Exercice 1:
On souhaite définir différents types d'articles dans un magasin. Tous les articles partagent des caractéristiques communes, une référence, un libellé, un fournisseur et un prix d'achat. Des promotions pourront être appliquées à l'ensemble des articles. On enregistrera donc aussi pour chaque article un prix de l'article et un prix de vente pouvant être inférieur au prix de l'article.

1. Créer une classe `Article` caractérisée par une référence de type `int`, un libellé, un fournisseur de type `String`, un `prixAchat`, un `prixArticle` et un `prixVente` de type `double`. Ces attributs auront une visibilité **privée**.

   Cette classe fournira un constructeur ayant pour paramètres la référence, le libellé, le fournisseur, le prix d'achat et le prix de l'article. Le prix de vente sera initialisé au prix de l'article. Elle fournira également les méthodes suivantes :

   a. `public void promotion (int pourcentage)` qui affecte au prix de vente le prix de l'article auquel on applique une réduction du pourcentage indiqué en paramètre.
   b. `public void remonterPrix ()` qui affecte au prix de vente la valeur du prix de l'article.
   c. `public String toString()` qui retourne l'état d'un article sous forme d'une chaîne de caractères décrivant toutes les caractéristiques de celui-ci.

On distingue trois types d'articles, les articles électromenagers, les articles d'habillement et les articles primeurs.

2. Créer une classe `ArticleElectromenager` héritant de la classe `Article` ayant pour caractéristique supplémentaire le poids : `private int poids`.

   Cette classe fournira un constructeur ayant pour paramètres la référence, le libellé, le fournisseur, le prix d'achat, le prix de l'article et le poids.
   Elle redéfinira la méthode `public String toString()` qui retourne l'état d'un article tel que défini dans la classe `Article` suivi, sur une autre ligne, du poids de cet article.

3. Créer une classe `ArticleHabillement` héritant de la classe `Article` ayant pour caractéristique supplémentaire la taille : `private int taille`.

   Cette classe fournira un constructeur ayant pour paramètres la référence, le libellé, le fournisseur, le prix d'achat, le prix de l'article et la taille.
   Elle redéfinira la méthode `public String toString()` qui retourne l'état d'un article tel que défini dans la classe `Article` suivi, sur une autre ligne, de la taille de cet article.

4. Créer une classe `ArticlePrimeur` héritant de la classe `Article` ayant pour caractéristique supplémentaire la date de péremption : `private LocalDate datePeremption`.
   (voir la documentation de `java.time.LocalDate` en annexe1).

   Cette classe fournira un constructeur ayant pour paramètres la référence, le libellé, le fournisseur, le prix d'achat, le prix de l'article et la date de péremption.
   Elle redéfinira la méthode `public String toString()` qui retourne l'état d'un article tel que défini dans la classe `Article` suivi, sur une autre ligne, de la date de péremption de cet article.

   Elle fournira la méthode `public boolean estPerime()` qui retourne `true` si l'article est périmé et `false` sinon.

5. On souhaite gérer les articles d'un magasin. Écrire une classe `Magasin` caractérisée par
    - un attribut `lesArticles` de visibilité privée et de type un tableau d'objets de type `Article`.
    - un attribut `capacite` fixant la capacité du tableau à 1000 ;
    - un attribut `nb` correspondant au nombre d'articles dans le magasin qui sera initialisé à 0.

   La classe `Magasin` fournira un constructeur sans paramètre ainsi que les méthodes suivantes:
    - `public boolean estPresent (Article a)` qui retourne vrai si l'article est présent dans le magasin et faux sinon. Que faut-il redéfinir dans la classe `Article` ?
    - `public boolean ajouterArticle(Article a)` qui ajoute l'article passé en paramètre à la fin du tableau et retourne `true` à condition que l'article ne soit pas déjà présent dans le magasin et que la capacité le permette. Sinon, la méthode retourne `false`. On pensera à incrémenter le nombre d'articles du magasin.
    - `public boolean promotion(Article a, int pourcentage)` qui applique à l'article passé en paramètre la promotion correspondant au pourcentage passé en paramètre et retourne vrai. La méthode retourne faux si l'article n'est pas présent dans le magasin.
    - `public String toString()` qui retourne l'état de chacun des articles du magasin. Les informations de chaque article apparaissant sur une ligne différente.

