ensiie
école nationale supérieure d'informatique pour l'industrie et l'entreprise

TP ILO n° 2

Prérequis

- Chaque exercice se présente sous la forme de classes Java à compléter dont vous pourrez trouver un squelette d'implémentation dans /pub/ILO/TP2.zip.
  - Copiez cette archive chez vous et dézippez la dans un sous répertoire ILO par exemple
  - Lancez votre IDE préféré et importez le projet « TP Ensembles » que vous venez de dézipper. Vous pourrez constater qu'il subsiste des erreurs de compilation qui seront résolues lorsque vous copierez vos classes IListe<E> et Liste<E> dans ce projet.
- Documentation Java :
  - http://docs.oracle.com/javase/8/docs/api/

1. Ensembles

On souhaite réaliser plusieurs implémentations des ensembles. Un ensemble est défini comme une collection non ordonnée d'éléments sans doublons. Les différentes opérations sur un ensemble sont les suivantes :

- Ajout d'un élément : $\{ a \quad b \quad c \} + b = \{ a \quad b \quad c \}$
  $\{ a \quad b \quad d \} + c = \{ a \quad b \quad d \quad c \}$
- Retrait d'un élément : $\{ a \quad b \quad c \} - c = \{ a \quad b \}$
- Ensemble vide : $\{ a \quad b \quad c \} \equiv \varnothing = faux$
- Appartenance d'un élément à un ensemble : $a \in \{ a \quad b \quad c \} = vrai$
- Inclusion d'un ensemble dans un autre : $A \subset B = \{ a \quad b \} \subset \{ a \quad b \quad c \} = vrai$
  $\varnothing \subset B = \varnothing \subset \{ a \quad b \quad c \} = vrai$, et $B \subset \varnothing = faux$
- Cardinal : $card(\{ a \quad b \quad c \}) = 3$
- Complément : $A - B = \{ a \quad b \quad c \} - \{ c \quad d \quad e \} = \{ a \quad b \}$
  $B - A = \{ c \quad d \quad e \} - \{ a \quad b \quad c \} = \{ d \quad e \}$
  et $A - B \neq B - A$
- Union : $A \cup B = \{ a \quad b \quad c \} \cup \{ c \quad d \quad e \} = \{ a \quad b \quad c \quad d \quad e \}$ et $A \cup B \equiv B \cup A$
- Intersection : $A \cap B = \{ a \quad b \quad c \} \cap \{ c \quad d \quad e \} = \{ c \}$ et $A \cap B \equiv B \cap A$
- Différence symétrique : $A \Delta B = \{ a \quad b \quad c \} \Delta \{ c \quad d \quad e \} = \{ a \quad b \quad d \quad e \}$
  $A \Delta B \equiv B \Delta A = (A - B) \cup (B - A) = (A \cup B) - (B \cap A)$
- Égalité de deux ensembles : $A == B = (A \subset B) \,\&\, (B \subset A)$

Ces ensembles peuvent être implémentés de différentes manières, avec des « `Vector` » (vu au TD n°1), ou bien avec des `Tableau<E>` (voir Figure 1).

