

Introduction

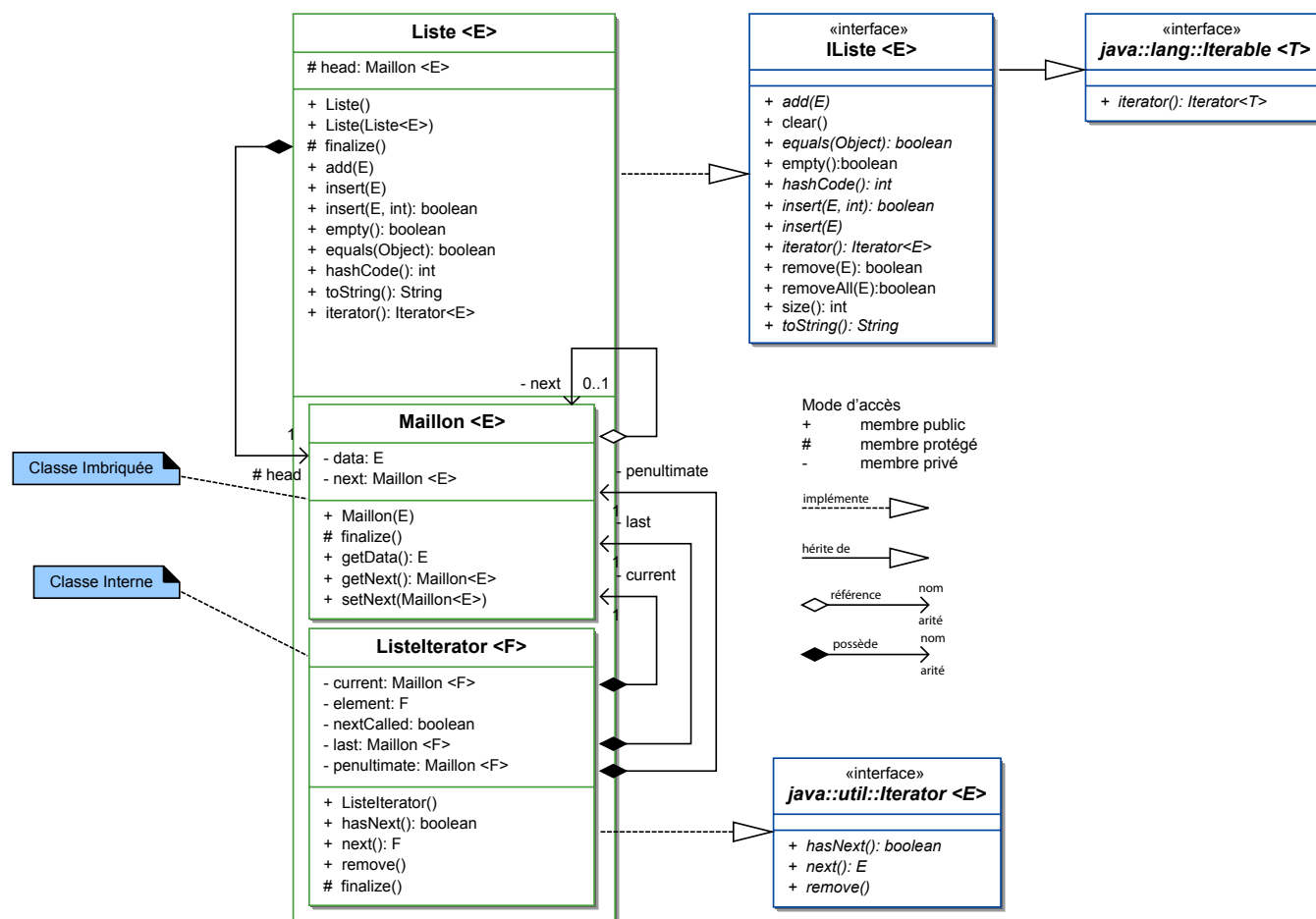
Le but de ce TP est de vous familiariser avec l'environnement de programmation Java, après avoir vu les principes de ce Langage Orienté Objet en cours et en TD. Pour ce faire, nous allons reprendre en partie la fin du TD n° 1 qui traitait des listes chaînées.

