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

Langages Objets:

TP Lists

Préambule

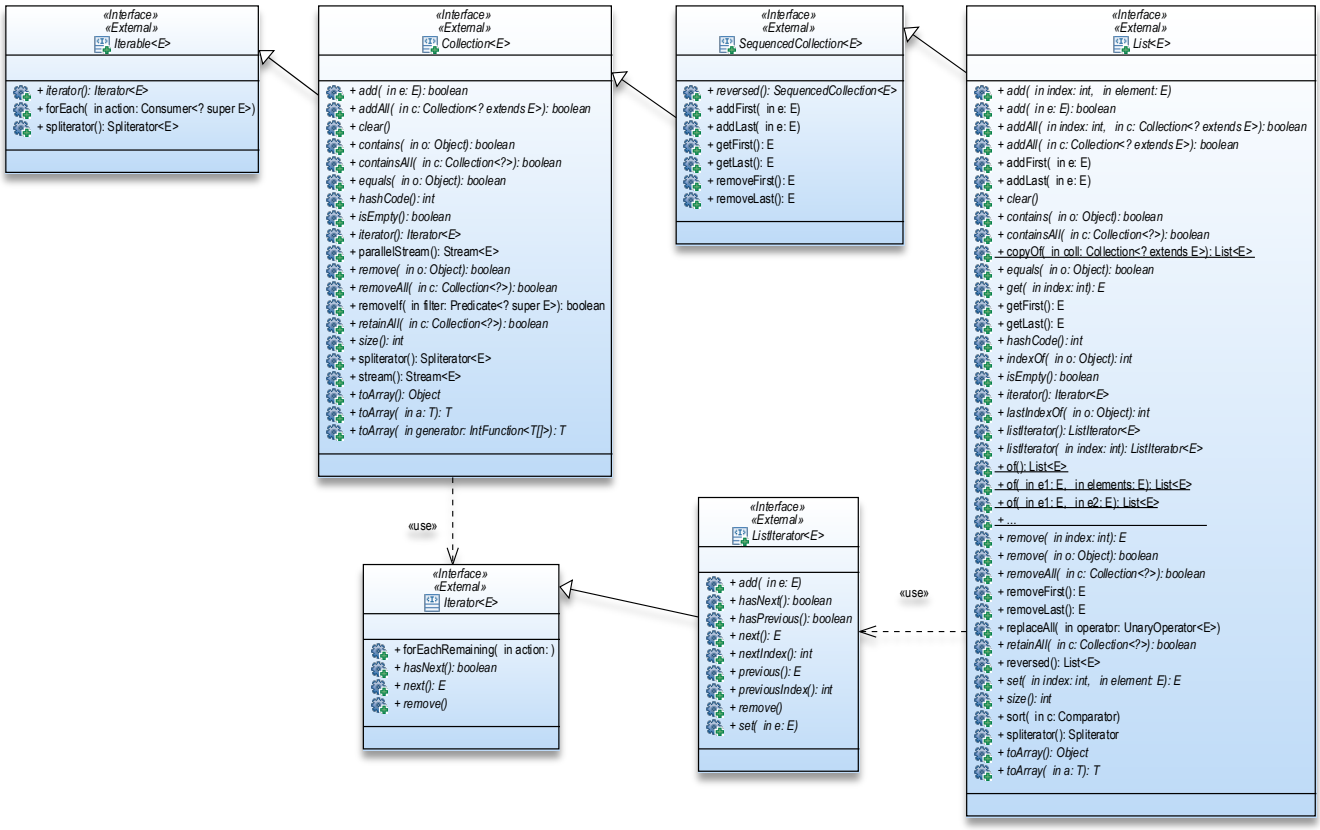
Ce TP s'inscrit dans la continuation directe du TP précédent dans lequel nous avions implémenté des Collection<E> dont l'une d'elle (`NodeCollection<E>`) utilisait des nœuds doublement chaînés (`Node<E>`) que nous allons réutiliser.

1. Introduction

Le but de ce TP est de vous faire implémenter des listes (au sens Java du terme, c'est à dire des classes qui implémentent l'interface `List<E>` qui est une spécialisation de l'interface `Collection<E>`). Une liste en Java est une "collection séquentielle" dans laquelle on peut :

- Accéder aux éléments de début et de fin
- Ajouter ou retirer des éléments au début ou à la fin.
- Accéder à l'élément d'indice `i` dans la séquence d'éléments.
- Ajouter, retirer ou modifier un élément à l'indice `i` dans la séquence d'éléments.
- Trier et inverser la séquence d'éléments.
- Ainsi que d'autres opérations.

