TP 2 : INTERVALLES ET CONSTRAINTES

Introduction

Pour le deuxième TP, vous devez écrire le code de deux classes : Intervalle et Contrainte. Ces deux classes permettent la gestion de contraintes. Une contrainte est une technique afin de limiter les valeurs possibles dans un ensemble. Dans notre cas, les contraintes vont limiter l'ensemble des entiers ($\mathbb{Z}$). Dans notre cas, une contrainte est un ensemble d'intervalles disjoint. Les intervalles que nous utilisés sont des intervalles semi-fermés à gauche et semi-ouvert à droite.

Ces classes vont contenir des constructeurs, des méthodes de consultations et des opérateurs d'union, d'intersection et de soustraction. Nous allons tester ces services à l'aide de testes unitaire JUnit.

Description des TDAs

Intervalle

La classe `Intervalle` va décrire des intervalles semi-fermés à gauche et semi-ouverts à droite avec leurs services. Dans cette description, nous dénoterons un intervalle qui commence à la borne d (inclusive) et se termine à la borne f (exclusive) par la notation suivante : $[d..f[$. Il est important que d soit toujours plus petit que f. Il n'y a pas d'intervalle vide. Cette classe va contenir un constructeur acceptant les deux bornes en arguments.

Il est possible de consulter un intervalle afin de vérifier l'appartenance d'une valeur à l'intervalle. Nous disons que v appartient à l'intervalle : $[d..f[$ si et seulement si $d \leq v < f$. Un autre service important est la possibilité de comparer deux intervalles afin de vérifier s'ils sont égaux. Deux intervalles sont égaux si et seulement s'ils ont la même borne à gauche et la même borne à droite.

Contrainte

La classe `Contrainte` va permettre de décrire un ensemble d'intervalles disjoint. Pour cette description, nous dénoterons une contrainte c comme étant un ensemble d'intervalles : $c = \{r_1, r_2, \dots, r_n\}$. Ici l'ensemble contient n intervalle (r_i). Deux intervalles sont disjoints lorsqu'ils n'ont aucune valeur en commun. En d'autres termes, $r_1 = [d_1..f_1[$ est disjoint de $r_2 = [d_2..f_2[$ si et seulement si le prédicat suivant est vrai :

$$(\forall\, v_1 \in r_1, v_1 \notin r_2) \wedge (\forall\, v_2 \in r_2, v_2 \notin r_1)$$

Il y a deux constructeurs possibles pour cette classe. Un premier permet de construire une contrainte vide. C'est-à-dire, qui n'a pas d'intervalle. Le deuxième permet de construire une contrainte à partir d'une `Collection` d'intervalles. Il est important de remarquer que cette collection d'intervalles ne contient pas nécessairement des intervalles disjoints. Il faudra modifier les intervalles afin d'obtenir des intervalles disjoints dans la contrainte résultante afin qu'elle représente les mêmes valeurs que la collection de base.

La méthode d'observation 'appartient' (∈) est un premier service important pour une contrainte. Ce service vérifie si une valeur v appartient à une contrainte c. Nous disons qu'une valeur v appartient à une contrainte $c = \{r_1, r_2, \dots, r_n\}$ si et seulement s'il existe un $1 \leq i \leq n$ de tel sorte que $v \in r_i$.

Il faut aussi des services pour ajouter (`add`) et enlever (`remove`) des valeurs de la contrainte. Ces deux méthodes vont avoir comme paramètre un intervalle indiquant les valeurs à ajouter ou enlever.

- `add` : Soit une contrainte c et un intervalle r. Alors le résultat c^* = c.`add` (r) contient toutes les valeurs de c et r.

$$\forall v \in c^* \leftrightarrow v \in c \vee v \in r$$

- `remove` : Soit une contrainte c et un intervalle r. Alors le résultat c^* = c.`remove`(r) contient les valeurs de c qui ne sont pas dans r.

$$\forall v \in c^* \leftrightarrow v \in c \wedge v \notin r$$

Finalement, nous aurons un service qui permet de calculer le nombre de valeurs que représente une contrainte. Par exemple, la contrainte $\{\ [1..2[, [4..6[, [7..9[\ \}$ représente les valeurs $\{1, 4, 5, 7, 8\}$. Donc la taille de cette contrainte est 5, car la contrainte représente 5 valeurs.

