Exception : Agendas hiérarchiques
Corrigé

Nous allons modéliser des agendas simplifiés (exercices 1, 2, 3 et 4) et des groupes hiérarchiques d'agendas (exercice 5). L'exercice 6 demande de dessiner le diagramme de classe de l'application mais il est bien sûr conseillé de le dessiner et le faire évoluer au fil des exercices. L'exercice 7 propose de généraliser les agendas.

Exercice 1 : Agenda

Pour simplifier, on considère un agenda pour une année avec un seul rendez-vous possible par journée. Les créneaux sont repérés par un numéro de 1 à 366 correspondant au jour du rendez-vous. Par exemple, 10 correspond au créneau du 10 janvier.

Un agenda, modélisé par l'interface `Agenda`, fournit les quatre opérations suivantes :

- obtenir le nom d'un agenda,
- enregistrer pour un créneau un rendez-vous qui sera ici limité à une chaîne de caractères non vide (`String`). Ainsi on peut enregistrer le rendez-vous "Examen" pour le créneau 14. Si le créneau est déjà occupé, une exception `OccupeException` est levée,
- annuler le rendez-vous d'un créneau donné. Par exemple, on peut supprimer le rendez-vous du créneau 14. Si le créneau est libre, rien ne se passe. La valeur de retour de cette opération indique si l'agenda a été modifiée (valeur vrai, le rendez-vous a été annulé) ou non (valeur faux, le créneau était libre).
- obtenir le rendez-vous correspondant à un créneau. Si ce créneau ne contient pas de rendez-vous enregistré, une erreur est signalée [1] grâce à l'exception `LibreException`.

Les trois dernières opérations attendent un créneau valide. Si ce n'est pas le cas, elles lèveront l'exception `CreneauInvalideException`.

1.1. `CreneauInvalideException` est une exception non vérifiée car on s'attend à ce que l'appelant fournisse un créneau valide. Il n'est donc pas souhaitable que le compilateur lui dise que le créneau pourrait être invalide. Les exceptions `OccupeException` et `LibreException` sont vérifiées car l'appelant ne peut généralement pas connaître le contenu de l'agenda. Il est donc souhaitable que le compilateur lui signale qu'il pourrait y avoir une erreur qu'il devra prendre en compte (enregistrer un rendez-vous sur un créneau occupé ou obtenir le rendez-vous d'un créneau libre).

Écrire ces trois classes d'exception.

1. Un autre choix possible aurait été de retourner une valeur particulière. Ici **`null`** est un bon candidat pour indiquer qu'il n'y a pas de rendez-vous pour le créneau demandé.

Solution : Pour une exception vérifiée, le compilateur vérifie que le programmeur l'a bien prise en compte, sont en mettant un **`try ... catch`** (dans ce cas il la récupère pour la traiter) soit en mettant un **`throws`** (il dit explicitement qu'il la laisse se propager). Une exception est vérifiée si elle n'est sous-type ni de `RuntimeException`, ni de `Error`.

Ainsi, `OccupeException` et `LibreException` héritent de `Exception` pour être vérifiées. `CreneauInvalideExcepti` hérite de `RuntimeException`.

```java
public class OccupeException extends Exception {

    public OccupeException(String message) {
        super(message);
    }

    public OccupeException() {
        super();
    }

    public OccupeException(Throwable cause) {
        super(cause);
    }

    public OccupeException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

```java
public class LibreException extends Exception {

    public LibreException(String message) {
        super(message);
    }

    public LibreException() {
        super();
    }

    public LibreException(Throwable cause) {
        super(cause);
    }

    public LibreException(String message, Throwable cause) {
        super(message, cause);
    }

}
```

```java
/**
 * CreneauInvalideException indique qu'une date n'est pas valide.
 */
public class CreneauInvalideException extends IllegalArgumentException {

```