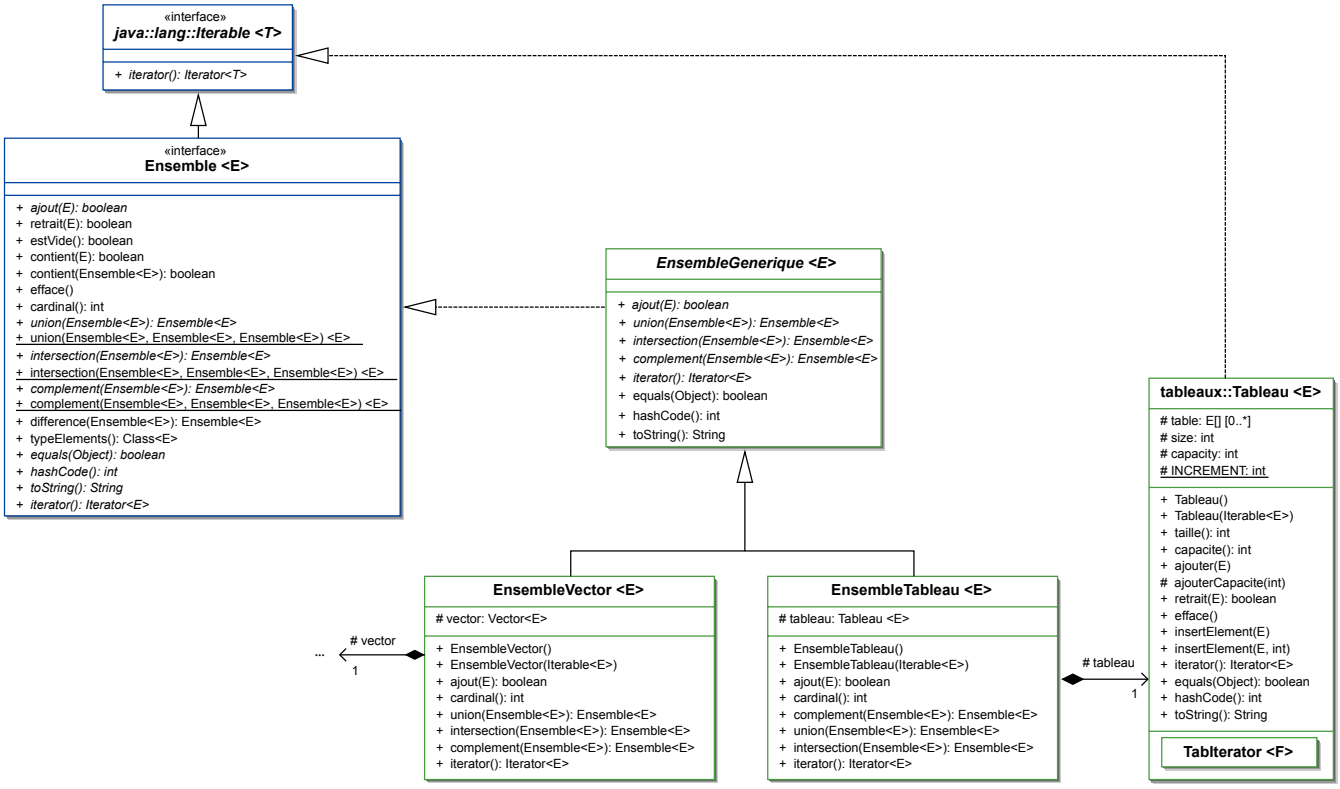


Figure 1: Implémentation des ensembles

L'interface « `Ensemble<E>` » définit les opérations à réaliser sur tous les ensembles.

On veut réaliser deux implémentations des ensembles, une première utilisant comme conteneur sous-jacent un `Tableau<E>` et une seconde utilisant un `Vector<E>`. Le Tableau comme le Vector étant toutes les deux des classes itérables (implémentant l'interface `Iterable<E>`) une grande partie des opérations sur les ensembles peuvent être réalisées grâce à un iterator, on peut donc factoriser une grande partie de ces opérations dans l'interface « `Ensemble<E>` » en utilisant les opérations fournies par l'iterator.

1. Étudiez l'interface `Ensemble<E>` pour vous familiariser avec ses opérations.
2. Complétez les implémentations partielles en utilisant les indications contenues dans les commentaires de documentation de :
    a. L'interface `Ensemble<E>` (en utilisant des méthodes par défaut et des méthodes de classes).
    b. La classe abstraite `EnsembleGenerique<E>` implémentant (partiellement) l'interface `Ensemble<E>`.
3. Étudiez la classe `Tableau<E>` pour vous familiariser avec ses opérations.
4. Complétez la première implémentation concrète des ensembles (`EnsembleTableau<E>`) héritant de la classe abstraite `EnsembleGenerique<E>` et utilisant comme conteneur sous-jacent pour stocker les éléments de l'ensemble une instance de la classe Tableau. Vous remarquerez que comme la plupart des opérations ont déjà été implémentées dans les classes/interfaces mères, il ne reste plus à implémenter que les méthodes qui allouent ou qui ajoutent des éléments.
5. Testez vos ensembles avec la classe de test AllEnsembleTest.java du package tests : `Menu contextuel → Run As → JUnit Test`.
   La classe `AllEnsembleTest` teste les fonctionnalités des `Ensemble<E>` :