Opérateurs

Finalement, les intervalles et les contraintes vont contenir les opérateurs d'`union` (∪), d'`intersection` (∩) et de `soustraction` (-). Les mêmes définitions de ces opérateurs sont utilisées pour les intervalles et les contraintes.

Soit deux ensembles e_1 et e_2.

- $e_U = e_1 \cup e_2 \leftrightarrow \forall\ v \in e_U, v \in e_1 \vee v \in e_2$
- $e_I = e_1 \cap e_2 \leftrightarrow \forall\ v \in e_I, v \in e_1 \wedge v \in e_2$
- $e_S = e_1 - e_2 \leftrightarrow \forall\ v \in e_S, v \in e_1 \wedge v \notin e_2$

Code

Vous devez prendre le code de départ sur Moodle et compléter les classes en écrivant le code pour chaque méthode.

- Vous pouvez ajouter des méthodes au besoin. Dans ce cas, n'oubliez pas de commenter vos méthodes.
- Vous pouvez ajouter des variables d'instances et de classes dans votre code.

- La classe Contrainte hérite d'un ArrayList<Intevalle>. Vous devez utiliser cet héritage à votre avantage. Il est interdit de placer une variable d'instance de type ArrayList dans la classe Contrainte.
- Vous devez surcharger la méthode equals dans la classe Intervalle. Vous devez ajouter la signature dans le code, elle n'est pas présente.
- Je vous conseille de surcharger la méthode toString dans la classe Intervalle, très pratique pour corriger les erreurs.

Votre code est testé à l'aide de tests unitaires JUnit. Deux fichiers de tests sont disponibles sur Moodle. L'environnement de développement va vous permettre de les lancer individuellement. Il est conseillé de construire vos méthodes dans cet ordre :

- Intervalle
    1. Constructeur
    2. appartient
    3. equals
- Contrainte
    1. Constructeur vide
    2. Constructeur Collection
    3. appartient
    4. add
    5. remove
    6. taille
- Intervalle et Contrainte
    1. union
    2. intersection
    3. soustraction

Directives

1. Le TP est à faire seul ou en équipe de deux.
2. Code :
    a. Pas de `goto`, `continue`.
    b. Les `break` ne peuvent apparaitre que dans les `switch`.
    c. Un seul `return` par méthode.
3. Indentez votre code. Assurez-vous que l'indentation est faite avec des espaces.
4. Commentaires
    - Commentez l'entête de chaque classe et méthode.
    - Une ligne contient soit un commentaire, soit du code, pas les deux.
    - Utilisez des noms d'identificateur significatif.
    - Une ligne de commentaire ne devrait pas dépasser 80 caractères. Continuez sur la ligne suivante au besoin.

- Nous utilisons Javadoc :
    - La première ligne d'un commentaire doit contenir une description courte (1 phrase) de la méthode ou la classe.
        - Courte.
        - Complète.
        - Commencez la description avec un verbe.
        - Assurez-vous de ne pas simplement répéter le nom de la méthode, donnez plus d'information.
    - Ensuite, au besoin, une description détaillée de la méthode ou classe va suivre.
        - Indépendant du code. Les commentaires d'entêtes décrivent ce que la méthode fait, ils ne décrivent pas comment c'est fait.
        - Si vous avez besoin de mentionner l'objet courant, utilisez le mot 'this'.
    - Ensuite, avant de placer les **tags,** placez une ligne vide.
    - Placez les **tag** @param, @return et @throws au besoin.
        - @param : décris les valeurs acceptées pour la méthode.
    - Dans les commentaires, placer les noms de variable et autre ligne de code entre les tags <code> … </code>.
    - Écrivez les commentaires à la troisième personne.

Remise

Remettre le TP par l'entremise de Moodle. Placez vos fichiers `'*.java'` dans un dossier compressé (**zip**) de **Windows**, vous devez remettre l'archive. Le TP est à remettre avant le 17 novembre 23:55.

Évaluation

- Fonctionnalité (9 pts) : les tests JUnit sont disponibles sur Moodle.
- Structure (2 pts) : veillez à utiliser correctement le mécanisme d'héritage et de méthode. Vous devez utiliser correctement l'héritage du ArrayList dans la classe Contrainte.
- Lisibilité (2 pts) : commentaire, indentation et noms d'identificateur significatif.