Vous pourrez trouver une ébauche du code à réaliser dans l'archive : /pub/ILO/TP1/Listes_Figures.zip. Copiez cette archive sur votre compte et dézippez la dans un sous-répertoire (ILO par exemple), puis après avoir lancé « eclipse » importez le projet : Import ... → Existing Projects into Workspace → Select root directory → Browse et allez chercher le répertoire dans lequel vous avez dézipé le projet.

Travail à réaliser

1) Listes

On cherche à réaliser dans un package « listes » le graphe de classe suivant. Une liste simplement chaînée (Liste<E>) est une liste de maillons (Maillon<E>). Chaque maillon de la chaîne est constitué d'une donnée ainsi que d'une référence vers le maillon suivant (qui peut être « null » s'il n'y a pas de maillon suivant). On utilisera un itérateur (Iterator<E>, ou plus précisément un ListIterator<E>) pour parcourir la liste.



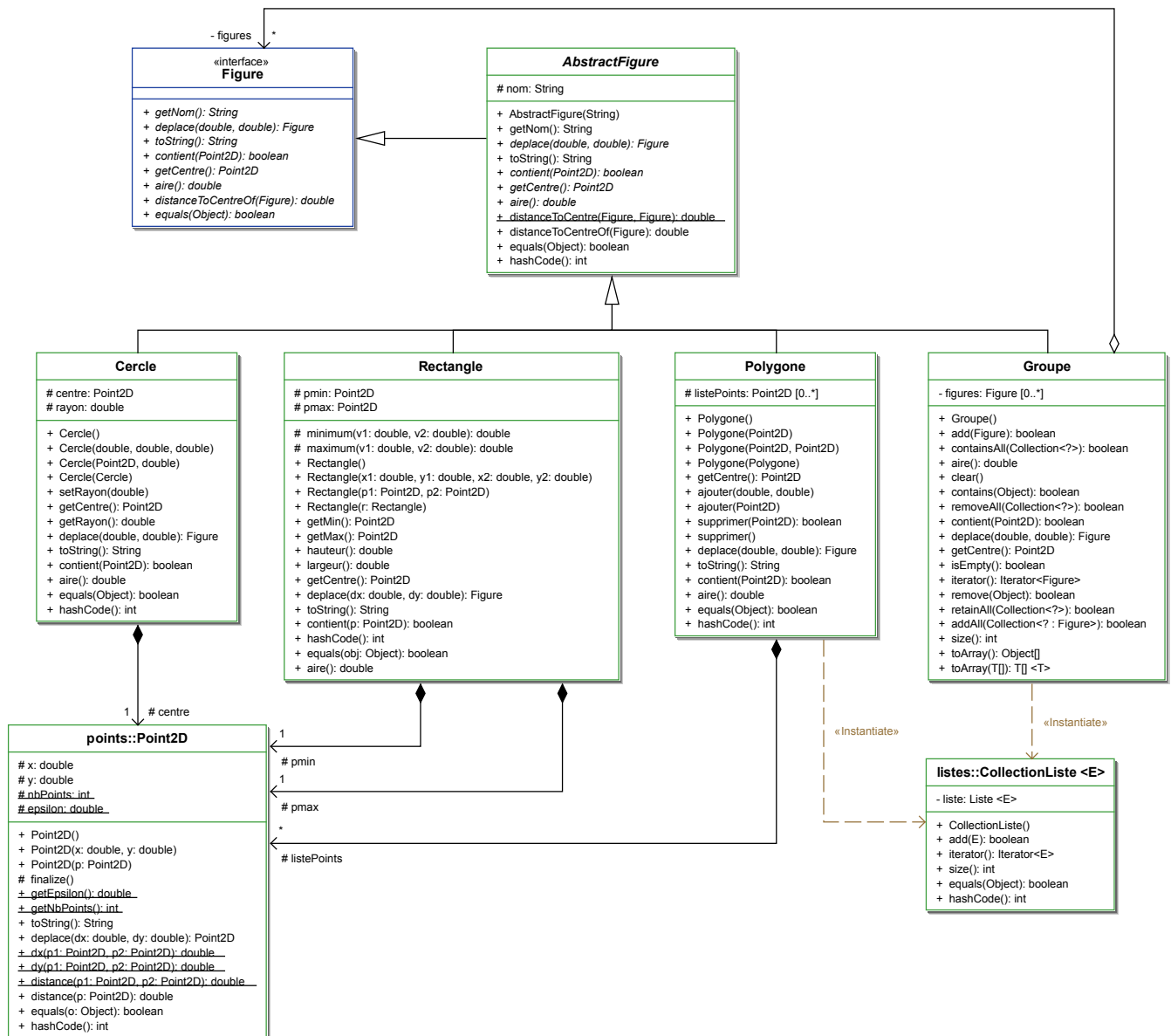
- a) Complétez les méthodes par défaut de l'interface `IListe<E>` utilisant l'itérateur fourni par la « factory method » : `Iterator<E> iterator()`; Les itérateurs permettent de parcourir tout type de collection (ou plus exactement des `Iterable<E>`) en utilisant uniquement les 3 méthodes suivantes sur l'itérateur :
- `boolean hasNext()` : indique s'il reste des éléments à itérer dans la collection.
 - `E next()` throws `NoSuchElementException` : fournit l'élément pointé par l'itérateur puis avance d'un cran. S'il n'y a pas de prochain élément possible lève une `NoSuchElementException`.
 - `void remove()` throws `IllegalStateException` : retire de la collection l'élément qui vient d'être renvoyé par `next()`. Lève une `IllegalStateException` si la méthode `next` n'a pas été appelée au préalable, ou dans tout autre cas qui rendrait l'itérateur incohérent.
 - On ne peut donc pas utiliser `remove()` sans avoir utilisé `next()` auparavant : il doit donc y avoir une alternance entre les appels de `next` et `remove`.
- b) Générez le squelette de la classe `Liste<E>` implémentant l'interface `IListe<E>` et contenant une classe imbriquée (static) privée `Maillon<E>` et une classe interne privée `ListeIterator<E>` implémentant l'interface `Iterator<E>`.
- c) Implémentez le `Maillon<E>` :
- Définissez ses attributs, puis utilisez les capacités de votre IDE à générer automatiquement les accesseurs à ces attributs.
- d) Puis implémentez la classe `Liste<E>` en respectant l'interface `IListe<E>`.