6. Créer une classe `Main` contenant le `main (String[] args)`. Créer un article de chaque type en définissant des variables du type général `Article`.
   Exemple : **`Article`** `a = new` **`ArticlePrimeur`**`(1, "Salade", "f", 0.4, 0.9, LocalDate.of (2019,10,1))`.
7. Créer un magasin et y ajouter les articles.
   Afficher l'état des articles contenus dans le magasin.
   Appliquer une promotion à certains articles puis afficher de nouveau l'état des articles contenus dans le magasin.
   Tester si l'article `a` est périmé.Est-ce possible ? Que faut-il faire ?

Exercice 2. On souhaite manipuler des listes de n-uplets de manière à pouvoir trier les n-uplets contenus dans la liste.

Créer une classe `ListeNuplets` caractérisée par un attribut `lesNuplets` de type un tableau de `Nuplet`.

On devra donc créer une classe `Nuplet` pour manipuler les n-uplets du tableau.

Cette classe pourra être interne à la classe `ListeNuplets`. On la déclarera statique et dans un premier temps, on lui attribuera une visibilité privée.

1) Construction de la classe `Nuplet` **:**

1. On définit un n-uplet $(u_1, u_2, \ldots, u_k)$ de taille quelconque `k` dans la classe `Nuplet` par un attribut privé `private int[] content` contenant les valeurs des u_i.

2. Définir un constructeur `public Nuplet (int k)` qui construit un n-uplet de taille k. Les valeurs u_i doivent être saisies par lecture interactive et doivent être positives. Gérer la possibilité de saisir des valeurs non entières ou négatives.
3. Redéfinir la méthode `public String toString()` qui retourne l'état de l'objet sous la forme : "$(u_1, u_2, ..., u_k)$".
4. Définir deux autres méthodes :
    - `public int nbElements()` qui retourne le nombre d'éléments du n-uplet.
    - `public int getElement(int index)` qui retourne l'élément du n-uplet correspondant à l'indice passé en paramètre si l'indice est valide et -1 sinon.

2) Construction de la classe `ListeNuplets` :

1. Définir le constructeur `public ListeNuplets(int...lesTailles)` permettant de passer en paramètre une suite variable de valeurs entières correspondant à la taille de chacun des n-uplets de la liste que l'on veut créer.

   Exemple : `ListeNuplets l = new ListeNuplets(3,3,4,2);`

   permet de créer une liste de 4 n-uplets, le premier et le second de taille 3, le troisième de taille 4 et le dernier de taille 2.
2. Redéfinir la méthode `toString()`.
3. Définir les méthodes suivantes :
    - `public Nuplet getNuplet (int index)` qui retourne le `Nuplet` présent dans la liste à l'indice indiqué en paramètre si l'indice est valide et `null` sinon.
    - `public void trier()` qui trie la liste des n-uplets.

      Quelle classe de l'API Java devons-nous utiliser pour réaliser cela ?

      Comment doit-on modifier la classe `Nuplet` pour que cela fonctionne ? Implémentez-le sachant que l'ordre de comparaison de deux n-uplets se base sur la définition suivante[1] :

      Soient a = $(a_1, a_2, ..., a_i)$ et b = $(b_1, b_2, ..., b_j)$ deux n-uplets.

      a < b si et seulement si

      i. $i < j$ et $\forall k \leqslant i, a_k = b_k$ ou

      ii. jusqu'à l'indice $k < i$ et $k < j, a_k = b_k$ et $a_{k+1} < b_{k+1}$.

