ensIIE
Ecole Nationale Supérieure d'Informatique pour l'Industrie et l'Entreprise

# Langages Objets:

# TP Collections

## 1 Introduction

Le but de ce TP est de vous faire implémenter 2 collections (l'une utilisant une liste doublement chaînée et l'autre utilisant un tableau) en se basant sur l'implémentation partielle de l'interface `Collection<E>` apportée par la classe abstraite `AbstractCollection<E>`.

### 1.1 Implémentation à partir de `AbstractCollection<E>`.

Si l'on devait implémenter une collection directement il faudrait écrire une classe qui implémente les 16 méthodes abstraites de l'interface `Collection<E>`.

La classe `AbstractCollection<E>` est une classe abstraite qui peut nous faciliter la tâche en implémentant une grande partie de l'interface `Collection<E>` grâce à l'utilisation de la template méthode `Iterator<E> iterator()` qui permet d'utiliser un itérateur abstrait dans la classe `AbstractCollection<E>` même si seules les classes filles de cette classe fourniront un itérateur concret.

Implémenter une nouvelle collection concrète par héritage de la classe `AbstractCollection<E>` ne requiert donc que peu de membres. Il faudra implémenter dans nos classes :

- Un conteneur interne : une liste chaînée, un tableau, ou bien une autre collection.
- Des constructeurs :
  - `...Collection()` Un constructeur par défaut pour créer une nouvelle collection vide.
  - `...Collection(Collection<E> c)` : Un constructeur de copie à partir d'une autre collection.
- Une surcharge de la méthode `boolean add(E e)` car celle apportée par la classe `AbstractCollection<E>` ne fait que lever une `UnsupportedOperationException`.
  - Cette méthode renvoie `true` si l'objet `e` a été ajouté avec succès à la collection et `false` dans le cas contraire.
  - On considérera que cette méthode doit lever une `NullPointerException` si l'on tente d'ajouter un élément `e == null`.
- Une implémentation de la méthode `int size()` indiquant le nombre d'éléments dans la collection.
- Une implémentation des méthodes `boolean equals(Object o)` ainsi que `int hashCode()` car la classe `AbstractCollection<E>` ne surcharge pas ces méthodes héritées de la classe `Object`.
- Une implémentation de la factory méthode `Iterator<E> iterator()` fournissant un itérateur sur notre conteneur interne.
  - Il faudra donc pour ce faire implémenter des classes implémentant l'interface `Iterator<E>` et permettant de parcourir les éléments de notre conteneur interne.

### 1.2 Utilisation d'un `Iterator<E>`

L'interface `Collection<E>` dérive de l'interface `Iterable<E>` qui ne définit qu'une seule méthode abstraite : `Iterator<E> iterator()`. C'est grâce à cette "Factory Method" que la classe `AbstractCollection<E>` a pu implémenter en partie l'interface `Collection<E>`. Toute collection concrète est donc responsable de la

création à travers la méthode `Iterator<E> iterator()` d'un `Iterator<E>` concret qui permettra d'itérer sur tous les éléments de la collection.

Une classe qui implémente l'interface `Iterator<E>` doit implémenter les opérations suivantes:

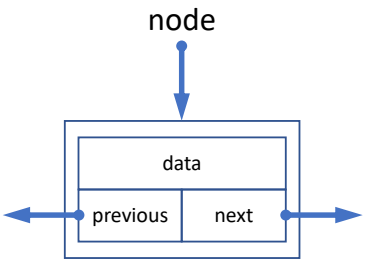
- `boolean hasNext()` : pour indiquer s'il reste des éléments à itérer.
- `E next() throws NoSuchElementException` : qui permet de renvoyer la valeur de l'élément courant de l'itération tout en passant à l'élément suivant. Un nouvel appel à cette méthode renverra la valeur de l'élément suivant puis passera au suivant et ainsi de suite.
  - S'il n'y a plus d'éléments à itérer (lorsque `hasNext()` est faux par exemple) et que l'on tente d'utiliser la méthode `next()`, celle-ci doit lever une `NoSuchElementException`
- `void remove() throws IllegalStateException` : permet de supprimer de la collection sous-jacente l'élément **qui vient d'être renvoyé** par la méthode `next()`.
  - Si l'on tente d'utiliser la méthode `remove()` sans avoir utilisé la méthode `next()` au préalable, la méthode `remove()` doit lever une `IllegalStateException`. On ne peut donc pas appeler la méthode `remove()` deux fois de suite; il faut qu'il existe un appel à `next()` entre les deux.

# 2 Implémentation d'une `Collection<E>` en utilisant les nœuds d'une liste chaînée : `NodeCollection<E>`.

Nous allons maintenant construire les classes nécessaires à l'implémentation d'une classe `NodeCollection<E>` qui utilisera comme conteneur une chaîne de nœuds (`Node<E>`) et comme itérateur un `NodeIterator<E>` qui permettra de parcourir la chaîne de nœuds.

