Seul document autorisé : une feuille A4 recto-verso manuscrite (non photocopiée). Aucune machine.
Le barème est indicatif. Une question peut toujours être traitée en utilisant les précédentes (traitées ou non).
Les morceaux de code Java devront être clairement présentés, indentés et commentés.
L'utilisation de collections de l'API Java (en particulier `LinkedList` et `ArrayList`) est interdite.

- Tous les attributs d'instance de toutes les classes introduites doivent être privés.
- Il est possible d'écrire des méthodes non spécifiquement demandées si c'est plus pratique, en spécifiant pour chaque méthode dans quelle classe elle se trouve.

Au fast-food "Le gland de chêne", les sandwichs sont conçus à la demande : le client commande auprès du caissier, celui-ci se tourne alors vers les préparateurs situés derrière lui en cuisine, leur crie le nom du sandwich et encaisse le client. En cas de grande affluence ce mode de fonctionnement à l'ancienne a des ratés. Le gérant décide donc d'informatiser la situation et c'est ce que nous allons modéliser dans cette épreuve.

Sandwichs

Il y a 20 types de sandwichs, chacun étant identifié par un entier entre 0 et 19. Pour des raisons de comptabilité, on veut connaître à la fin de la journée le nombre de sandwichs de chaque type commandés.

Un sandwich sera donc représenté par une instance de la classe **`Sandwich`**. Cette classe contient (au moins) un attribut privé entier **`type`** qui représente le type d'un sandwich et un attribut de type tableau d'entiers **`commandes`** permettant de stocker le nombre de sandwichs de chaque type déjà commandés.

L'interface de cette classe est donnée par (sans que soit précisé le caractère statique ou non de ces méthodes) :

```
/** renvoie le type du sandwich */
int type();

/** renvoie le nombre de sandwichs commandés du type donné en argument */
int nbSandwichs(int type);

/** affiche pour chaque type (entier) de sandwich le nombre d'unités commandées */
void recapitulatifQuotidien();

/** remet à zéro le nombre de sandwichs commandés, pour tous les types */
void debutDeJournee();
```

Cette classe possède en outre un constructeur qui prend en argument le type de sandwich (sous forme d'un entier).

Exercice 1. (4,5 points) Écrire la classe **`Sandwich`** (attributs, constructeurs et méthodes). Penser à spécifier ce qui est statique.

Optimiser les commandes

Dans cette partie, on modélise la liste des commandes sous forme d'une liste doublement chaînée : la liste est représentée par la classe **`APreparer`**, les cellules sont représentées par la classe **`UneCommande`**.

Lorsqu'un nouveau sandwich est commandé, une cellule de valeur ce sandwich (donc de type **`Sandwich`**) est ajoutée en fin de liste. Dès qu'un préparateur est libre, il récupère et prépare le sandwich en tête de liste.

L'interface de la classe **`APreparer`** contient au moins les méthodes suivantes :

```
/** renvoie le nombre de sandwichs en attente de préparation (ie la longueur de
la liste) */
int nbSandwichs();

/** renvoie le nombre de sandwichs en attente de préparation du type donné en
argument */
int nbSandwichs(int type);
```

```
/** renvoie le type du prochain sandwich à préparer (tête de liste) et le
supprime de la liste */
int prochainAPreparer();

/** ajoute en fin de liste un nouveau sandwich dont le type est donné en
paramètre */
void nouvelleCommande(int type);
```

Exercice 2. (3 points) Donner les attributs et les constructeurs des classes **`APreparer`** et **`UneCommande`**. Il s'agit de décider quels constructeurs écrire pour que ces deux classes soient utilisables et qu'il soit possible d'écrire toutes les méthodes données dans le cadre ci-dessus pour **`APreparer`**.

Exercice 3. (2,5 points) Écrire les deux méthodes **`nbSandwichs`**. A-t-on le droit de leur donner le même nom ? (Justifier.)

Exercice 4. (2 points) Écrire la méthode **`prochainAPreparer`**.

Exercice 5. (2 points) Écrire la méthode **`nouvelleCommande`**.

Optimiser les préparations

Le gérant du "gland de chêne" ayant remarqué que préparer plusieurs sandwichs d'un même type en même temps est plus rapide que les préparer les uns après les autres, nous allons adapter notre modélisation à cette remarque, en ajoutant dans l'interface de la classe **`APreparer`** les méthodes suivantes :

```
/** renvoie un tableau d'entiers, la valeur de la case i correspondant au nombre
de sandwichs de type i à préparer, puis vide la liste */
int[] enAttente();

/** met à jour le tableau des sandwichs à préparer: pour chaque type de sandwich
on ajoute à ce qui est dans la case du tableau le nombre de sandwichs apparaissant
dans la liste chaînée; puis vide la liste */
void miseAJourAttente(int[] att);

/** met à jour le tableau des sandwichs à préparer en tenant compte de la largeur
du plan de travail: pour chaque type de sandwich on ajoute à ce qui est dans la
case du tableau le nombre de sandwichs apparaissant dans la liste chaînée sans
dépasser max; et on supprime de la liste les sandwichs correspondant */
void miseAJourAttenteMax(int[] att, int max);
```

Exercice 6. (3 points) Implémenter les méthodes **`enAttente`** et **`miseAJourAttente`**, en respectant les contraintes suivantes :

- chacune de ces méthodes ne parcourt qu'une fois la liste chaînée,
- une de ces méthodes fait un appel à l'autre (au choix).

Exercice 7. (3 points) Le plan de travail pour préparer les sandwichs n'étant pas extensible à l'infini, on se propose de limiter les reports à un nombre maximal de sandwichs : écrivez la méthode **`miseAJourAttenteMax`** qui met à jour le nombre de sandwichs de chaque type à préparer dans le tableau **`att`**, sans pouvoir dépasser la valeur **`max`**, et enlève les sandwichs ainsi comptabilisés de la liste des sanwichs (en enlevant en priorité ceux qui apparaissent d'abord dans la liste).