      Exemples :

1 Vous remarquerez que cet ordre est l'ordre lexicographique utilisé pour comparer deux chaînes de caractères.

(4, 3) < (4, 3, 5, 7) par i.

(4, 3, 2) < (4, 3, 5, 7) par ii.

(4, 1, 2) < (4, 3) par ii.

3) Créer une classe `TestTriNuplets` contenant la méthode `main` et permettant de créer une liste de n-uplets puis de la trier. Afficher l'état de la liste avant et après le tri.

Récupérer le `Nuplet` de la liste correspondant à l'indice de votre choix et afficher son nombre d'éléments et tous ses éléments. La visibilité privée de la classe `Nuplet` permet-elle de le faire ? Modifier cette visibilité si nécessaire.

Exercice 3 Nous allons reprendre l'exercice 1 et réécrire plusieurs fois la classe `Magasin` en utilisant différentes interfaces de la bibliothèque des collections Java consultables en Annexe.

Dans les trois cas décrits ci-dessous, la classe `Magasin` fournira un constructeur sans paramètre et les méthodes suivantes :

- `public boolean ajouterArticle(Article a)` qui ajoute l'article passé en paramètre au magasin.
- `public Article rechercherArticle(int reference)` qui retourne l'article correspondant à la référence passée en paramètre si elle existe dans le magasin et `null` sinon.
- `public boolean supprimerArticle (int ref )` qui supprime l'article correspondant à la référence passée en paramètre si elle existe dans le magasin et retourne `true` sinon retourne `false`.
- `public boolean promotion(int ref, int pourcentage)` qui applique la promotion à l'article correspondant à la référence passée en paramètre si elle existe dans le magasin et retourne `true` sinon retourne `false`.
- `public void liquidationTotale(int pourcentage)` qui applique une promotion à tous les articles du magasin selon le pourcentage passé en paramètre.
- `public String toString()` qui retourne les informations des articles présents dans le magasin sous forme d'une chaîne de caractères.

  Vous utiliserez les différentes méthodes permettant d'itérer sur une collection :

  - la méthode `Iterator<T> iterator()` de l'interface `Iterable<T>` et sa version `for(…:…)`
  - la méthode `default void forEach (Consumer< ? super T> action)` de l'interface `Iterable<T>`.

1. Créer une première classe `Magasin` caractérisée par un attribut `lesArticles`. Choisir le type de l'attribut parmi les interfaces de la bibliothèque des collections sachant qu'on souhaite gérer la collection des articles comme un ensemble, sans doublons. Quelles méthodes doit-on redéfinir dans la classe `Article` ?

Ajouter à cette classe `Magasin` la méthode ci-dessous et la compléter pour que les articles de la liste retournée soient triés par ordre croissant de leur prix de vente.

```
public List<Article> triParOrdreCroissant(){
   List<Article> liste= new ArrayList<>(lesArticles);
   ...
   return liste ;
}
```

2. Créer une deuxième classe `Magasin` caractérisée par un attribut `lesArticles`. Choisir le type de l'attribut parmi les interfaces de la bibliothèque des collections sachant qu'on souhaite gérer la collection des articles comme un ensemble, sans doublons dont les éléments sont munis d'une relation d'ordre.
   Que doit-on ajouter à la classe `Article` ? Implémentez-le sachant qu'on souhaite comparer les articles par ordre croissant de leur référence.

   Ajouter à cette classe `Magasin` les méthodes :

   - `public Article lePremierArticle()` qui retourne l'article ayant la plus petite référence.
   - `public Article leDernierArticle()` qui retourne l'article ayant la plus grande référence.

3. Quelle modification doit-on apporter à la classe `Magasin` ci-dessus si on souhaite modifier la relation d'ordre sans modifier l'ordre naturel défini dans la classe `Article`. On souhaite maintenant comparer les articles par ordre croissant de leur libellé.