Ces classes se trouvent dans le package `collections/nodes`.

## 2.1 Classe `Node<E>`

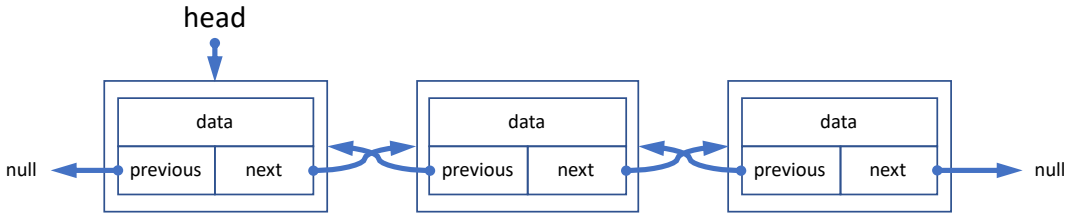



Cette classe représente un nœud d'une liste chaînée :

Ce nœud contient :

- `-data : E` : Une donnée.
- `-previous : Node<E>` : Un lien vers un nœud précédent qui peut être `null` s'il n'y a pas de nœud précédent.
- `-next : Node<E>` : Un lien vers un nœud suivant qui peut être `null` s'il n'y a pas de nœud suivant.

Une liste chaînée est donc constituée d'une chaîne de nœuds commençant par un nœud `head` qui n'a pas de précédent et se termine par un nœud qui n'a pas de suivant :




➦**Complétez la classe `Node<E>`**

➦**Testez là avec la classe de test `tests/NodeTest`.**

## 2.2 Interface Headed<E>

L'interface `Headed<E>` est une interface qui sera utilisée par toute structure ou collection qui possède un nœud de tête (comme dans la figure ci-dessus) suivi d'autres nœuds.

Elle définit donc uniquement des opérations pour lire ou changer ce nœud de tête :

- `Node<E> getHead()` : pour obtenir le nœud de tête
- `void setHead(Node<E> head)` : pour mettre en place un nouveau de tête (ce qui peut arriver quand la tête de liste change par exemple).

Par la suite, toutes les collections qui utiliseront des nœuds (`Node<E>`) implémenteront cette interface. Et plus exactement cette interface sera utilisée par les itérateurs sur ces collections de manière à pouvoir connaître le nœud de tête (avec la méthode `Node<E> getHead()`) mais aussi de manière à pouvoir changer le nœud de tête (avec la méthode `void setHead(Node<E> head)`) lors d'un appel à la méthode `remove()` par exemple.

## 2.3 Classe NodeIterator<E>

Cette classe implémente l'interface `Iterator<E>` et permet de voyager (d'itérer) dans une chaîne de `Node<E>`.

Elle contient (*en notation UML*) :

- `#headed : Headed<E>` : une référence au conteneur du nœud de tête (qui sera une référence vers la collection laquelle nous allons itérer)
- `#next : Node<E>` : une référence au prochain nœud dont on doit renvoyer la valeur lors d'un prochain appel à la méthode `+next() : E`.
- `#lastReturned : Node<E>` : une référence au dernier nœud renvoyé par la méthode `+next() : E`. Ce qui facilitera notre travail si ce nœud doit être supprimé avec un appel à la méthode `+remove()`.
- `#nextCalled : boolean` : un booléen indiquant si la méthode `next()` a été appelée et s'il est maintenant possible d'utiliser la méthode `remove()`. Ce booléen nous servira à assurer l'alternance entre ces deux dernières méthodes.
- Un constructeur à partir d'une instance de `Headed<E>` afin d'obtenir l'élément en tête de la liste de nœuds.
- Une implémentation des méthodes :
  - `+hasNext() : boolean`
  - `+next() : E`
    - Qui doit lever une `NoSuchElementException` s'il est impossible d'itérer plus loin.
  - `+remove()`
    - Qui doit lever une ou des `IllegalStateException` s'il s'avère impossible de réaliser l'opération (lorsque `next()` n'a pas été appelé au préalable par exemple).

➦**Complétez la classe `NodeIterator<E>`.**

➦**Vous pourrez tester la classe `NodeIterator<E>` avec les classes de test**

- `tests/NodeIteratorTest` et
- `tests/IteratorTest`

La Figure 1 ci-dessous résume les classes `Node<E>`, et `NodeIterator<E>`.