Pour faciliter la progression des tests, vous pourrez commenter / décommenter les lignes suivantes de la classe `AllEnsembleTest.java (ligne 73)` au fur et à mesure de votre progression dans l'implémentation des classes, le dernier élément non commenté ne doit pas contenir de virgule :

```
/**
 * Les différentes natures d'ensembles à tester
 */
@SuppressWarnings("unchecked")
private static final Class<? extends Ensemble<String>>[] typesEnsemble =
(Class<? extends Ensemble<String>>[]) new Class<?>[]
{
        EnsembleTableau.class,
        EnsembleVector.class // à commenter / décommenter
        …
};
```

6. Complétez de la même manière la seconde implémentation concrète des ensembles (`EnsembleVector<E>`) utilisant comme conteneur sous-jacent une instance de la classe `Vector`.

2. Ensembles triés.

On veut maintenant réaliser de la même manière que précédemment deux implémentations des ensembles triés dans lesquels les éléments doivent être maintenus dans un ordre croissant au sens de l'interface `Comparable<E>` (voir la doc). On veut pour ce faire réutiliser autant que possible ce que l'on a déjà réalisé sur les ensembles.
Néanmoins, certaines méthodes devront être spécialisées pour respecter les spécificités des ensembles triés :

- L'opération d'ajout devra insérer un nouvel élément de manière triée dans l'ensemble $\{ a \quad b \quad d \} + c = \{ a \quad b \quad c \quad d \}$. On rajoutera donc une opération de calcul de rang d'un élément dans l'interface « `EnsembleTri<E>` » définissant les opérations spécifiques aux ensembles triés.
- Les méthodes `equals(Object o)` et `hashCode()` devront être réimplémentées pour les ensembles triés :
    - Le test d'égalité (`equals(Object o)`) devra cette fois prendre en compte l'ordre des éléments ce qui correspond au equals d'une `Collection<E>` tel que vu en cours.
    - Le calcul du `hashCode` devra lui aussi prendre en compte l'ordre des éléments (on pourra donc s'inspirer du calcul du `hashCode` de la classe Tableau).

Il existe pour implémenter les ensembles triés deux méthodes (voir Figure 2, page 4) :

- La première méthode consiste à faire directement des classes filles à `EnsembleTableau<E>` et `EnsembleVector<E>` et implémentant l'interface `EnsembleTri<E>` : C'est le cas de la classe `EnsembleTriVector<E>` de la Figure 2.
- La seconde méthode consiste à introduire une classe abstraite `EnsembleTriGenerique<E>` implémentant partiellement l'interface `EnsembleTri<E>` et possédant une instance d'`Ensemble<E>` (qu'on appellera ensemble interne) décorant un Ensemble ordinaire avec les fonctionnalités propres au tri : C'est le cas de la classe `EnsembleTriTableau2<E>`.
  Un tel ensemble trié contient donc une instance d'ensemble ordinaire et l'on s'appuie en grande partie sur ses méthodes pour implémenter les opérations spécifiques aux ensembles triés.
    - Les classes filles de `EnsembleTriGenerique<E>` fourniront les implémentations concrètes des ensembles triés en implémentant les méthodes qui sont restées abstraites dans l'`EnsembleTriGenerique<E>`.

