Université Lille1 — Licence mention Informatique — Semestre 4

**UE Programmation Orientée Objet**
**Examen première session – 27 mai 2011**

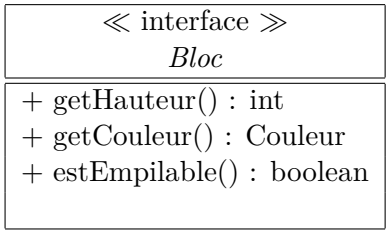
3 heures - notes et polycopiés de cours/TD/TP autorisés
livres, calculatrices et appareils de communication interdits
dictionnaires de langue autorisés pour candidats étrangers

---

▷ Les exercices sont totalement indépendants et l'ordre dans lequel ils apparaissent ne préjuge pas de leur difficulté réelle ou supposée.

**Exercice 1 : Un monde de blocs**

(Sauf indication contraire, les types de cet exercice appartiendront au paquetage `blocs`.)
Nous considérons dans cet exercice un environnement constitué de blocs avec lesquels il est possible de construire des empilements. Les blocs sont définis par l'interface suivante :




- `getHauteur` fournit la hauteur du bloc,
- `getCouleur` fournit la couleur du bloc,
- `estEmpilable` indique si il est ou non possible d'empiler quelque chose sur ce bloc,

Il y a 4 couleurs possibles pour les blocs : bleu, rouge, vert, jaune.

**Q 1 .** Donnez le code JAVA pour le type `Couleur` du package `blocs.util`.

**Q 2 .** Donnez le code JAVA de l'interface `Bloc`

Il existe des blocs de différentes formes : cubes, cylindres et cônes, auxquelles correspondent les classes : `Cube`, `Cylindre` et `Cone`. Les cônes ont la particularité de ne pas être empilables (on ne peut rien poser dessus).

**Q 3 .** Donnez un code JAVA pour la classe `Cube`. Hauteur et couleur d'un cube sont fixées à la création.

**Q 4 .** La classe `Empilement` permet de représenter des empilements de blocs. La pile des blocs est gérée par un attribut de type `java.util.Stack<Bloc>` (voir l'annexe pour la classe `java.util.Stack<E>` où `E` représente le type des éléments).

Cette classe dispose des méthodes :

- `getHauteur` qui renvoie la hauteur de l'empilement. Cette hauteur correspond au cumul des hauteurs des blocs qui composent l'empilement.
- `ajoute` qui permet d'ajouter au sommet de l'empilement un bloc passé en paramètre. Une exception `EmpilementImpossibleException` est déclenchée si le bloc au sommet n'est pas empilable[1].
- `retire` qui retire le bloc au sommet de l'empilement, le bloc retiré est renvoyé en résultat, une exception `java.util.EmptyStackException` est déclenchée si l'empilement est vide.
- `raz` qui retire un par un tous les blocs

Donnez un code JAVA pour cette classe `Empilement` (un empilement est initialement vide).

**Q 5 .** Donnez le code d'une méthode `main` d'une classe `Test` qui

[1] Cette classe d'exception dispose d'un constructeur qui prend en paramètre un message sous forme d'une chaîne de caractères.

1. crée 1 cube bleu de hauteur 4, 1 cylindre rouge de hauteur 2 et un cone bleu de hauteur 5,
2. crée un empilement `pile`
3. tente d'ajouter à `pile` successivement le cube, le cone et le cylindre ci-dessus, si un des ajouts est impossible un messsage est affiché, sans interruption du programme,
4. affiche la hauteur de l'empilement créé,

**Exercice 2 : Tournois et classements**

(Les types de cet exercice appartiendront au paquetage `competition`.)

Des disciplines sont organisées en fédérations qui gèrent ses adhérents en leur délivrant une licence, on parle alors de *licenciés d'une fédération*. Cette licence donne le droit de participer aux tournois organisés par la fédération.

Dans certains cas (tennis, tennis de table, échecs, etc.) un joueur se voit attribuer des points en fonction des résultats qu'il obtient à l'occasion de sa participation à ces tournois, en fonction des matchs qu'il y remporte.

On s'intéressera donc dans cet exercice à la gestion de tournois et à l'attribution de ces points.

### Les Joueurs

Un joueur est caractérisé par un nom, un prénom, un numéro de licence (une chaîne de caractères composée de chiffres et de lettres et supposée unique) et un nombre de points (initialement 0).

**Q 1 .** Donnez le code Java d'une classe `Joueur` conforme à cette description :

- attributs, constructeurs (à partir des nom, prénom et numéro de licence), méthodes accesseurs et la méthode `toString` (reprenant les valeurs des différents attributs),
- des méthodes `equals` et `hashCode` adaptées,
- une méthode pour ajouter des points au joueur,
- enfin cette classe implémente l'interface `Comparable`, un joueur sera considéré "plus petit" qu'un autre si son nombre de points est inférieur (NB : on accepte ici que cette comparaison ne soit pas consistante dans le cas de l'égalité avec `equals`).

