Des interfaces et classes abstraites vues en cours sont rappelées à la fin de ce TD.

Exercice 1. On considère le code suivant.

```
1  MaClasse a,b;
   ... // code modifiant a et b
```

Dans quelles circonstances les cas (1), (2), (3) et (4) du tableau ci-dessous se produisent ?

		a.equals(b)	
		true	false
a==b	true	(1)	(2)
	false	(3)	(4)

Que peut afficher le code ci-dessous dans les cas (5), (6), (7), (8) ?

```
System.out.print(a == b)          // affiche                    true  ou  false
2 a.methodeModifiantLInstance();
System.out.print(a == b)          // que peut s'afficher ici (5)  ou   (6)
4 System.out.print(a.equals(b))   // que peut s'afficher ici (7)  ou   (8)
```

On considère maintenant les lignes suivantes **à la place** du code précédent. Que peut-il s'afficher dans les cas (9), (10), (11) et (12) ?

```
System.out.print(a.equals(b));  // affiche                    true  ou  false
2 a.methodeModifiantLInstance();
System.out.print(a.equals(b));  // que peut s'afficher ici  (9)  ou   (10)
4 System.out.print(a == b);     // que peut s'afficher ici (11)  ou   (12)
```

Exercice 2. Le but de cet exercice est de créer une classe pour récupérer les n plus grands entiers d'une `Collection<Integer>`. Concevoir une classe `MultiMax` pour cela ; elle doit comporter :

- un constructeur qui prend n en argument ;
- une méthode `processAll` qui prend la collection en argument.
- une méthode pour récupérer les n plus grands entiers (itérateur ou autre).

Il est conseillé d'ajouter une fonction `process` qui traite un seul entier.

Exercice 3. Qui n'a jamais voulu renvoyer deux objets différents avec la même fonction ? Implémenter une classe (doublement) générique `Paire<X,Y>` qui a deux attributs publiques `premier` et `deuxieme` et un constructeur.

Exercice 4. Quand une fonction peut renvoyer quelque chose ou rien, il n'est pas conseillé de renvoyer `null` dans ce deuxième cas. Concevoir une classe générique `Optionnel<T>` qui a soit une valeur de type `T` soit n'en a pas. Lancer une exception si l'utilisateur demande une valeur alors que l'objet n'en a pas. (Dans java 8, la classe `java.util.Optional<T>` existe).

Exercice 5. Une file est une liste *fifo* (*first in first out* ou, en français, premier entré premier sorti) qui fonctionne donc comme une file d'attente. Elle doit avoir une méthode pour enfiler un élément (`add`) et une méthode pour "défiler" un élément (`defile`). Tout nouvel élément doit être ajouté *à la fin* ; utiliser une liste chaînée classique est donc inefficace.
Implémenter une classe générique `File` qui utilise (deux instances de) la classe générique `Pile` vue en cours, comme expliqué ci-dessous.

— La première pile modèle la fin de la file ; quand on enfile (`add`) un nouvel élément, il est empilé dans cette pile.
— La deuxième pile modèle le début de la file ; quand on défile un élément (`defile`), on le dépile de cette pile. Si l'on doit retirer un élément de cette seconde pile et qu'elle est vide, on dépile entièrement la première pile dans la seconde (ce qui inverse l'ordre des éléments).

Proposer une autre façon d'implémenter "efficacement" une file.

Exercice 6. Refaire l'exercice 2 de façon générique : implémenter une classe `Multimax` qui permet de trouver les n plus grandes valeurs d'une `Collection<E extends Comparable<E>>`. (Attention aux tableaux [1], utiliser des `Vector` à la place.)

Exercice 7. Implémenter une classe générique `ListeTriee` qui représente une liste doublement chaînée dont les éléments sont classés par ordre croissant. On utilisera (correctement et si c'est nécessaire) `Comparable`, `AbstractList`, `Iterable`, `Collection` et `Iterator`. La liste doit-être une `Collection` pleinement fonctionnelle (ajout, suppression, test d'appartenance).

```
public interface Comparable<T> {
  public int compareTo(T obj);
    // Compares this avec obj, renvoie entier
    //   - negatif  si this < T
    //   - nul      si this et T sont ``equals''
    //   - positif  si this > T
}
```

```
public interface Iterable<T> {
    public Iterator<E> iterator();
}
```

```
public interface Iterator<T> {
    public T next(); // Renvoie le prochain element (et passe au suivant).
    public boolean hasNext(); // Renvoie true s'il y a un prochain element.
    public void remove(); // Supprime le dernier element renvoye par next().
}
```

```
public abstract class AbstractCollection<E> implements Collection<E> {
    boolean add(E e); // renvoie une exception
    boolean addAll(Collection<? extends E> c); // utilise add de this
    void clear(); // utilise remove de l'iterateur
    boolean contains(Object o);
    boolean containsAll(Collection<?> c);
    boolean isEmpty(); // utilise size de this
    abstract Iterator<E> iterator();
    boolean remove(Object o);           // utilise remove de l'iterateur
    boolean removeAll(Collection<?> c); // utilise remove de l'iterateur
    boolean retainAll(Collection<?> c); // utilise remove de l'iterateur
    abstract int size();
    Object[] toArray();     // ne pas s'occuper de ces methodes
    <T> T[] toArray(T[] a); // ne pas s'occuper de ces methodes
    String  toString();
}
```

1. Il est interdit d'instancier le type `E[]` (sans contorsions excessives) si `E` est un type générique : on ne peut pas utiliser l'instruction `new E[n]`.