TP

Nous allons développer un programme Java permettant de jouer au jeu de Nim en mode console. Nous commencerons par développer une version dans laquelle deux joueurs humains s'affrontent puis ensuite nous ferons évoluer le programme de manière à permettre à un joueur humain de jouer contre l'ordinateur.

Pour jouer au jeu de Nim, nous choisissons un nombre de lignes au départ et nous plaçons un nombre impair d'allumettes par ligne. Sur la ième ligne, nous placerons `2*i - 1` allumettes(i>=1).
A chaque tour, un des joueurs sélectionne une ligne non vide et retire au moins une allumette.
Le joueur gagnant est celui qui prend la dernière allumette.

Par exemple, si on choisit 3 tas, l'état initial de la partie sera :

Tas 1 -> 1 allumette
Tas 2 -> 3 allumettes
Tas 3 -> 5 allumettes

Affichage des tas :

```
   |
  |||
|||||
```

I. Version Humain contre Humain

Le scénario :
Avant de commencer à jouer, le programme demande le nombre de tas avec lequel on souhaite jouer.
L'utilisateur saisit un nombre de tas qui devra être >=1.
Le programme demande le nom du joueur 1.
Le joueur 1 saisit son nom.
Le programme demande le nom du joueur 2.
Le joueur 2 saisit son nom.
La partie peut alors commencer.
Le programme affiche l'état initial de la partie.
Le joueur 1 commence.
Avant chaque coup, le programme affichera le nom du joueur qui doit jouer (ex : Dupont : à vous de jouer un coup sous la forme '`m n`' où `m` est la ligne choisie et `n` le nombre d'allumettes à retirer sur cette ligne.)
A chaque fois qu'un joueur joue son coup, il faudra vérifier la validité du coup avant de modifier l'état de la partie. Si le coup est invalide, on en informe le joueur et on lui propose de rejouer. Si le coup est valide, on affiche le nouvel état de la partie.
Le programme doit ensuite tester s'il reste des allumettes dans le jeu. S'il en reste, c'est à l'autre joueur de jouer sinon le joueur qui vient de jouer a gagné.
Le programme affiche le nom du gagnant et propose de rejouer une partie.
Si les joueurs décident de rejouer une partie, le programme affiche de nouveau l'état initial des tas défini au départ et les joueurs rejouent une partie. S'ils décident d'arrêter, le programme affiche le nombre de parties gagnées par chaque joueur et le nom du vainqueur ou "ex aequo" si les deux joueurs ont remportés le même nombre de parties.

Choix des classes:
On va répartir les classes en trois packages qui seront nommés : `modele`, `vue`, `controleur`.

Classes du package `modele`:
`Joueur` : caractérisé par un nom et un nombre de parties gagnées.
`Coup` : caractérisé par un numéro de tas et un nombre d'allumettes à retirer sur ce tas.
`Tas` : enregistre le nombre d'allumettes de chaque tas sous la forme d'un tableau d'entiers.

Toutes les classes devront respecter le principe d'encapulation et fournir les services nécessaires à la manipulation des données qu'elles contiennent.

On peut également noter que dans la classe `Tas`, la méthode permettant de changer l'état d'un objet de type `Tas` suite à un coup devra vérifier la validité du coup avant d'appliquer ce changement et faire en sorte de permettre de remonter l'erreur au niveau de la vue pour que le joueur puisse ressaisir un autre coup.

Classe du package `vue`:
`Ihm` : intéragit avec l'utilisateur en affichant des messages et en récupérant les informations qu'il saisit. Informations qu'il transmet à la classe `ControleurJeu`.

Classe du package `controleur` :
`ControleurJeu` : gère le jeu, c'est à dire l'enchaînement des parties jusqu'à ce que les joueurs décident d'arrêter. A sa création, le `controleurJeu` reçoit l'objet de type `Ihm` et crée les deux joueurs en intéragissant avec l'ihm.

On ajoute un package `nim` contenant une classe `JeuNim` qui contient le `main`. Voici le code de la méthode `main` :

```
public static void main(String[] args) {
     Ihm ihm = new Ihm();
     ControleurJeu controleurJeu=new ControleurJeu(ihm);
     controleurJeu.jouer();
}
```

On notera que seules les méthodes de l'objet de type `Ihm` permettront d'intéragir avec l'utilisateur. En dehors de cette classe, il n'y aura aucune utilisation de `System.out.println` et d'un objet de type `Scanner`.
Vous devez prévoir les situations où l'utilisateur saisit des données invalides comme par exemple, un caractère à la place d'une entier, un entier négatif à la place d'un entier positif.
La classe `Ihm` contient un attribut de type `ControleurJeu` qui lui permettent si besoin de transmettre les informations saisies par l'utilisateur à ce controleur en appelant des méthodes que vous devez définir.

II. Version Humain contre Humain ou Humain contre Ordinateur

Nous vous demandons pour cette seconde version de reprendre le code déjà réalisé et de le compléter. Essayer de choisir les solutions qui permettent d'étendre le code pour ajouter les fonctionnalités demandées en limitant au maximum les modifications du code existant. En particulier, essayez de ne pas modifier la classe `ControleurJeu`. Par contre, vous pouvez ajouter

de nouvelles classes.

Le scénario :
Le programme demande si on souhaite jouer contre l'IA.
L'utilisateur fait son choix.
Si l'utilisateur répond non, on reprend le scénario de la première version sinon
Le programme demande le nombre de tas avec lequel on souhaite jouer.
L'utilisateur saisit un nombre de tas qui devra être >=1.
Le programme demande le nom du joueur 1.
Le joueur 1 saisit son nom.
La partie peut alors commencer.
Le programme affiche l'état initial de la partie.
Le joueur 1 commence.
La partie se déroule comme précédemment excepté le fait que le programme demande le coup du joueur 1 puis ensuite le coup du joueur 2 est joué par l'ordinateur et affiché sous la forme :
"**IA joue : `m n`"** où `m` est le tas choisi et `n` le nombre d'allumettes à retirer dans ce tas".