### Les matchs

Un match se joue entre 2 joueurs et se conclut nécessairement par un vainqueur.

La classe `Match` est conforme au diagramme UML suivant :

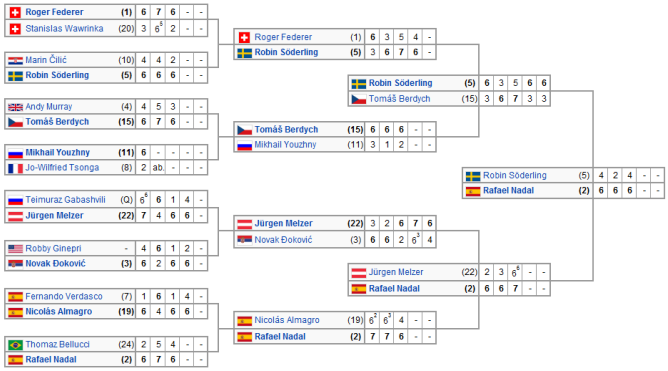
| **Match** |
|---|
| - joueur1 : Joueur<br>- joueur2 : Joueur<br>- vainqueur : Joueur |
| + Match(j1 : Joueur, j2 : Joueur)<br>+ getJoueur1() : Joueur<br>+ getJoueur2() : Joueur<br>+ getVainqueur() : Joueur<br>+ getVaincu() : Joueur<br>+ setVainqueur(vainqueur : Joueur) |

- Les méthodes `getVainqueur` et `getVaincu` lèvent une exception `java.lang.IllegalStateException` si le match n'a pas encore été joué et donc que le vainqueur n'est pas encore connu.
- La méthode `setVainqueur` lève une exception `java.lang.IllegalArgumentException` si son paramètre n'est pas l'un des deux joueurs du match.

**Q 2 .** Donnez la JavaDoc et le code Java des méthodes `getVainqueur` et `setVainqueur`.

## Les tournois

Un tournoi est organisé en "tours" (cf. image ci-dessous). On joue les tours les uns après les autres, un tour ne pouvant être commencé avant que tous les matchs du tour précédent ne soient terminés. Chaque tour comporte un nombre de matchs égal à une puissance de 2.




Les matchs d'un tour sont calculés en fonction des résultats du tour précédent : le premier match oppose les vainqueurs des matchs 1 et 2 du tour précédent, le second les vainqueurs des matchs 3 et 4, etc. A chaque tour on a donc 2 fois moins de matchs qu'au tour précédent. On parle de $\frac{1}{32}^{\text{ème}}$ lorsqu'il y a 32 matchs à jouer, puis de $\frac{1}{16}^{\text{ème}}$ pour les 16 matchs du tour suivant et ainsi de suite jusqu'aux $\frac{1}{2}$ finales et enfin la finale dont le vainqueur est le vainqueur du tournoi.

La classe `Tournoi` permet de modéliser de tels objets. Les attributs suivants sont définis dans cette classe `Tournoi` :

```
/** le nombre de points forfaitaire bonus attribué au vainqueur d'un tournoi
 */
public static final int NB_POINTS_VICTOIRE = 20;
/** vaut true si et seulement si ce tournoi est terminé
 */
private boolean fini;
/** liste des matchs qui ont déjà été joués dans ce tournoi, (NB : les matchs
 * ayant été joués leurs vainqueurs sont donc connus)
 */
private List<Match> lesMatchsJoues;
/** tableau de tous les matchs du tour en cours, lorsque le tournoi est
 * fini, ce tableau ne contient que le match de la finale
 */
private Match[] tourEnCours;
```

**Q 3 .** Un objet de cette classe `Tournoi` est construit à partir d'un paramètre de type `List<Joueur>` qui contient la liste des joueurs participant au tournoi. On supposera que la taille de cette liste est une puissance de 2, sans le vérifier. A la construction, le tableau des matchs du premier tour est créé par tirage aléatoire des joueurs (cf. en annexe un extrait de la documentation de la classe `java.util.Random`) .

Donnez un code JAVA pour un tel constructeur de `Tournoi`.

**Q 4 .** Donnez le code d'une méthode `joueUnTour` qui, si le tournoi n'est pas fini,

1. fait jouer tous les matchs de `tourEnCours`, ce qui revient à déterminer le vainqueur de chacun de ces matchs. Dans votre réponse, le vainqueur sera choisi aléatoirement entre les deux joueurs. Les matchs sont alors ajoutés aux matchs joués.
2. si le tour en cours était la finale indique que le tournoi est fini sinon, construit les matchs du tour suivant en mettant à jour le tableau des matchs pour le tour suivant.