- Implémentez un constructeur par défaut qui crée une liste vide ainsi qu'un constructeur de copie.
 - Puisque cette liste est aussi un `Iterable<E>` cela suppose qu'elle fournisse au travers de la « factory method » `iterator()` un itérateur qui permet de parcourir la liste (en l'occurrence `ListeIterator<E>` qui lui-même implémentera l'interface `Iterator<E>`). On essaiera autant que faire se peut d'utiliser cet itérateur pour réaliser les opérations de la liste comme on l'a déjà fait dans l'interface `IListe<E>`.
 - On considérera que l'on ne peut pas ajouter ou insérer d'éléments null à la liste. C'est pourquoi les méthodes d'ajout et d'insertion soulèvent des `NullPointerException`.
 - Lors du test d'égalité avec un autre Object, on considérera l'égalité avec tous les `Iterable<?>` plutôt qu'avec les `Liste<?>` seules puisque `Liste<E>` est aussi un `Iterable<E>`.
- e) Implémentez ensuite la classe `ListeIterator<E>` qui permettra de concrétiser l'itérateur sur la liste.
- Les conventions à utiliser dans l'itérateur sont les suivantes : on considère que l'on est en fin de liste lorsque le maillon courant (`current`) est null. Vous pourrez vous inspirer pour la méthode `remove()` de la solution proposée dans l'annexe « Suppression d'un élément de liste dans l'itérateur » page 5 qui propose une solution basée sur 3 maillons (`current`, `last` & `penultimate`) pour supprimer le dernier maillon renvoyé par `next()`. Notez qu'il est important d'utiliser dans le `ListeIterator` l'attribut « head » présent dans la classe `Liste` et non pas une copie car la méthode `remove` du `ListeIterator` doit pouvoir éventuellement modifier l'attribut « head » (voir le cas n°2 page 5) lors de la suppression de la tête de la liste.
- f) Et enfin, implémentez la classe `CollectionListe<E>` qui héritera de la classe `AbstractCollection<E>` tout en utilisant comme conteneur une `Liste<E>`.
- Ajoutez un constructeur par défaut pour créer une collection vide ainsi qu'un constructeur de copie à partir d'une `Collection<E>` plutôt que d'une `CollectionListe<E>`.
 - N'oubliez pas de réimplémenter la méthode `add` de la `CollectionListe<E>` car l'implémentation par défaut de `AbstractCollection<E>` ne fait que soulever une exception.
 - Les autres méthodes à réimplémenter dans la `CollectionListe<E>` sont :
 - `iterator()` en utilisant l'`iterator()` de la liste