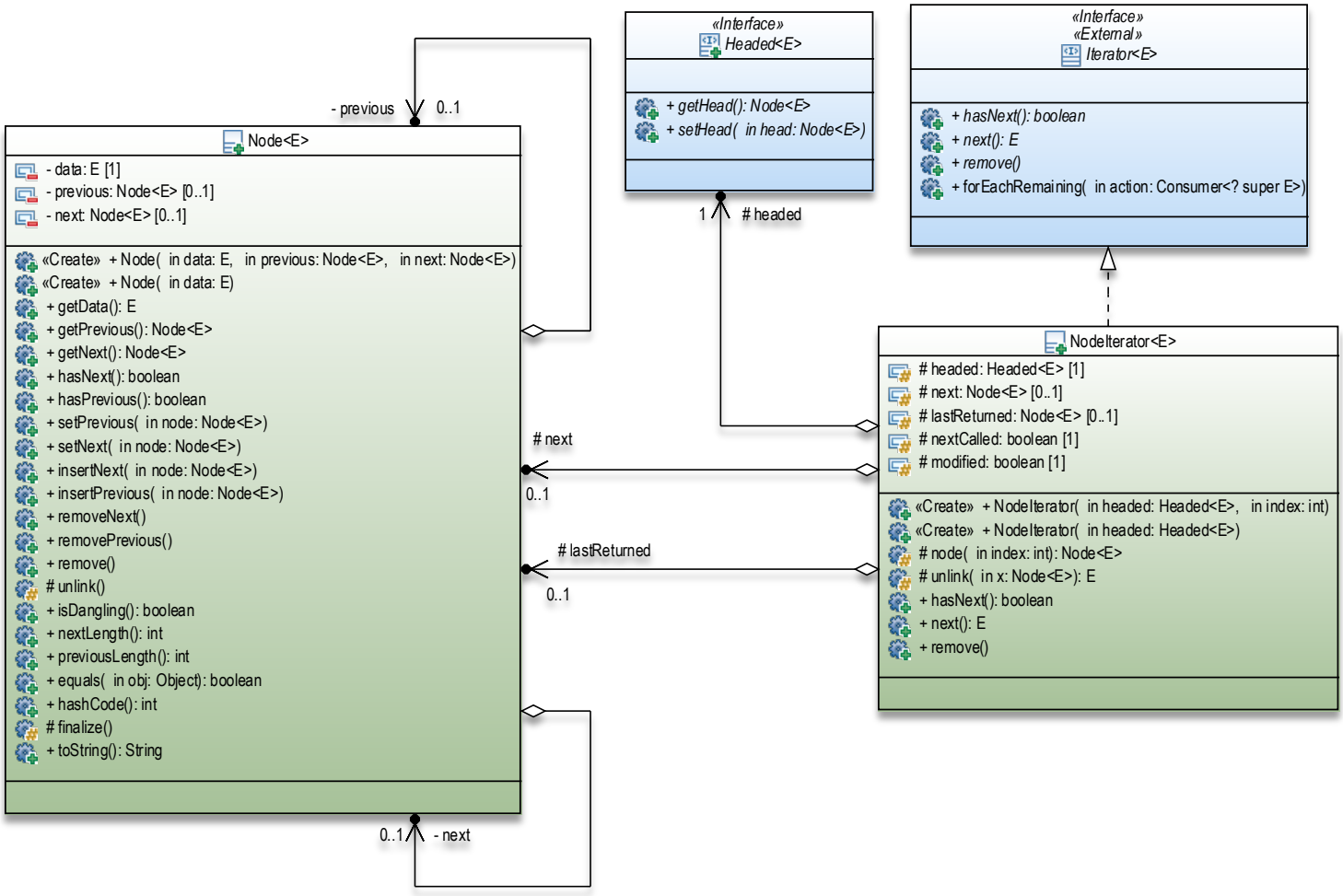



*Figure 1 : Node<E> et NodeIterator<E>*

## 2.4 Classe `NodeCollection<E>`

Maintenant que nous avons créé :

- La classe `Node<E>` pour gérer les nœuds contenant des données.
- La classe `NodeIterator<E>` pour itérer sur ces nœuds de données.

Nous sommes prêts pour implémenter une collection `NodeCollection<E>` par héritage de la classe `AbstractCollection<E>` qui utilisera une chaîne de nœuds de données.

La classe `NodeCollection<E>` hérite de la classe abstraite `AbstractCollection<E>` et implémente l'interface `Headed<E>`. Elle contient:

- `- head : Node<E>` : Un nœud de tête que l'on pourra manipuler grâce aux méthodes de l'interface `Headed<E>`.
- Des constructeurs :
  - Un constructeur par défaut qui crée une collection vide.
  - Un constructeur de copie à partir d'une autre `Collection<E>`.
- Une surcharge de la méthode `+add(E e) : boolean`
  - Cette méthode est concrète dans la classe `AbstractCollection<E>` mais son implémentation ne fait que lever une `UnsupportedOperationException`.
  - On considérera qu'il est impossible d'ajouter un élément `null` dans nos collections et on lèvera une `NullPointerException` si l'élément fourni `e` est `null`.