```
6       public CreneauInvalideException(String message) {
7           super(message);
8       }
9
10      public CreneauInvalideException() {
11          super();
12      }
13
14      public CreneauInvalideException(Throwable cause) {
15          super(cause);
16      }
17
18      public CreneauInvalideException(String message, Throwable cause) {
19          super(message, cause);
20      }
21
22  }
```

1.2. Exécuter le programme de test ExceptionsTest et corriger les éventuelles erreurs signalées.
Solution : Une erreur sera signalée si les exceptions ne respectent pas le caractère vérifié ou non.
1.3. L'interface agenda contient des erreurs. Les identifier et les corriger.
Solution : Il manque le **throws** pour les méthodes qui peuvent lever des exceptions vérifiées !

```
1   public interface Agenda {
2
3       /** Le plus petit créneau possible. */
4       int CRENEAU_MIN = 1;
5
6       /** Le plus grand créneau possible. */
7       int CRENEAU_MAX = 366;
8
9       /**
10       * Obtenir le nom de l'agenda.
11       * @return le nom de l'agenda
12       */
13      String getNom();
14
15      /**
16       * Enregistrer un rendez-vous dans cet agenda.
17       *
18       * @param creneau le créneau du rendez-vous
19       * @param rdv le rendez-vous
20       * @throws CreneauInvalideException si le créneau est invalide
21       * @throws IllegalArgumentException si nom vaut null
22       * @throws OccupeException si le créneau n'est pas libre
23       */
24      void enregistrer(int creneau, String rdv) throws OccupeException;
25
26      /**
27       * Annuler le rendez-vous pris à une creneau donnée.
28       * Rien ne se passe si le créneau est libre.
```

```
29          * Retourne vrai si l'agenda est modifié (un rendez-vous est annulé),
30          * faux sinon.
31          *
32          * @param creneau créneau du rendez-vous à annuler
33          * @return vrai si l'agenda est modifié
34          * @throws CreneauInvalideException si le créneau est invalide
35          */
36         boolean annuler(int creneau);
37
38         /**
39          * Obtenir le rendez-vous pris à une creneau donnée.
40          *
41          * @param creneau le créneau du rendez-vous
42          * @return le rendez-vous à le créneau donnée
43          * @throw LibreException si pas de rendez-vous à ce créneau
44          */
45         String getRendezVous(int creneau) throws LibreException;
46
47     }
```

1.4. Expliquer pourquoi Agenda a été modélisé par une interface plutôt qu'une classe.
Solution : Rien n'est dit sur la manière de stocker les rendez-vous. Ainsi, on ne peut pas écrire une classe concrète ni le code d'aucune des méthodes de l'agenda.

Écrire une classe abstraite ne serait pas judicieux car alors on serait obligé d'en hériter et on ne pourrait plus décider d'hériter d'une autre classe (héritage simple sur les classes en Java).

Exercice 2 : ObjetNomme

La classe `ObjetNomme` modélise un objet qui a un nom. Même si cette classe est complètement écrite, elle est déclarée abstraite car on considère qu'elle ne devra pas être utilisée telle quelle mais devra être spécialisée.

2.1. Compléter `ObjetNomme` : une exception `IllegalArgumentException` doit être levée si le nom fourni en paramètre du constructeur n'a pas au moins un caractère.
Solution : On pourrait ajouter une conditionnelle en début du constructeur. Il est cependant souhaitable de définir une méthode qui fait la vérification. En fait, nous en définissons 2, la première qui vérifier que le paramètre est non nulle (`verifierNonNull`), la seconde vérifie qu'une chaîne de caractères a au moins 1 caractères (`verifierChaineNonVide`).

Nous avons défini ces méthodes **`protected`** pour qu'elles puissent être utilisées pas les sous-classes.

Notons que la première méthode est déjà présente en Java. Il s'agit de la méthode `nonNull(Object)` de la classe utilitaire `Objects`.