- size()
- equals(Object) avec éventuellement hashCode()

Un programme utilisant la classe Liste<E> se trouve dans TestListe.java. Toutefois les véritables classes de tests (au sens de JUnit) se trouvent dans le package « tests » dans ListeTest.java et CollectionListeTest.java : Lancez ces tests pour vérifier le bon fonctionnement de vos classes (Run As → JUnit Test).

2) Figures

On cherche maintenant à réaliser la hiérarchie de classes correspondant aux figures dans un package « figures ». On supposera que l'on dispose d'un package « listes » contenant une CollectionListe<E> et d'un package « points » contenant le Point2D et son descendant Vecteur2D permettant d'introduire les notions de produits scalaire et vectoriel utilisables dans les polygones.



a) Complétez une classe abstraite nommée AbstractFigure qui servira de classe mère à toutes les figures que nous pourrons faire par la suite (cercle, rectangle, triangle, etc.) et respectant l'interface définie dans « Figure.java ».

L'interface Figure contient les méthodes suivantes que devront implémenter toutes les figures suivantes :

- « deplace » pour déplacer une figure
- « toString » pour l'affichage

- « equals » qui teste l'égalité de contenu de deux figures
 - « getCentre » qui renvoie le point2D correspondant au barycentre de la figure
 - « contient(Point2D point) » qui vérifie si un point donné en argument est contenu dans la figure
 - « distanceToCentre » qui calcule la distance entre deux centres de figures quelles que soient ces figures
 - « aire » qui calcule l'aire couverte par la figure.
- b) Définissez la classe Cercle contenant un point central et un rayon double.
- c) Définissez la Rectangle droit avec un point en bas à gauche et un point en haut à droite.
- d) Définir la classe Polygone contenant une collection de points (par exemple une `CollectionListe<Point2D>`) ainsi que les méthodes nécessaires pour ajouter ou enlever des points.

Un programme exemple utilisant les figures se trouve dans `TestListe.java`. La classe de test des Figures (pour toutes les figures) se trouve dans `FigureTest.java` : Lancez ce test pour vérifier le bon fonctionnement de vos classes.

- e) S'il vous reste du temps, définissez de la manière la plus simple et la plus rapide possible un « Groupe » de figure qui est une Figure comme les autres qui contient d'autres figures (elle doit donc implémenter l'interface de `Collection<Figure>`). Ajoutez un constructeur à partir d'une `Collection<Figure>`.

Annexes

Il est fortement conseillé d'utiliser un environnement de développement intégré comme Eclipse, NetBeans ou IntelliJ pour développer efficacement en Java. Néanmoins, il est aussi bon de savoir quels sont les outils sur lesquels se basent ces environnements de développement.

Java Development Kit

Le Java Development Kit (ou JDK) est l'ensemble des outils nécessaires à la création d'applications Java. Il comprend (parmi d'autres) les outils suivants : un compilateur (`javac`), une machine virtuelle pour exécuter les objets compilés (`java`), et un utilitaire pour documenter les classes que vous avez écrites (`javadoc`).

Compilation

« `javac` » : le compilateur java.

Ce compilateur génère à partir du fichier `XXX.java` les différents fichiers `XXX.class` correspondant à chacune des classes définies dans le fichier `XXX.java`. On comprendra aisément qu'il est donc judicieux de ne définir qu'une seule classe par fichier afin d'avoir une correspondance `*.class ↔ *.java`.