Le deuxième joueur étant l'ordinateur, il faut définir une stratégie pour choisir le coup à jouer par l'ordinateur.
On vous demande de programmer une stratégie permettant à l'ordinateur d'identifier une situation gagnante, c'est à dire une situation qui lui permet d'être sûr de gagner, en la transformant en situation perdante pour l'adversaire.

Cette stratégie est basée sur la représentation binaire des entiers.

Pour gagner, l'ordinateur doit laisser à l'adversaire une situation dans laquelle le OU exclusif du nombre d'allumettes de chacun des tas vaut 0. Appelons cette situation, situation paire.
L'ordinateur se retrouvera donc dans une situation gagnante si elle est impaire, c'est à dire que le OU exclusif du nombre d'allumettes de chacun des tas est différent de 0. Dans ce cas, il pourra toujours enlever des allumettes pour se ramener à une situation paire qui laissera l'adversaire en situation perdante.
De son côté, l'adversaire, en retirant des allumettes dans un des tas, transformera toujours cette situation paire en situation impaire permettant ainsi à l'ordinateur de retrouver une situation gagnante.

Quand c'est au tour de l'IA de jouer, il faut donc calculer le OU exclusif du nombre d'allumettes de chacun des tas. On obtient un entier nommé `resultatXor`.

- S'il est égal à 0, l'ordinateur n'est pas dans une situation gagnante. Il va donc jouer un coup quelconque. On fixe qu'il va retirer une allumette sur le premier tas contenant des allumettes.
- S'il est différent de 0, il est dans une situation gagnante.
  Dans ce cas, pour chaque tas, il doit rechercher le nombre d'allumettes qu'il devrait laisser dans ce tas pour que le joueur se retrouve en situation perdante. On sait qu'au moins un tas lui permet de le faire. Dès qu'il l'a trouvé, il retourne le coup correspondant.
  Pour calculer le nombre d'allumettes `nb` qu'il doit laisser dans le tas, il fera le OU exclusif du nombre d'allumettes du tas avec `resultatXor`. Si `nb` est stritement inférieur au nombre d'allumettes du tas, on peut en déduire le nombre d'allumettes à retirer pour laisser `nb` allumettes dans le tas.
  Vous trouverez ci-dessous un exemple pour illustrer l'identification d'une situation gagnante pour l'IA et le calcul du coup à jouer.

  Supposons qu'on ait défini une partie avec 3 tas et que le joueur 1 ait pris une allumette dans

le deuxième tas (coup : 2 1).

```
  |
 | |
| | | | |
```

Calculons le OU exclusif du nombre d'allumettes de chacun des tas :
`resultatXor` = 1 ⊕ 10 ⊕ 101 = 110
`resultatXor` **≠ 0**.
Donc l'IA est dans une situation gagnante. Il peut donc jouer de manière à laisser une situation perdante pour le joueur 1.
Pour chacun des tas, on calcule le nombre d'allumettes à laisser pour que le joueur se retrouve en situation perdante.
Tas 1 : 1 ⊕ 110 = 111 Il faudrait laisser 6 allumettes dans ce tas. Il n'y en a qu'une . Impossible
Tas 2 : 10 ⊕ 110 = 100 Il faudrait laisser 4 allumettes dans ce tas. Il n'y en a que 2 . Impossible
Tas 3 : 101 ⊕ 110 = 11 Il faudrait laisser 3 allumettes dans ce tas. Il y en a 5. Donc l'IA va retirer 2 allumettes dans le tas 3. Il joue 3 2.

```
  |
 | |
| | |
```

Calculons le OU exclusif du nombre d'allumettes de chacun des tas :
`resultatXor` = 1 ⊕ 10 ⊕ 11 = 0
Le joueur 1 est en situation perdante. Quelque soit son coup, il laissera une situation gagnante à l'IA.

III. Version avec contrainte sur le nombre d'allumettes

On souhaite compléter la version précédente en donnant la possibilité de limiter à 3 le nombre d'allumettes que l'on peut retirer à chaque coup.

Dans cette version, après avoir choisi le nombre de lignes pour la partie, le programme proposera de jouer cette partie avec une contrainte sur le nombre d'allumettes à retirer (max 3).
Il faudra donc vérifier, avant de valider le coup d'un joueur humain, qu'il respecte bien cette contrainte.

Concernant l'ordinateur, on devra implémenter une autre stratégie.
On va implémenter une stratégie naïve puisque l'ordinateur va jouer des coups aléatoires.
Le coup à jouer sera déterminé en sélectionnant celui-ci parmi la liste des coups possibles qui doit être calculée.
Pour gérer cette liste, vous pourrez utiliser l'interface `List<E>` de l'api Java et choisir la classe `ArrayList<E>` qui l'implémente en utilisant un tableau redimensionnable.

Pour choisir aléatoirement un indice dans cette liste, vous pourrez utiliser la méthode `public static double random()` de la classe `Math` qui retourne une valeur de type double supérieure ou égale à `0` et strictement inférieure à `1`.
On souhaite écrire un code qui puisse évoluer facilement dans le cas où l'on envisagerait de nouvelles stratégies pour l'ordinateur.
On va donc encapsuler ce qui est susceptible de varier, à savoir la stratégie, dans une interface

`IStrategie` qui fournira une méthode :
`public Coup appliquerStrategie( Partie partie).`

L'implémentation concrète de chaque stratégie se traduira par la création d'une classe implémentant l'interface `Istrategie.`