds dans l'arbre, l'arbre se restructure de lui-même pour ne pas dégénérer.

2.1. Valeurs comparables

Les valeurs stockées dans un ABR doivent donc être comparables les une aux autres : Pour tous couple (a,b) de valeurs du type de données de l'ABR, il doit exister une "relation d'ordre stricte" :

- soit a < b
- soit a > b
- soit a = b

En java l'expression de cette relation d'ordre pour un type de données se fait par l'implémentation de l'interface générique `Comparable<K>` (avec généralement K le type de la classe qui implémente l'interface) qui impose l'implantation d'une méthode `int compareTo(K k)`.

Plus d'information sur l'interface CompareTo sur la Javdoc officielle.

Dans ce projet, les valeurs stockées dans l'arbre sont de type Appointment (cf. description ci-dessous). Cette classe implémente déjà l'interface Comparable.

2.1. TreeSet

Le TreeSet est une classe générique qui implémente un ABR à valeurs uniques rouge-noir. Les valeurs insérées dans un TreeSet doivent être comparable les unes au autres. Cette classe générique est utilisé dans ce projet avec le type Appointment, comparable à lui-même (cf. description ci-dessous).

La documentation de la classe TreeSet est à trouver sur la Javdoc officielle.

2.1.2. Le Rendez-vous : classe Appointment

Cette classe est déjà entièrement implémentée ; les informations suivantes sont fournies à titre indicatif.

Le rendez-vous est modélisé par la classe *Appointment*. les instances de cette dernière sont immuables (on ne peut pas modifier les valeurs de leurs attributs une fois l'instance créée) et comparables entre elles.

Un rendez-vous est modélisé par le attributs suivant :

- **start** : le début du rendez-vous de type *LocalDateTime*, datetime locale (date et temps). Le gestionnaire de rendez-vous s'assurera à la création d'un rendez-vous que cette valeur :
  - ne soit jamais null ;
  - soit tronquée à la minute (les secondes, millisecondes, etc. sont mises à 0).
- **durationMinutes** : la durée du rendez-vous en minutes de type *int*. Le gestionnaire de rendez-vous s'assurera à la création d'un rendez-vous que cette valeur :
  - ne soit jamais négative ou égale à 0 ;
  - ne dépasse pas 24h.
- **name** : le nom du rendez-vous. Le gestionnaire de rendez-vous s'assurera à la création d'un rendez-vous que cette valeur :
  - ne soit jamais nulle ;
  - ne soit jamais vide ;
  - ne soit jamais "blanche" : remplie uniquement de caractères blancs (espace, tabulation, saut de ligne...).
  - le nom enregistré sera au préalable tronqué de tout caractère blanc en son début ou à sa fin.

Un rendez-vous est comparable à un autre. Un rendez-vous est considéré égal à un autre si et seulement si les deux ont le même début ET on des noms égaux, sans respect de la casse.

Un rendez-vous A et considéré strictement inférieur à un rendez-vous B si et seulement si :

- le début de A est strictement inférieur (antérieur) à celui de B ;
- OU si A et B ont des débuts égaux et si le nom de A est lexigraphiquement strictement inférieur à celui de B (sans respect de la casse).

Un rendez-vous A et considéré strictement supérieur à un rendez-vous B si et seulement si :

- le début de A est strictement supérieur (postérieur) à celui de B ;
- OU si A et B ont des débuts égaux et si le nom de A est lexigraphiquement strictement supérieur à celui de B (sans respect de la casse).

2.1.3. : Le calendrier : classe AppointmentCalendarImpl

Cette classe à compléter propose d'ors-et-déjà :

- L'attribut **appointments**, de type `TreeSet<Appointment>` pour stocker les rendez-vous.
- La méthode privée utilitaire `truncateDateTimeToMinutes(LocalDateTime dateTime)`. Cette méthode permet de tronquer un dateTime de type *LocalDateTime* à la minute en retourne un nouveau dateTime tronqué. si le dateTime fournit est nul, cette méthode retourne null.

2.1.4 Tips :

- la classe TreeSet propose des méthodes très utiles pour simplifier l'implémentation des méthodes du calendrier : consultez bien sa Javadoc avant de vous lancer dans le développement d'une méthode.

- la classe LocalDateTime est une classe immuable (comme String) : toutes les méthodes sur une instance de LocalDateTime (ex.: plusDays, truncatedTo...) ne modifie pas l'instance mais en crééent une nouvelle.
- Plusieurs méthodes de la classe String pourraient vous être utiles : *empty*, *isBlank*, *trim*.
- Plusieurs méthodes de la classe LocalDateTime pourraient vous être utiles : *plusMinutes*, *isBefore*, *minusHours*.
- Pour calculer un nombre de minutes entre 2 LocalDateTime t1 et t2, vous pouvez simplement utiliser les méthodes `Duration.between(t1, t2).toMinutes()`, de la classe java.time.Duration.
- Pour générer une collection non modifiable à partir d'une collection existante *col*, il suffit d'utiliser la méthode Collections.unmodifiableCollection(col), fournie par la classe java.util.Collections.
- un rendez-vous avec un titre vide "" sera toujours strictement antérieur à tous les rendez-vous de même date de début.
- Comme la précision des dates de début sont à la minute, un rendez-vous de début *start* + 1 minute et de titre "" sera toujours strictement postérieur à tous les rendez-vous de début *start* mais toujours strictement antérieur à tous les rendez-vous de début > *start*.

3. Question (reporté sur la classe java, réponse à apporter directement dessus)

Question 1

Quelle est la complexité algorithmique de la méthode *add* ?

Question 2

Si nous souhaitons implémenter une méthode pour modifier le nom et/ou la date de début d'un rdv, et partant du principe que la classe Appointment propose un setter pour ces deux propriétés, pensez-vous que l'approche suivante soit correcte ?

```
public boolean updateName(localDateTime start, String oldName,
localDateTime newStart, String newName) throws IllegalArgumentException {
  // vérifier la validité des paramètres (non null, et non vide ou blanc
pour oldName et newName)
  if (/* échec vérification */) {
    throw new IllegalArgumentException("Propriété de rendez-vous
invalide");
  }

  Appointment apt;
  // Récupérer apt: l'instance du rdv correspondant au début start tronqué
à la minute et au nom oldName.trim()

  if (apt == null) {
    return false;
  }

  // Modifier les propriété de apt
  apt.setStart(/*newStart tronqué à la minuté*/);
```

```
  apt.setName(/* newName, tronqué de ses caractères blancs avant et après
*/);

  return true;
}
```