```
1   /**
2    * Un objet nommé est un objet qui a un nom.
3    */
4   public abstract class ObjetNomme {
5
6       private String nom;
7
8       /**
```

```java
     * Initialiser le nom de l'agenda.
     *
     * @param nom le nom de l'agenda
     * @throws IllegalArgumentException si nom n'a pas au moins un caractère
     */
    public ObjetNomme(String nom) {
        verifierNonNull(nom);
        verifierChaineNonVide(nom);

        this.nom = nom;
    }


    /**
     * Obtenir le nom de cet objet.
     * @return le nom de cet objet
     */
    public String getNom() {
        return this.nom;
    }


    /**
     * Vérifier que la chaîne est de longueur strictement positive
     *
     * @throws IllegalArgumentException si la chaîne est vide
     */
    protected void verifierChaineNonVide(String chaine) {
        verifierNonNull(chaine);
        if (chaine.length() == 0) {
            throw new IllegalArgumentException("nom vide");
        }
    }


    /**
     * Vérifier que la poignée est non nulle.
     *
     * @throws IllegalArgumentException si la poignée est nulle
     */
    protected void verifierNonNull(Object p) {
        if (p == null) {
            throw new IllegalArgumentException("argument is null");
        }
    }

}
```

2.2. La tester avec la classe `ObjetNommeTest`.
Solution : Il suffit de l'exécuter...

Exercice 3 : AgendaAbstrait

La classe abstraite `AgendaAbstrait` a été définie. Elle réalise l'interface `Agenda` et hérite de

`ObjetNomme`. Elle permet donc de factoriser la définition du nom.

On constate que de nombreuses opérations de l'agenda prennent en paramètre un créneau. Il est nécessaire que ce créneau soit valide. Pour éviter d'écrire de nombreuses fois les mêmes instructions, on va définir une méthode [2] `verifierCreneauValide(int creneau)` qui lèvera l'exception `CreneauInvalideException` si le créneau n'est pas dans les limites attendues.

3.1. Écrire dans la classe `AgendaAbstrait` la méthode `verifierCreneauValide`.

Solution :

```java
/**
 * AgendaAbstrait factorise la définition du nom et de l'accesseur associé.
 */
public abstract class AgendaAbstrait extends ObjetNomme implements Agenda {

    /**
     * Initialiser le nom de l'agenda.
     *
     * @param nom le nom de l'agenda
     * @throws IllegalArgumentException si nom n'a pas au moins un caractère
     */
    public AgendaAbstrait(String nom) {
        super(nom);
    }


    /**
     * Est-ce que le créneau est valide ?
     */
    protected static void verifierCreneauValide(int creneau) {
        if (creneau < CRENEAU_MIN || creneau > Agenda.CRENEAU_MAX) {
            throw new CreneauInvalideException("pour créneau : " + creneau);
        }
    }

}
```

3.2. Expliquer l'intérêt d'avoir à la fois l'interface `Agenda` et la classe abstraite `AgendaAbstrait`.

Solution : La classe abstraite permet de factoriser du code (la gestion du nom, mais qui est déjà factorisé dans `ObjetNomme` et surtout `verifierCreneauValide`) mais oblige à hériter d'une classe et interdit donc d'hériter d'une autre classe. L'interface permet de garder la relation de sous-typage avec Agenda, laisse la possibilité d'hériter d'une classe mais ne permet pas de factoriser du code.

Exercice 4 : AgendaIndividuel

La classe `AgendaIndividuel` est une implantation de l'agenda pour laquelle on décide de stocker les rendez-vous dans un tableau.

4.1. Compléter la classe `AgendaIndividuel` pour ajouter les aspects liés aux exceptions.

2. En Java8, cette méthode pourrait être définie comme méthode de classe de l'interface `Agenda`. La classe `AgendaAbstrait` deviendrait inutile.

Solution :

1. Les rendez-vous sont stockés dans un tableau indicé par les créneaux de chaînes de caractères. Ceci est possible puisqu'il ne peut y avoir qu'un seul rendez-vous par créneau.
2. Deux méthodes sont définies pour vérifier qu'un créneaux est bien libre ou occupé.