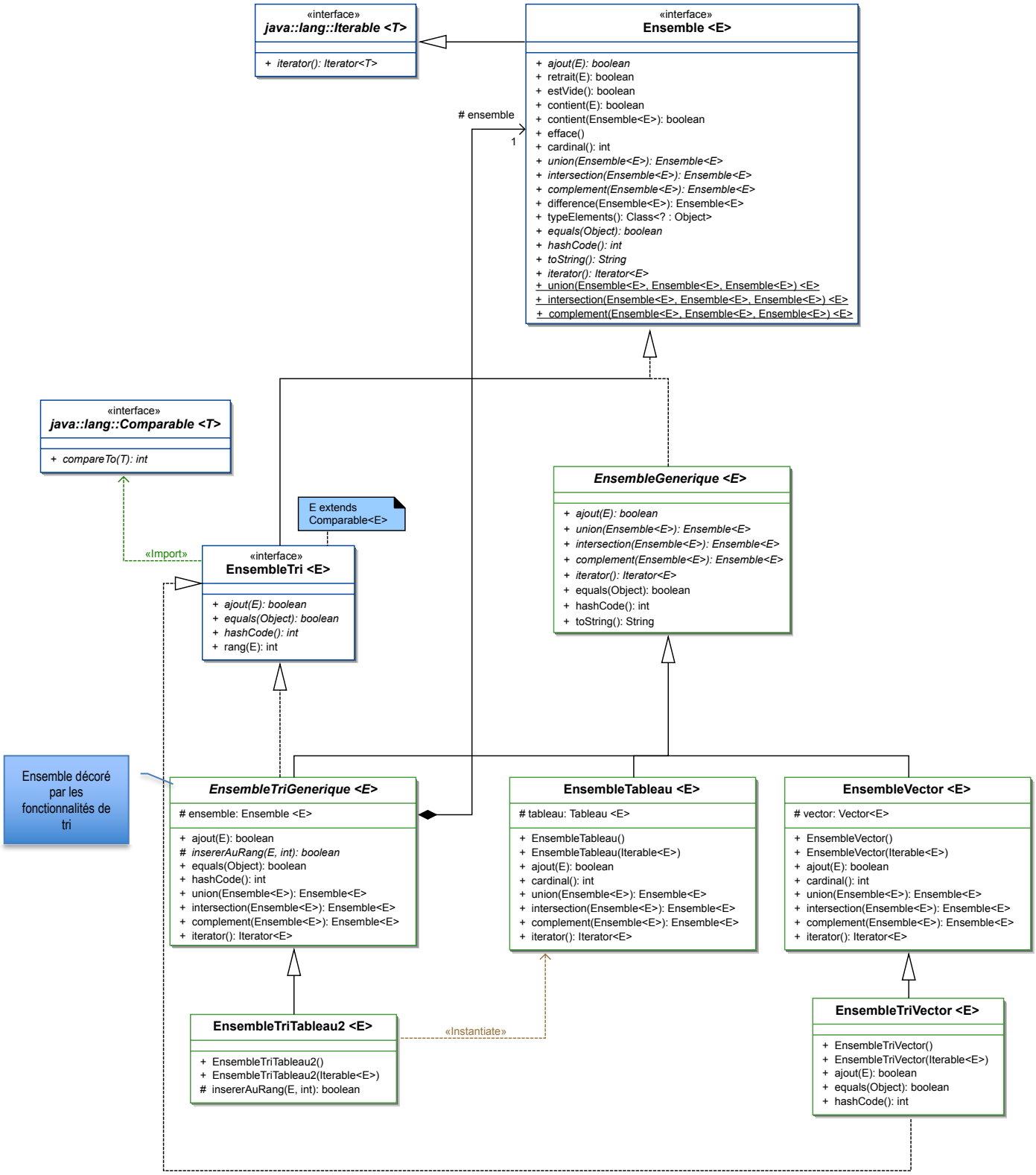


Figure 2 : Implémentation des ensembles triés

1. Créez la classe `EnsembleTriVector<E>` ou la classe `EnsembleTriTableau<E>` avec la première méthode implémentant l'interface `EnsembleTri<E>` et héritant (respectivement) de la classe `EnsembleVector<E>` ou `EnsembleTableau<E>`.
2. Complétez la classe abstraite `EnsembleTriGenerique<E>` implémentant l'interface `EnsembleTri<E>` et héritant de la classe `EnsembleGenerique<E>`. Celle-ci servira de base aux classes `EnsembleTriTableau2<E>` et `EnsembleTriVector2<E>`. Vous pourrez remarquer que

l'implémentation de ces classes est **très** similaire à celles de `EnsembleTriVector<E>` ou `EnsembleTriTableau<E>`.