4. Créer une troisième classe `Magasin` caractérisée par un attribut `lesArticles`. Choisir le type de l'attribut parmi les interfaces de la bibliothèque des collections sachant qu'on souhaite enregistrer des paires d'informations de type (référence, Article). Ajouter un article `a` de référence `r` consistera à ajouter une paire `(r, a)`.
   Pour cette classe, la méthode `toString` affichera les paires sur des lignes différentes au format : Référence : r – informations sur l'article.

Exercice 4. On souhaite réaliser une application de gestion d'un annuaire téléphonique permettant d'associer à une personne un ou plusieurs numéros de téléphone.

On va donc travailler avec deux classes `Personne` et `NumTel` qui seront fournies en TP et dont la documentation est donnée en annexe.

• **`Personne`**: classe représentant une personne définie par un nom, un prénom et une civilité (M, Mlle, Mme).

• **`NumTel`** : classe représentant un numéro de téléphone, défini par le numéro proprement dit et un code indiquant la nature du numéro (numéro de portable, numéro de poste fixe professionnel, numéro de poste fixe à domicile, numéro de Fax).

1. Créer une classe `ListeNumTel` permettant de gérer une liste de numéros de téléphone.

   Proposer un constructeur sachant qu'une liste doit toujours contenir au moins un numéro de téléphone ainsi que les méthodes suivantes:

   - `boolean ajouter(int index, NumTel num)`

   Ajoute un numéro à une position donnée dans la liste, sans effet si le numéro est déjà présent dans la liste.

   - `boolean ajouterFin(NumTel num)`

   Ajoute un numéro à la fin de la liste, sans effet si le numéro est déjà présent dans la liste.

   - `boolean contientNumero(int num)`

   Teste la présence d'un numéro dans la liste.

   - `java.util.Iterator<NumTel> iterator()`

   Renvoie un itérateur sur les numéros de téléphone contenus dans la liste.

   - `int nbNumeros()`

   Retourne le nombre de numéros de la liste (>=1).

   - `NumTel numero(int index)`

   Retourne le $i^{ème}$ numéro de la liste.

   - `NumTel premierNumero()`

   Retourne le premier numéro de la liste (il existe forcément).

   - `boolean retirer(int num)`

   Enlève un numéro de la liste. Si le numéro existe, retourne `true` sinon `false`. Cette opération n'est possible que si la liste contient au moins deux numéros (`nbNumero()>1`).

   - `String toString()`

   Retourne la séquence des numéros contenu dans cette liste.

2. Créer une classe `Annuaire` permettant d'associer à une personne une liste de numéros de téléphone. Chaque personne ne doit se trouver qu'une seule fois dans l'annuaire.

   Cette classe proposera un constructeur sans paramètre et les méthodes suivantes :

   - `boolean ajouterEntree(Personne p, ListeNumTel nums)` : ajoute une nouvelle entrée dans l'annuaire.

   Si p n'existe pas, on crée une nouvelle association (p,nums) et le booléen `true` est retourné; sinon le booléen `false` est retourné et la méthode est sans effet.

   - `ListeNumTel numeros(Personne p)` : retourne la liste des numéros d'une personne. Si la personne est absente retourne `null`.

   Quelle méthode doit-on ajouter à la classe Personne pour effectuer cette recherche ? Proposer une implantation.

   - `java.util.Iterator <Personne> personne()` : retourne un itérateur sur l'ensemble des personnes contenues dans l'annuaire.
   - `void ajouterNumeroFin(Personne p, NumTel n)` : ajoute un numéro à la fin de la liste des numéros d'une personne.

Si la personne n'existe pas, on crée une nouvelle entrée pour cette personne avec comme liste de numéros associée la liste constituée du numéro passé en paramètre.

- `public NumTel premierNumero (Personne p)` : retourne le premier numéro d'une personne si cette personne est présente dans l'annuaire, sinon retourne `null`.
- `public void supprimer (Personne p)` : supprime une personne dans l'annuaire si celle-ci est présente.
- `public void supprimerNumero (int n, Personne p)` : supprime le numéro pour une personne donnée. Si le numéro est le seul numéro de la personne, retire la personne.
- `public Set<Personne> lesEntreesPour (String nom)` : retourne l'ensemble des personnes ayant le nom passé en paramètre.
- `public String toString ()` : retourne une chaîne de caractères décrivant l'ensemble des personnes de l'annuaire. Chaque ligne contient les informations d'une seule personne.