  - L'implémentation concrète consiste à ajouter un nouveau nœud contenant la donnée `e` après le dernier nœud de la chaîne. D'où l'intérêt de la méthode `#lastNode() : Node<E>` qui permet d'accéder au dernier nœud de la chaîne.
- Une implémentation concrète des méthodes restées abstraites dans la classe `AbstractCollection<E>`
  - `+size() : int` : pour compter le nombre d'éléments dans cette collection, ce qui revient ici à compter le nombre de nœuds à partir du nœud de tête.
  - `+iterator() : Iterator<E>` : pour créer un nouvel itérateur sur cette collection. On renverra ici une nouvelle instance de la classe `NodeIterator<E>` que vous avez complétée précédemment.
- Une implémentation des méthodes de l'interface `Headed<E>`
  - `+getHead() : Node<E>` : pour renvoyer le nœud de tête
  - `+setHead(Node<E> head)` : pour modifier le nœud de tête
- Une surcharge de méthodes héritées de la classe `Object` :
  - `+hashCode() : int` : pour calculer le code de hachage de cette collection (voir le cours sur les classes standard Java).
  - `+equals(Object obj)` : pour tester l'égalité avec un autre objet en termes de contenu (voir aussi le cours sur les classes standard Java).

Remarque : Si la classe `NodeIterator<E>` avait été déclarée comme une classe **interne** à la classe `NodeCollection<E>`, celle-ci aurait pu accéder directement au nœud `-head` et nous n'aurions pas eu besoin d'implémenter l'interface `Headed<E>` qui expose le nœud `-head` révélant ainsi en partie la structure de données utilisée dans `NodeCollection<E>`. Néanmoins, `NodeIterator<E>` étant une classe indépendante, elle pourra être réutilisée dans toute autre classe utilisant des nœuds de liste chaînée.

La Figure 2 ci-dessous résume les relations de la classe `NodeCollection<E>` :