3. Créez la classe concrète `EnsembleTriTableau2<E>` implémentant la seconde méthode en héritant de la classe `EnsembleTriGenerique<E>`. Cette classe instanciera comme ensemble décoré un `EnsembleTableau<E>`.
4. Créez la classe `EnsembleTriVector2<E>` implémentant la seconde méthode en héritant de la classe `EnsembleTriGenerique<E>`. Cette classe instanciera comme ensemble décoré un `EnsembleVector<E>`.
    a. Si l'on avait choisi qu'une seule méthode sur les deux, laquelle serait la plus avantageuse lorsque l'on veut deux implémentations (avec Tableau et avec Vector) des ensembles triés ?
7. Testez vos ensembles triés avec la classe de test `AllTest.java` qui teste à la fois les ensembles (tous les ensembles triés ou pas : `AllEnsembleTest.java`) et les ensembles triés (`EnsembleTriTest.java`) : Menu contextuel → Run As → JUnit Test.
   Pour faciliter la progression des tests, vous pourrez commenter / décommenter les lignes suivantes de la classe AllEnsembleTest.java (ligne 73) au fur et à mesure de votre progression dans l'implémentation des classes, le dernier élément non commenté ne doit pas contenir de virgule :

```
/**
 * Les différentes natures d'ensembles à tester
 */
@SuppressWarnings("unchecked")
private static final Class<? extends Ensemble<String>>[] typesEnsemble =
(Class<? extends Ensemble<String>>[]) new Class<?>[]
{
        EnsembleVector.class,
        EnsembleTableau.class,
        EnsembleTriVector.class,            // à commenter / décommenter
        EnsembleTriTableau2.class,    // à commenter / décommenter
        EnsembleTriVector2.class            // à commenter / décommenter
};
```

La classe `EnsembleTriTest` teste les fonctionnalités spécifiques aux ensembles triés.

Pour faciliter la progression des tests, vous pourrez commenter / décommenter les lignes suivantes de la classe `EnsembleTriTest.java (ligne 67)` au fur et à mesure de votre progression dans l'implémentation des classes, le dernier élément non commenté ne doit pas contenir de virgule :

```
/**
 * Les différentes natures d'ensembles à tester
 */
@SuppressWarnings("unchecked")
private static final Class<? extends EnsembleTri<String>>[] typesEnsemble =
(Class<? extends EnsembleTri<String>>[]) new Class<?>[]
{
        EnsembleTriVector.class,
        EnsembleTriTableau2.class,    // à commenter / décommenter
        EnsembleTriVector2.class            // à commenter / décommenter
};
```