```java
/**
 * Définition d'un agenda individuel.
 */
public class AgendaIndividuel extends AgendaAbstrait {

    private String[] rendezVous;    // le texte des rendezVous


    /**
     * Créer un agenda vide (avec aucun rendez-vous).
     *
     * @param nom le nom de l'agenda
     * @throws IllegalArgumentException si nom nul ou vide
     */
    public AgendaIndividuel(String nom) {
        super(nom);
        this.rendezVous = new String[Agenda.CRENEAU_MAX + 1];
            // On gaspille une case (la première qui ne sera jamais utilisée)
            // mais on évite de nombreux « creneau - 1 »
    }


    @Override
    public void enregistrer(int creneau, String rdv) throws OccupeException {
        verifierChaineNonVide(rdv);
        verifierLibre(creneau);

        this.rendezVous[creneau] = rdv;
    }


    @Override
    public boolean annuler(int creneau) {
        verifierCreneauValide(creneau);

        boolean modifie = this.rendezVous[creneau] != null;
        this.rendezVous[creneau] = null;
        return modifie;
    }


    @Override
    public String getRendezVous(int creneau) throws LibreException {
        verifierOccupe(creneau);

        return this.rendezVous[creneau];
    }
```

```java

    /**
     * Vérifier que le creneau est libre.
     * Lève l'exception OccupeException si ce n'est pas le cas.
     *
     * @throws OccupeException si le créneau est occupé
     */
    protected void verifierLibre(int creneau) throws OccupeException {
        verifierCreneauValide(creneau);
        if (this.rendezVous[creneau] != null) {
            throw new OccupeException(this.getNom() + "@" + creneau
                    + " : " + this.rendezVous[creneau]);
        }
    }


    /**
     * Vérifier que le creneau est occupé.
     * Lève l'exception LibreException si ce n'est pas le cas.
     *
     * @throws LibreException si le créneau est libre
     */
    protected void verifierOccupe(int creneau) throws LibreException {
        verifierCreneauValide(creneau);
        if (this.rendezVous[creneau] == null) {
            throw new LibreException("pour : " + this.getNom() + "@" + creneau);
        }
    }