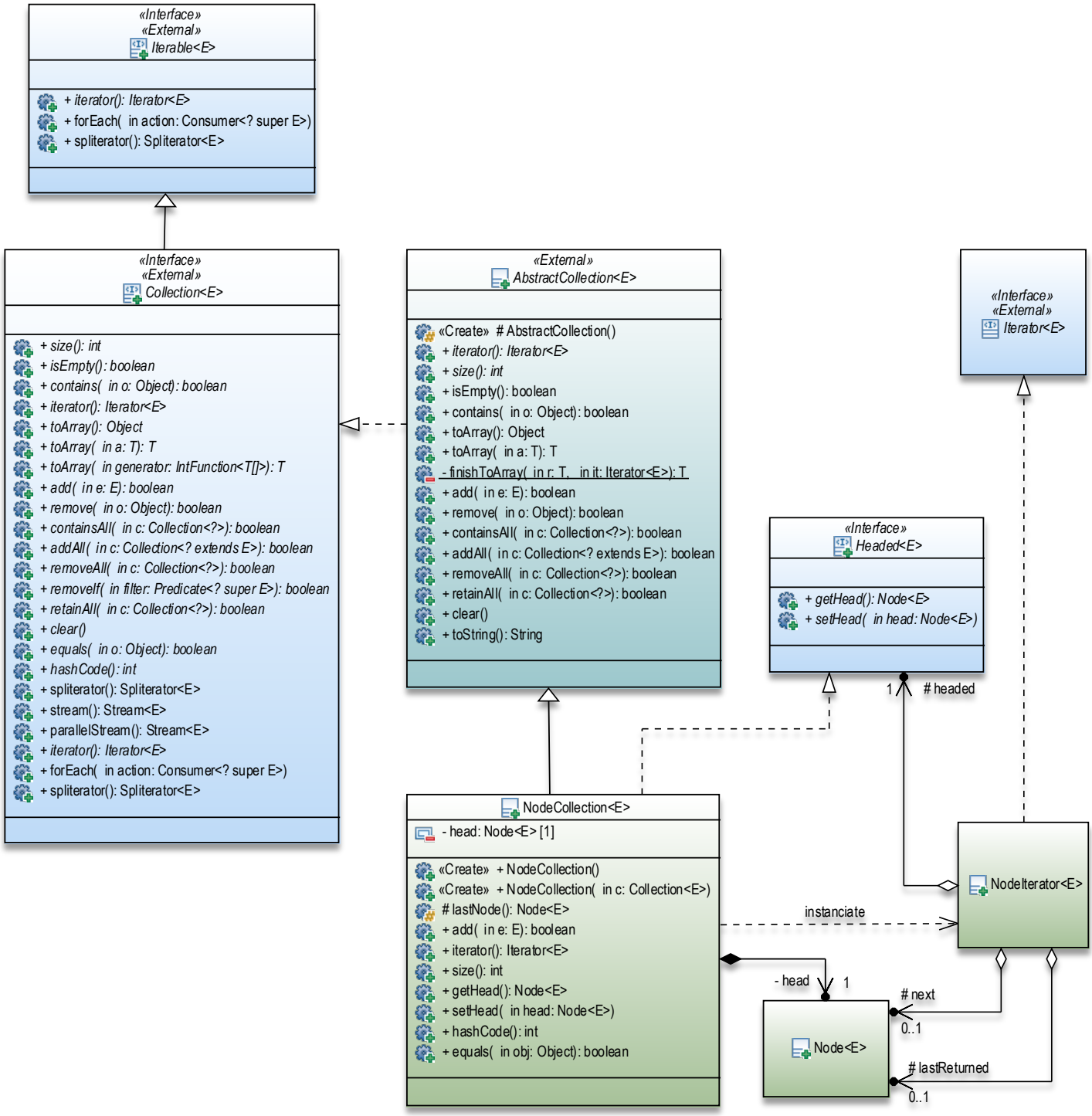



*Figure 2 : NodeCollection<E>*

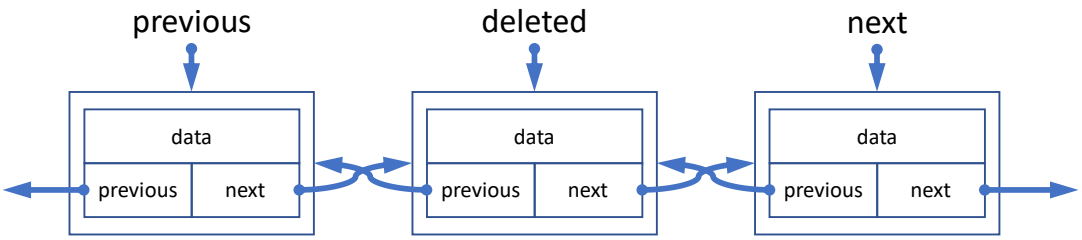
➡**Complétez la classe `NodeCollection<E>`**

➡**Vous pourrez tester la classe `NodeCollection<E>` avec la classe de test**

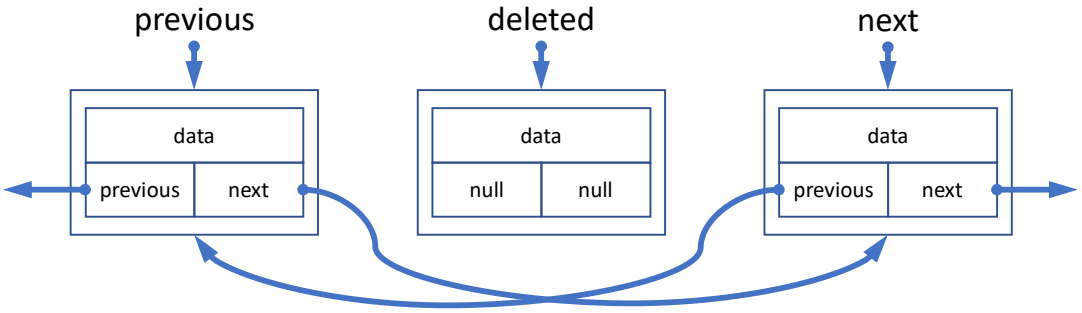
- `tests/CollectionTest`

### 2.4.1 Suppression d'un nœud (pour l'opération `remove()`)

- Lorsqu'une chaîne de nœuds se présente comme suit :




- Supprimer le nœud `deleted` revient à relier les nœuds `previous` et `next` :




- Dans tous les cas, il faudra toujours vérifier que le nœud sur lequel on cherche à travailler soit non null.
- Cas particulier : Lorsque le nœud `previous` est null cela indique que l'on est en train de supprimer le nœud en tête de liste. Il faudra donc changer le nœud de tête pour le nœud `next`.

# 3 Implémentation d'une Collection<E> en utilisant un tableau : ArrayCollection<E>

Nous allons maintenant créer une autre collection qui s'appuiera cette fois sur un tableau d'éléments : `E[]`. Si celui s'avère trop petit pour contenir tous les éléments voulus, il sera alors réalloué.

Les classes suivantes se trouvent dans le package `collections/arrays`.