3. Ensembles utilisant des Listes (à rendre sur le serveur de dépôt `ilo-ensembles` avant lundi 16 avril 2018).

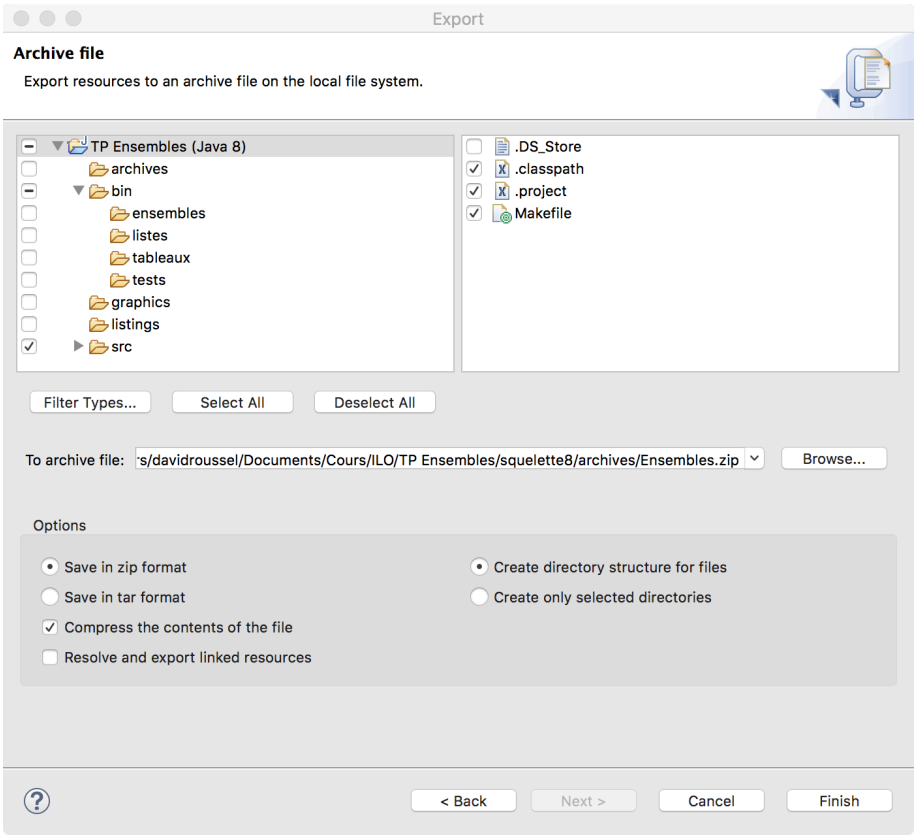
Nous souhaitons maintenant créer une troisième implémentation des ensembles en utilisant cette fois une instance de la classe `Liste<E>` que vous avez créée lors du dernier TD.

1. Copiez vos fichiers IListe.java et Liste.java (complétés) du TP précédent dans le package « listes ».
2. Testez votre classe `Liste<E>` en utilisant la classe de test `ListeTest`.
3. Créez une classe `EnsembleListe<E>` héritant de la classe abstraite `EnsembleGenerique<E>` et utilisant comme conteneur sous-jacent une instance de la classe `Liste<E>`.
4. Testez la classe `EnsembleListe<E>` avec la classe de test `AllEnsembleListeTest`.
5. Créez une classe `EnsembleTriListe<E>` héritant de la classe `EnsembleListe<E>` et implémentant l'interface `EnsembleTri<E>`.
6. Testez la classe EnsembleTriListe<E>.
7. Créez une classe `EnsembleTriListe2<E>` héritant de classe abstraite `EnsembleTriGenerique<E>`.
8. Testez la classe `EnsembleTriListe2`.

Votre rendu devra contenir au moins les classes et interfaces suivantes (uniquement les fichiers java) :

- `IListe<E>`
- `Liste<E>`
- `Ensemble<E>`
- `EnsembleGenerique<E>`
- `EnsembleListe<E>`
- `EnsembleTri<E>`
- `EnsembleTriListe<E>`
- `EnsembleTriGenerique<E>`
- `EnsembleTriListe2<E>`