3. On souhaiterait consulter les personnes de l'annuaire dans l'ordre alphabétique.

   Quelles modifications devrait-on apporter et dans quelles classes?

4. Proposer une classe `TestAnnuaire` avec une méthode `main` permettant d'insérer des personnes dans un annuaire, chacune avec une liste de numéros de téléphone associée.

ANNEXE : Quelques interfaces de la bibliothèque des collections

Interface `List<E>`

Modifier and Type	Method and Description
boolean	**add**(E e) Appends the specified element to the end of this list (optional operation).
void	**add**(int index, E element) Inserts the specified element at the specified position in this list (optional operation).
boolean	**contains**(Object o) Returns true if this list contains the specified element.
E	**get**(int index) Returns the element at the specified position in this list.
int	**indexOf**(Object o) Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.
boolean	**isEmpty**() Returns true if this list contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this list in proper sequence.
int	**lastIndexOf**(Object o) Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.

ListIterator<E>	listIterator() Returns a list iterator over the elements in this list (in proper sequence).
E	remove(int index) Removes the element at the specified position in this list (optional operation).
boolean	remove(Object o) Removes the first occurrence of the specified element from this list, if it is present (optional operation).
E	set(int index, E element) Replaces the element at the specified position in this list with the specified element (optional operation).
int	size() Returns the number of elements in this list.
default void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified Comparator.

Interface Set<E>

Modifier and Type	Method and Description
boolean	add(E e) Adds the specified element to this set if it is not already present (optional operation).
boolean	contains(Object o) Returns true if this set contains the specified element.
boolean	isEmpty() Returns true if this set contains no elements.
Iterator<E>	iterator() Returns an iterator over the elements in this set.
boolean	remove(Object o) Removes the specified element from this set if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes from this set all of its elements that are contained in the specified collection (optional operation).
boolean	retainAll(Collection<?> c) Retains only the elements in this set that are contained in the specified collection (optional operation).
int	size() Returns the number of elements in this set (its cardinality).

Interface SortedSet<E>

Modifier and Type	Method and Description
Comparator<? super E>	comparator() Returns the comparator used to order the elements in this set, or null if this set uses the **natural ordering** of its elements.
E	first() Returns the first (lowest) element currently in this set.

SortedSet<E>	headSet(E toElement) Returns a view of the portion of this set whose elements are strictly less than toElement.
E	last() Returns the last (highest) element currently in this set.
SortedSet<E>	subSet(E fromElement, E toElement) Returns a view of the portion of this set whose elements range from fromElement, inclusive, to toElement, exclusive.
SortedSet<E>	tailSet(E fromElement) Returns a view of the portion of this set whose elements are greater than or equal to fromElement.

Rappel : Constructeurs de la classe TreeSet<E>

Constructor and Description
TreeSet() Constructs a new, empty tree set, sorted according to the natural ordering of its elements.
TreeSet(Comparator<? super E> comparator) Constructs a new, empty tree set, sorted according to the specified comparator.

Interface Map<K,V>

Modifier and Type	Method and Description
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V>>	entrySet() Returns a Set view of the mappings contained in this map.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
V	put(K key, V value) Associates the specified value with the specified key in this map (optional operation).
default V	putIfAbsent(K key, V value) If the specified key is not already associated with a value (or is mapped to null) associates it with the given value and returns null, else returns the current value.
V	remove(Object key) Removes the mapping for a key from this map if it is present (optional operation).
default boolean	remove(Object key, Object value) Removes the entry for the specified key only if it is currently mapped to the specified value.
Collection<V>	values() Returns a Collection view of the values contained in this map.

int	**size**() Returns the number of key-value mappings in this map.