## 3.1 Implémentation de l'interface Capacity<E>

L'interface `Capacity<E>` représente toute entité contenant un tableau d'éléments `E[]` qui doit pouvoir être réalloué si besoin. La collection que nous allons implémenter devra implémenter cette interface, et l'itérateur sur cette collection utilisera cette même interface. Cette interface définit les opérations (*abstraites*, concrètes et de classe) suivantes :

- *`+getArray(): E[]`* : permet d'obtenir le tableau d'éléments.
- *`+getCapacity() : int`* : permet d'obtenir le nombre d'éléments que l'on peut actuellement stocker dans notre tableau interne (qui peut être différent du nombre d'éléments actuellement stockés dans notre tableau).
- *`+getCapacityIncrement() : int`* : permet d'obtenir le nombre de cases à rajouter au tableau interne en cas de réallocation.
- *`+grow(int amount)`* : permet de réallouer le tableau interne avec `amount` cases supplémentaires.
- `+ensureCapacity(int minCapacity)` : apporte une implémentation par défaut permettant de réallouer (si besoin) le tableau interne avec au moins `minCapacity` cases.
- `+resizeArray(E[] array, int requiredSize) : E[]` : permet de copier le contenu d'un tableau `array` dans un nouveau tableau de `requiredSize` éléments puis de renvoyer ce nouveau tableau. Pourra être utilisé dans l'implémentation de `grow`.

Cette classe définit aussi deux constantes publiques :

- `DEFAULT_CAPACITY` qui définit une taille par défaut pour la capacité d'un tableau
- `DEFAULT_CAPACITY_INCREMENT` qui définit l'incrément de taille par défaut pour un tableau qui a besoin d'être réalloué.

➡ **Complétez l'interface `Capacity<E>`**

## 3.2 Implémentation d'un itérateur spécifique : ArrayIterator<E>

Dans le package `collection.arrays` se trouve une classe : `public ArrayIterator<E> implements Iterator<E>` qui contient :

- `E[] array` : Une référence à un tableau d'éléments qui pourra être initialisée au tableau d'un `ArrayCollection<E>` lors de sa construction.
- `#holder : Capacity<E>` : une référence à une entité contenant un tableau (en l'occurrence une instance de la classe `ArrayCollection<E>`).
- `#index : int` : un entier indiquant l'index courant de l'itération (entre 0 et `array.length - 1` dans le tableau).
- `#size : int` : un entier indiquant le nombre d'éléments non nulls actuellement dans le tableau `#array`. Cet attribut peut éventuellement être remplacé par une méthode `+size() : int`.
- `#lastReturnedIndex : int` : l'index dans le tableau du dernier élément renvoyé par la méthode `+next()` et aussi l'index de l'élément à supprimer par la méthode `+remove()`. Lorsque celui-ci ne peut pas être déterminé (lorsque `next()` n'a pas encore été appelé par exemple) on le mettra -1 par exemple

pour indiquer qu'il n'est pas valide. Cet attribut est l'équivalent de l'attribut `boolean nextCalled` du `NodeIterator<E>`.
- Une implémentation des méthodes `hasNext()`, `next()` et `remove()`.

C'est donc une instance de cet itérateur qui sera retournée par un appel à la méthode `Iterator<E> iterator()` de votre classe `ArrayCollection<E>`.

➦**Complétez la classe `ArraIterator<E>`.**

➦**Vous pourrez tester la classe `ArrayIterator<E>` avec la classe de test**

- `tests/IteratorTest`

## 3.3 Implémentation de la classe `ArrayCollection<E>`

➦ **Créez une nouvelle classe dans le package `collections` : `public ArrayCollection<E> extends AbstractCollection<E> implements Capacity<E>.`**

Cette classe contiendra donc :

- `-array : E[]` : Comme conteneur interne un simple tableau.
  - On considérera que la taille de cette collection correspond au nombre d'éléments **non nulls** contenus dans `array`.
  - De même, la "capacité" de cette collection correspond à la taille de son tableau interne : `array.length`.
- `-capacityIncrement : int` : Un entier indiquant le nombre de cases à rajouter au tableau lorsque celui-ci s'avérera trop petit pour stocker tous les éléments requis de la collection.
  - Lorsque le nombre d'éléments non nulls dans `array` est égal à `array.length` le tableau devra être réalloué avec `capacityIncrement` cases supplémentaires.
  - Nous serons donc amenés à réallouer le tableau interne de temps à autre (voir l'interface Capacity<E>).
- Il faudra donc des constructeurs adéquats pour initialiser tous ces attributs. Ces constructeurs devront lever des `IllegalArgumentException` si les arguments fournis aux constructeurs sont invalides. Typiquement la capacité initiale (`array.length`) doit être positive ou nulle et la capacité d'incrément strictement positive. On ajoutera à cela un constructeur par défaut ainsi qu'un constructeur de copie à partir d'une `Collection<E>`.
  - `public ArrayCollection(int capacity, int capacityIncrement) throws IllegalArgumentException {…}`
  - `public ArrayCollection(int capacity) throws IllegalArgumentException {…}`
  - `public ArrayCollection(){…}`
  - `public ArrayCollection(Collection<E> c) {…}`
- Il faudra implémenter les méthodes de l'interface `Capacity<E>` :
  - `+getCapacity() : int`
  - `+getCapacityIncrement() : int`
- Tout comme la classe `NodeCollection<E>` cette classe devra (re)implémenter les opérations suivantes :
  - `+size() : int`
  - `+add(E e) : boolean`
  - `+iterator() : Iterator<E>` : qui renverra une nouvelle instance de `ArrayIterator<E>`.