}
```

4.2. La tester grâce à la classe `AgendaIndividuelTest`.
Solution : Il suffit d'exécuter la classe de test.

Exercice 5 : GroupeAgenda

Un groupe d'agendas est utilisé pour manipuler plusieurs agendas, qu'ils soient des agendas individuels ou des groupes d'agendas. Un groupe permet d'enregistrer un rendez-vous sur tous les agendas d'un groupe ou d'obtenir le rendez-vous commun à l'ensemble des agendas du groupe. Une opération permet d'ajouter un nouvel agenda dans un groupe.

L'opération « enregistrer » garantit que soit le rendez-vous est enregistré dans tous les agendas du groupe, soit dans aucun. Dans le cas où le rendez-vous n'a pas pu être enregistré, l'exception `OccupeException` est levée.

Quand on demande le rendez-vous d'un créneau, on obtient l'exception `LibreException` si tous les agendas sont libres pour ce créneau, **`null`** si deux agendas ont des rendez-vous différents ou le rendez-vous commun à tous les agendas.

5.1. Écrire la classe `GroupeAgenda` sachant que l'on doit utiliser l'interface `java.util.List` pour stocker les agendas contenus dans le groupe.
Solution :

1. Cette classe hérite de `AgendaAbstrait`. Ceci permet de récupérer la gestion du nom.
2. On doit faire attention à ne pas créer un cycle en ajoutant un agenda à un groupe. Pour ce faire, nous nous appuyons sur une méthode `contient`.
3. Cette méthode `contient` pour savoir si un groupe d'agendas contient directement ou indirectement un agenda. Pour ce dernier point, on doit faire un test sur le type des agendas du groupe car la méthode contient n'est pas présente sur l'interface `Agenda`.
4. Pour cette méthode, on utilise un `foreach` car il est plus pratique et efficace sur des listes. Nous utilisons donc un **`return true`** quand on a trouvé l'intérêt cherché. En fin de méthode, on peut retourner **`false`** puisqu'il n'a pas été trouvé si nous arrivons ici.
5. La méthode `enregistrer` doit enregistrer le rendez-vous dans tous les agendas ou dans aucun. La solution retenue consiste à mémoriser les agendas pour lesquels le rendez-vous a été enregistré. Si un rendez-vous ne peut pas être enregistré dans l'un des agendas (exception `OccupeException`), on utilise cette liste pour annuler le rendez-vous déjà enregistrés. Une fois tous les rendez-vous annulés, on relève la même exception qui continuera donc à se propager.
6. Dans la méthode `annuler`, un booléen nous dit si l'un des agendas a été modifié. Attention à l'ordre. Il faut bien écrire :

   ```
   modifie = a.annuler(creneau) || modifie;
   ```

   et non :

   ```
   modifie = modifie || a.annuler(creneau);
   ```

   car sinon l'évaluation en court-circuit des booléens ferait que l'appel à `annuler` n'aurait plus lieu dès qu'un agenda serait modifié.
7. Pour implanter la logique de `getRendezVous`, on compte le nombre de d'agendas libres pour savoir s'il faut lever l'exception `LibreException` (ils sont tous libres) ou le rendez-vous commun. Le rendez-vous commun est **`null`** dès que deux agendas ont des rendez-vous différents. Pour avoir un rendez-vous commun, il faut que tous les agendas aient le même rendez-vous (`rdvCommun`). Notons l'égalité logique (`equals`) utilisée pour comparer les rendez-vous.

```
1  import java.util.*;
2
3  /** @has 0..* "" 0..* Agenda */
4  public class GroupeAgenda extends AgendaAbstrait {
5
6      protected List<Agenda> agendas;
7          // Il serait plus logique d'utiliser un ensemble plutôt qu'une liste
8          // car il ne faut pas ajotuer deux fois le même agenda dans le même
9          // groupe.
10
11
12     /**
13      * Créer un groupe d'agenda vide (avec aucun rendez-vous).
```

```java
     *
     * @param nom le nom de l'agenda
     * @throws IllegalArgumentException si nom nul ou vide
     */
    public GroupeAgenda(String nom) {
        super(nom);
        this.agendas = new ArrayList<>();
    }

    /**
     * Ajouter un nouvel agenda dans ce groupe.
     * Attention, il ne faut pas ajouter plusieurs fois le même groupe dans un
     * même groupe.  Ceci ne peut pas être vérifier avec la modélisation
     * proposée si on ajoute des dans un sous-groupe un agenda qui existe
     * déjà dans le super-groupe.
     *
     * @param a le nouvel agenda à ajouter
     *
     * @throws IllegalArgumentException si a est null ou déjà dans le groupe
     */
    public void ajouter(Agenda a) {
        verifierNonNull(a);
        if (this.contient(a)) {
            throw new IllegalArgumentException("déjà dans le groupe "
                    + this.getNom() + " : " + a.getNom());
        }
        if (a instanceof GroupeAgenda && ((GroupeAgenda) a).contient(this)) {
            throw new IllegalArgumentException("refus de créer un cycle : "
                    + a.getNom() + " contient " + this.getNom());
        }

        this.agendas.add(a);
    }

    /** Savoir si un agenda appartient à ce groupe.
     * @param cherche l'agenda cherché
     * @return vrai si l'agenda appartient à ce groupe
     */
    public boolean contient(Agenda cherche) {
        for (Agenda a : this.agendas) {
            if (a.equals(cherche)) {
                return true;
            } else if (a instanceof GroupeAgenda) {
                GroupeAgenda sousGroupe = (GroupeAgenda) a;
                if (sousGroupe.contient(cherche)) {
                    return true;
                }
            }
        }
        return false;
    }