La méthode n'a aucun effet si le tournoi est fini.

**Q 5 .** Donnez le code d'une méthode `getVainqueur` qui renvoie le joueur vainqueur du tournoi si le tournoi est fini, sinon une exception `TournoiNonFiniException` est levée.

On suppose que la classe `Tournoi` dispose également de la méthode :

- `matchsJoues()`

```
/** renvoie la liste des matchs joués pour un tournoi fini
 *
 * @return la liste de tous les matchs joués pour ce tournoi
 * @exception TournoiNonFiniException si le tournoi n'est pas fini
 *                 (càd qu'il reste des matchs à jouer)
 */
```

La classe d'exception `TournoiNonFiniException` est supposée défini. Elle contient un constructeur sans paramètre.

### La fédération et le classement.

La fédération regroupe les différents joueurs licenciés et est chargée de l'attribution des points en fonction du résultat aux tournois.
Les joueurs sont stockés dans une table de hachage, `lesLicencies` qui associe à un numéro de licence le joueur correspondant.
La classe `Federation` dispose d'une méthode

```
public static String getNouveauNumeroLicence()
```

qui à chaque appel fournit un nouveau numéro de licence unique (quelque soit la fédération).

Avec les objets de la classe `Federation`, on souhaite pouvoir inscrire un nouveau licencié à partir d'un nom et d'un prénom. Le joueur est alors créé et le numéro de licence du nouveau joueur est alors fixé à l'aide de la méthode ci-dessus.
Il faut également à partir d'un numéro de licence pouvoir :

- supprimer un licencié,
- connaître le nom du joueur licencié associé au numéro de licence,
- obtenir le nombre de points du joueur associé au numéro de licence.

Comme cela a été dit la fédération attribue des points aux joueurs en fonction de leurs résultats dans les tournois auxquels ils participent[2].

Si le tournoi est fini, pour chacun des matchs joués pendant le tournoi, on attribue 5 points au vainqueur du match. De plus si **avant** le tournoi, le vainqueur du match avait moins de points que le vaincu, ce vainqueur reçoit 1 point par tranche entière de 25 points d'écart. Ainsi, par exemple, si *tim* qui avait 132 points avant le tournoi est vainqueur d'un match contre *oleon* qui lui avait 197 points avant le tournoi, alors pour ce match *tim* reçoit 5 points de victoire plus 2 points pour les 65 points d'écart initial, donc en tout 7 points. Ce calcul est réalisé par la méthode :

```
private int calculPointsGagnes(Joueur vainqueur, Joueur vaincu)
```

Pour gérer correctement la prise en compte des points **avant** le tournoi, il faut calculer une table des points acquis par les joueurs pendant le tournoi. Ce qui est réalisé par la méthode :

```
private Map<String,Integer> calculPointsAcquis(Tournoi tournoi)
```

Dont la table (`Map`) résultat fournit en fonction du numéro de licence le nombre de points acquis pour le tournoi indiqué.

L'attribution effective aux joueurs des points gagnés au cours d'un tournoi est réalisé par la méthode `attribuePoints` qui prend en paramètre un objet `Tournoi`. Cette méthode déclenche l'exception `TournoiNonFiniException` si le tournoi n'est pas fini.
La méthode `attribuePoints` utilise les deux méthodes privées ci-dessus pour ajouter les points gagnés par les joueurs dans la table `lesLicencies` de la fédération.
Le vainqueur du tournoi reçoit en plus le nombre de points bonus défini dans `NB_POINTS_VICTOIRE` de la classe `Tournoi`.

**Q 6 .** Donnez le code JAVA d'une telle classe `Federation`.

**NB :** Aucun code n'est demandé pour la méthode `getNouveauNumeroLicence()`, mettre { ... } pour le corps de méthode.

[2]Le calcul de point proposé ici ne correspond a priori à aucune situation réelle.

# Annexe

## Extrait de la javadoc de `java.util.Stack<E>`

La classe `java.util.Stack<E>` implémente les interfaces `Collection<E>` et `Iterable<E>`.
En plus de ces méthodes de ces interfaces, les méthode suivantes sont ajoutées à la classe `Stack<E>` :

- `Stack()` Creates an empty Stack.
- `boolean empty()` Tests if this stack is empty.
- `E peek()` Looks at the object at the top of this stack without removing it from the stack. Throws `EmptyStackException` if this stack is empty.
- `E pop()` Removes the object at the top of this stack and returns that object as the value of this function. Throws `EmptyStackException` if this stack is empty.
- `E push(E item)` Pushes an item onto the top of this stack.

## Extrait de la javadoc de `java.util.Random`

- `Random()` Creates a new random number generator. This constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor.
- `int nextInt(int n)` Returns a pseudorandom, uniformly distributed int value between 0 (inclusive) and the specified value (exclusive), drawn from this random number generator's sequence.