La figure ci-dessous résume les classes à implémenter dans cette partie :

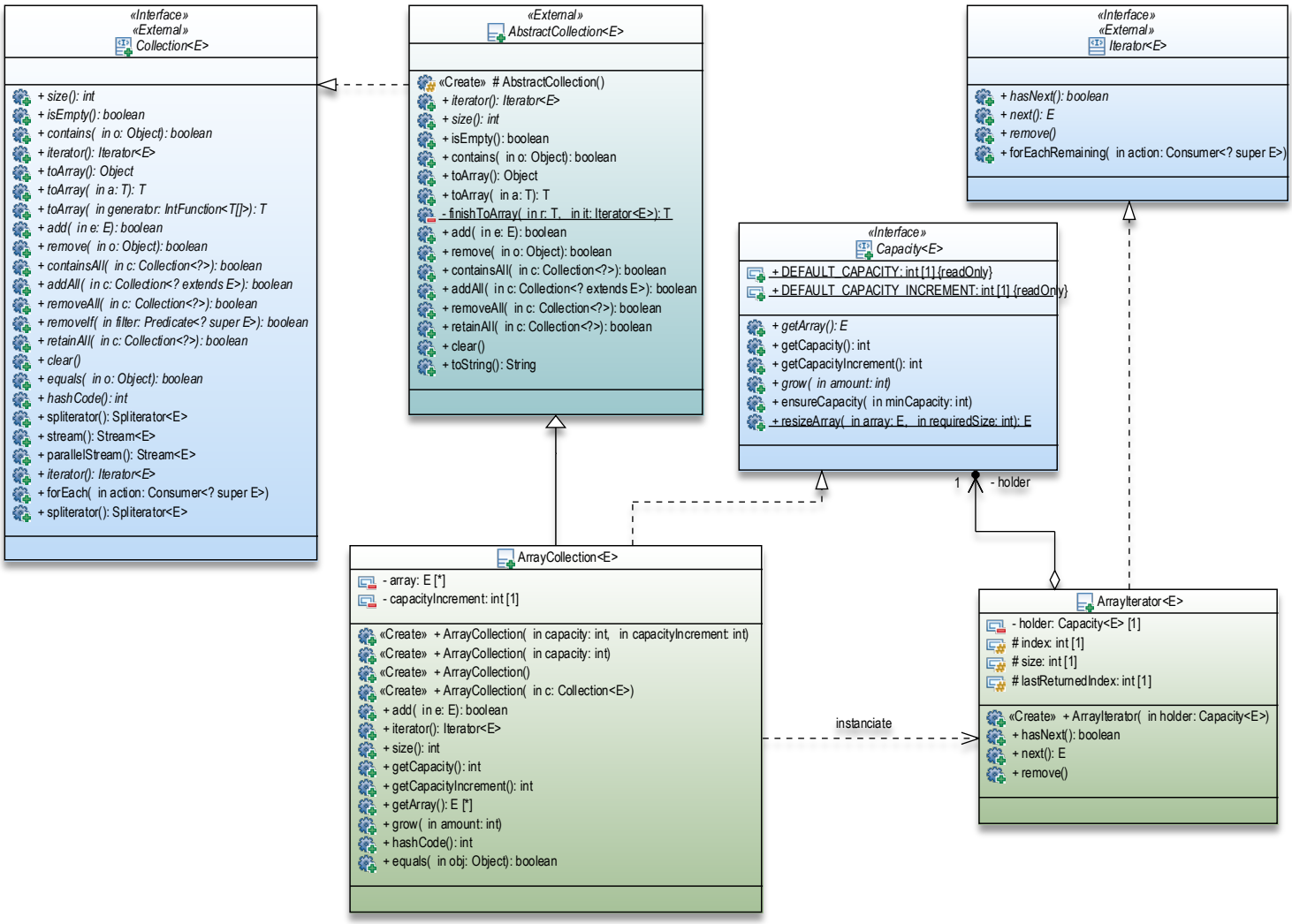



*Figure 3 : ArrayCollection<E> et ArrayIterator<E>*

➡**Vous pourrez tester la classe `ArrayCollection<E>` avec la classe de test**

- `tests/CollectionTest`

## Annexe

La figure ci-dessous résume la hiérarchie de classes dans laquelle vous allez travailler :