Pour invoquer la compilation d'un ou plusieurs fichiers `.java`, on utilise la commande suivante :

```
> javac MaClasse.java [MonAutreClasse.java ...]
```

En principe, la compilation d'un fichier dans lequel on utilise d'autres classes définies dans le même répertoire provoque la compilation de ces autres fichiers. Si tant est que l'on a respecté l'adéquation `*.class ↔ *.java`. Ces fichiers `*.class` sont des fichiers contenant du « bytecode » qui correspond au code exécutable par une machine virtuelle Java. Cette machine virtuelle permet de s'affranchir du système d'exploitation propre à telle ou telle machine. N'importe quelle machine virtuelle (quelle que soit la plateforme utilisée) est capable d'exécuter du « bytecode » même compilé sur un autre type de machine.

Exécution

« `java` » : La machine virtuelle proprement dite.

Pour que la machine virtuelle puisse invoquer l'exécution d'une classe compilée (un fichier bytecode), il faut que cette classe contienne une méthode `main` définie comme suit :

```
public static void main(String args[]) {...}.
```

Pour invoquer l'exécution de la classe contenue dans le fichier compilé « MaClasse.class » on utilisera la ligne de commande suivante :

```
> java MaClasse
```

Les browsers Internet possèdent eux aussi une machine virtuelle qui leur permet d'exécuter des classes java. Néanmoins, ils n'exécutent pas des programmes Java (une classe qui possède une méthode main) mais des Applets (APPLication gadGETS).

Documentation

« **javadoc** » : utilitaire de documentation.

Javadoc est un utilitaire qui permet de documenter les classes que vous écrivez. La documentation est alors générée au format hypertexte (html) de la même manière que la documentation du JDK. La documentation d'une classe avec javadoc utilise des commentaires particuliers à l'intérieur du fichier *.java.

```
> javadoc MaClasse.java
```

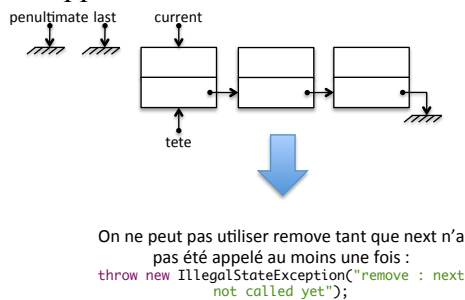
La documentation du JDK est générée avec ce même outil et est accessible à l'adresse suivante :

<http://docs.oracle.com/javase/8/docs/api/>

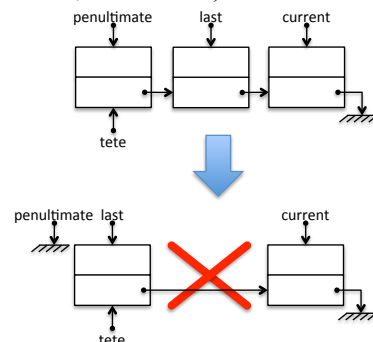
Suppression d'un élément de liste dans l'itérateur

Appel de la méthode remove de l'itérateur.

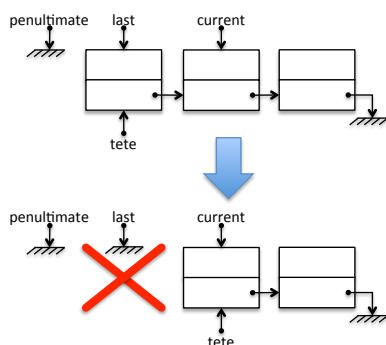
Cas n°1 (cas initial) : la méthode next n'a pas encore été appelée



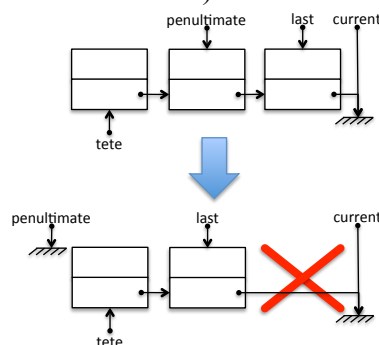
Cas n°3 : l'élément courant se trouve en milieu de liste (ni au début, ni à la fin)



Cas n°2 : la méthode next a été appelée une fois



Cas n°4 : l'élément courant se trouve en fin de liste (cas particulier du cas n° 3)



N'oubliez pas par ailleurs que `remove` ne peut pas être appelé deux fois de suite sans que `next` soit appelé entre temps.