    /**
```

```
67          * {@inheritDoc}
68          * Le rendez-vous est soit enregistré dans tous les agendas du groupe,
69          * soit dans aucun.
70          */
71         @Override
72         public void enregistrer(int creneau, String rdv) throws OccupeException {
73             verifierCreneauValide(creneau);
74             verifierNonNull(rdv);
75             verifierChaineNonVide(rdv);
76 
77             List<Agenda> agendasModifies = new ArrayList<>();
78             try {
79                 // Enregistrer dans les différents agendas
80                 for (Agenda a : this.agendas) {
81                     a.enregistrer(creneau, rdv);
82                     agendasModifies.add(a);
83                 }
84             } catch (OccupeException e) {
85                 // Supprimer les enregisrements faits
86                 for (Agenda a : agendasModifies) {
87                     a.annuler(creneau);
88                 }
89                 throw e;
90                     // On relève la même exception, donc l'exception de l'agenda
91                     // individuel qui était occupé !
92             }
93         }
94 
95         @Override
96         public boolean annuler(int creneau) {
97             verifierCreneauValide(creneau);
98 
99             boolean modifie = false;
100            for (Agenda a : this.agendas) {
101                modifie = a.annuler(creneau) || modifie;    // Attention à l'ordre !
102            }
103            return modifie;
104        }
105 
106        @Override
107        public String getRendezVous(int creneau) throws LibreException {
108            verifierCreneauValide(creneau);
109 
110            String rdvCommun = null;
111            int nbLibres = 0;
112            for (Agenda a : this.agendas) {
113                try {
114                    String rdv = a.getRendezVous(creneau);
115                    if (rdv == null) {
116                        // C'est donc que a est un groupe sans rendez-vous commun
117                        return null;
118                    } else {
119                        if (rdvCommun == null) {
```

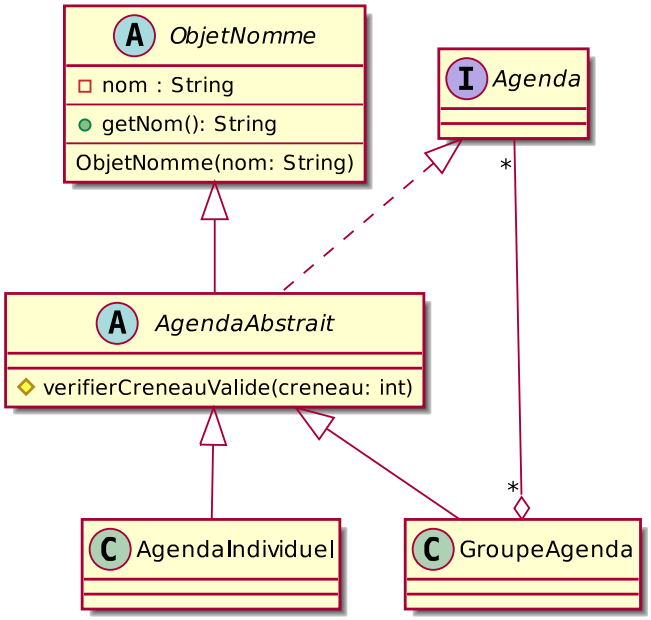
```
120                         rdvCommun = rdv;
121                     } else if (! rdv.equals(rdvCommun)) {
122                         // Deux rendez-vous différents !
123                         return null;
124                     } else {
125                     }
126                 }
127             } catch (LibreException e) {
128                 nbLibres++;
129             }
130         }
131         if (nbLibres == agendas.size()) {
132             throw new LibreException();
133         } else if (nbLibres > 0) {
134             return null;
135         } else {
136             return rdvCommun;
137         }
138     }
139
140     /**
141      * Proposer un rendez-vous à tous les agendas de ce groupe.
142      * Le rendez-vous n'est enregistré que dans les agendas qui sont libres
143      * pour le créneau considéré.  Il est ignoré par les autres.
144      *
145      * @param creneau le créneau du rendez-vous
146      * @param rdv le rendez-vous
147      * @throws CreneauInvalideException si le créneau est invalide
148      * @throws IllegalArgumentException si nom nul ou vide
149      */
150     public void proposer(int creneau, String rdv) {
151         verifierCreneauValide(creneau);
152         verifierNonNull(rdv);