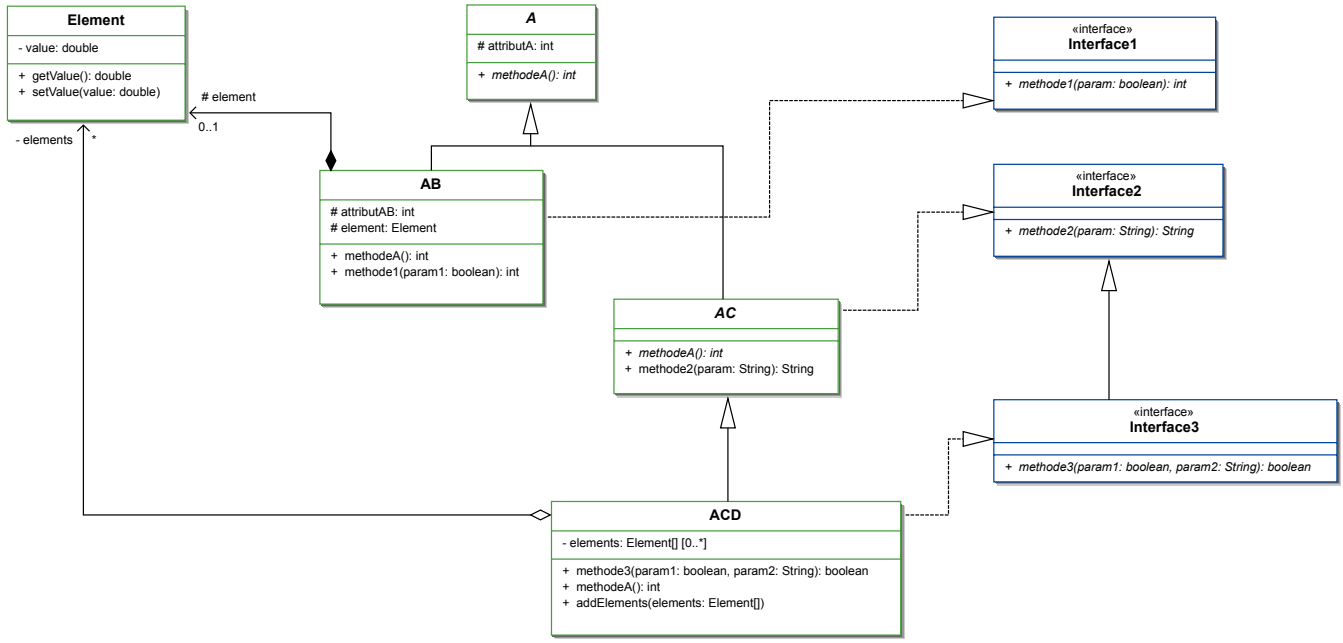
Pour fabriquer l'archive à rendre sur le dépôt, vous pourrez « exporter » votre projet : Clic Droit sur votre projet → export : Archive File → Next → vous pourrez alors sélectionner les fichiers à sauvegarder dans l'archive comme indiqué dans la figure ci-contre.



Rappels UML :

- L'héritage multiple n'existe pas en Java, néanmoins :
  - Une classe peut implémenter plusieurs interfaces; on parle alors d'héritage multiple de comportements.
  - Une interface peut hériter d'une autre interface
- Les membres publics sont précédés de : +
- Les membres protégés sont précédés de : #
- Les membres privés sont précédés de : -
- Les relations d'héritage sont indiquées par les flèches en traits pleins.
- Les relations d'implémentation sont indiquées par les flèches en traits pointillés.
- Les membres ou les classes concrèt(e)s sont indiqués en texte droit.
- Les membres ou les classes abstrait(e)s sont indiqués en texte italique.
- Les compositions sont indiquées par un losange plein : par exemple la classe AB possède une instance de la classe « Element » et elle est responsable de la création et de la destruction de cette instance.
- Les agrégations sont indiquées par un losange vide : par exemple la classe ACD possède une collection d'instances de la classe « Element », mais elle n'est pas responsable de l'instanciation de cette collection (on supposera qu'elle lui est donnée par la méthode addElements).

Exemple :



Questions (à rendre à la fin de la séance à votre chargé de TD)

Nom :	Prénom :	Groupe :

1) Quel était l'intérêt d'implémenter l'algorithmique des opérations ensemblistes (union, intersection, complément) en tant que méthodes de classes dans l'interface `Ensemble<E>` ?

2) Et pourquoi n'a t-on pas fait de même avec l'opération de différence symétrique ?

3) Pourquoi les méthodes `hashCode` des ensembles triés doivent-elles être différentes des méthodes `hashCode` des ensembles non triés ?

4) Parmi les deux manières d'implémenter des ensembles triés, lorsque l'on souhaite avoir au moins deux implémentations différentes, laquelle est la plus économique en termes de développements à réaliser ?
   a) Des classes `EnsembleTriXXX<E>` implémentant l'interface `EnsembleTri<E>` héritant de `EnsembleXXX<E>`.
   b) Des classes `EnsembleTriYYY<E>` héritant d'une classe abstraite `EnsembleTriGenerique<E>` qui implémente partiellement l'interface `EnsembleTri<E>`.

La méthode … car …