- Package `collections`
  - `NodeCollection<E>` : collection utilisant une liste chaînée.
  - `ArrayCollection<E>` : collection utilisant un tableau.
  - Package `nodes`
    - `Node<E>` : nœud de liste chaînée.
    - `NodeIterator<E>` : itérateur de nœuds de liste chaînée, utilisé dans `NodeCollection<E>`.
    - `Headed<E>` : interface définissant le porteur d'un nœud de tête de liste. Implémenté par `NodeCollection<E>` et utilisé par `NodeIterator<E>`.
  - Package `arrays`
    - `Capacity<E>` : interface définissant le porteur d'un tableau d'éléments qui doit pouvoir être réalloué si besoin. Implémenté par `ArrayCollection<E>` et utilisé par `ArrayIterator<E>`.
    - `ArrayIterator<E>` : itérateur sur les éléments d'un tableau, utilisé dans `ArrayCollection<E>`.

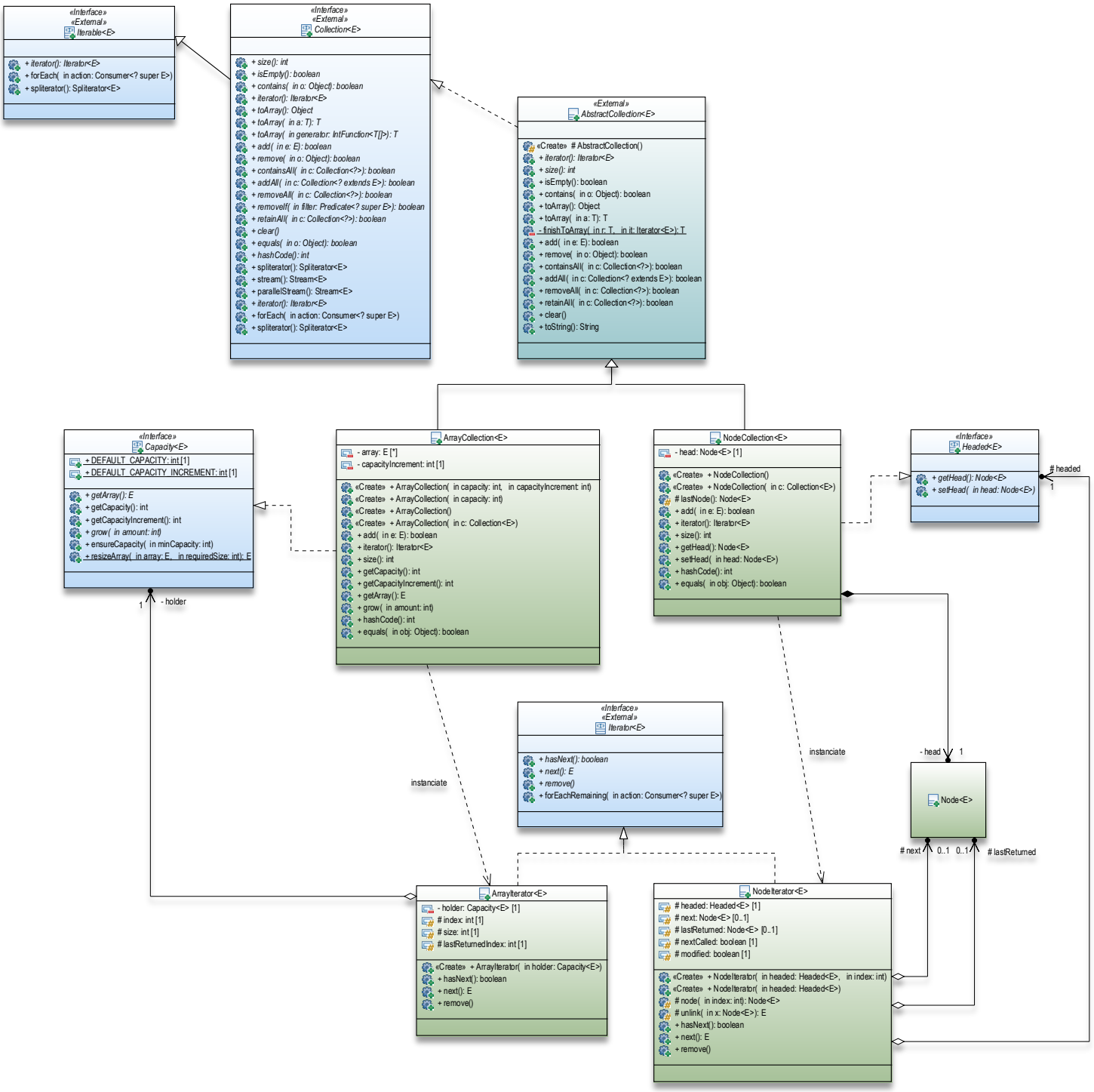



*Figure 4 : Toutes les classes*