153         verifierChaineNonVide(rdv);
154
155         for (Agenda a : this.agendas) {
156             if (a instanceof GroupeAgenda) {
157                 GroupeAgenda g = (GroupeAgenda) a;
158                 g.proposer(creneau, rdv);
159             } else {
160                 try {
161                     a.enregistrer(creneau, rdv);
162                 } catch (OccupeException e) {
163                     // Rien à faire
164                     assert ! (a instanceof GroupeAgenda);
165                 }
166             }
167         }
168     }
169 }
```

5.2. La tester avec la classe `GroupeAgendaTest`.
Solution : Il suffit d'exécuter la classe de test.

Exercice 6 : Diagramme de classe

Dessiner le diagramme de classe [3] de l'application faisant apparaître toutes les classes et interfaces de cette partie. On ne fera pas apparaître les constructeurs, attributs et opérations.

Solution :



Exercice 7 : Généralisation

Envisageons plusieurs extensions à notre application.

7.1. Définir une opération « proposer » sur un groupe d'agenda qui propose un nouveau rendez-vous à tous les agendas du groupe. Ce rendez-vous sera enregistré si l'agenda est libre sur le créneau considéré, ignoré sinon. La tester grâce à la classe `GroupeAgendaProposerTest`.

Solution : Cette méthode est implantée dans la classe `GroupeAgenda`.

7.2. Ajouter des informations [4] sur l'exception OccupeException : le créneau qui est occupé, le rendez-vous sur le créneau et le nom de l'agenda.

Solution : Il suffit de définir les attributs correspondants et de les initialiser via le constructeur. Bien sûr, on définira les accesseurs correspondants.

7.3. Souvent un agenda a peu de rendez-vous par rapport au nombre de créneaux disponibles. Par exemple, si on considère une année avec des créneaux de 15 minutes, on a plus de 35000 créneaux. Utiliser un tableau consomme donc beaucoup de place. Comment faire pour être plus économe en place mémoire tout en conservant des opérations efficaces pour les agendas ?

Solution : On peut utiliser une structure de données associative comme par exemple une table de hachage. La bibliothèque Java propose une interface Map (tableau associatif) qui a en particulier

3. Normalement, il a été construit au fur et à mesure de la réalisation des exercices précédents.

4. Attention aux informations qui sont associées à une exception car ceci peut conduire à violer le principe d'encapsulation. Par exemple, si un GroupeAgenda transmet celui de ses agendas occupé via une exception alors, quiconque récupèrera l'exception pourra modifier cet agenda occupé, et par exemple annuler le rendez-vous, sans passer par `GroupeAgenda`.

une implantation avec une table de hachage (`HashMap`). L'ancienne classe `Hashtable` ne devrait plus être utilisée.

7.4. Dans la solution proposée, un rendez-vous est réduit à une chaîne de caractères. Dans un autre contexte, on pourrait vouloir indiquer en plus un numéro de salle pour le rendez-vous. On pourrait aussi dans encore un autre contexte vouloir préciser un ordre du jour, etc.

Indiquer comment il serait possible de définir des agendas qui puissent simplement être adaptés à ces types de changement.

Solution : Il suffit de paramétrer les interfaces et classes décrivant les agendas par le type de l'information correspondant à une date. On s'appuie donc sur la généricité et on définit un paramètre de généricité.