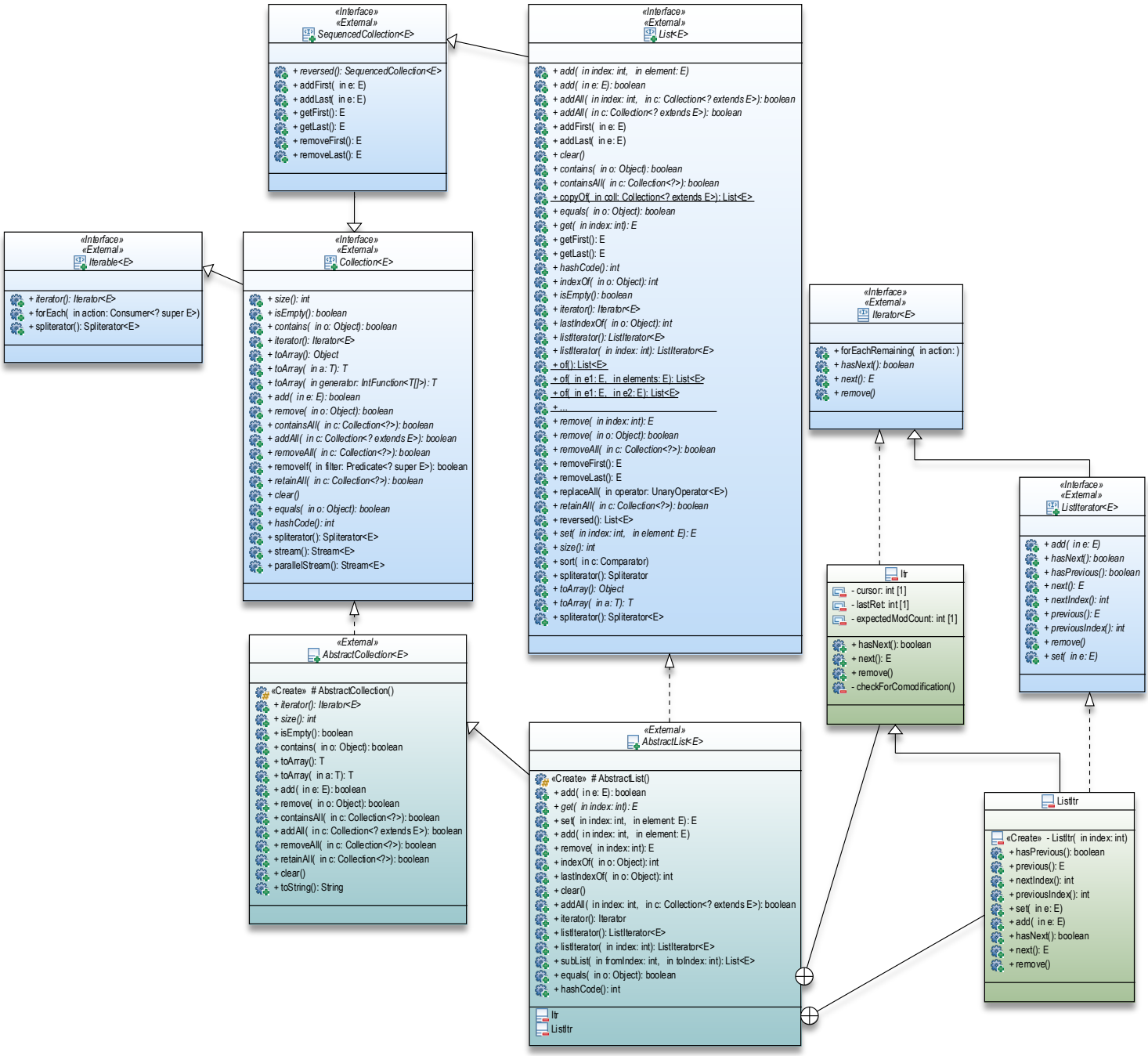
La figure ci-dessous montre les relations entre les Interfaces `Collection<E>` et `List<E>` ainsi que celles entre l'`Iterator<E>` utilisé dans les collections et le `ListIterator<E>` utilisé dans les listes.



Tout comme lors du TP sur les collections nous ne partirons pas de rien pour implémenter nos listes : Le plus simple est d'hériter d'une classe abstraite `AbstractList<E>` qui hérite elle-même de `AbstractCollection<E>` et qui implémente partiellement l'interface `List<E>`.

`AbstractList<E>`

La classe `AbstractList<E>` (voir la figure ci-dessous) est une classe abstraite qui fournit une implémentation quasi complète de l'interface `List<E>` à l'exception de seulement quelques méthodes.



La classe `AbstractList<E>` possède ses propres `Iterator<E>` et `ListIterator<E>` internes : Respectivement `Itr` et `ListItr`. Ces itérateurs sont utilisés dans des méthodes comme `int indexOf(Object o)` et `int lastIndexOf(Object o)` par exemple. Néanmoins, pour implémenter une liste concrète, il faudra implémenter ou surcharger les méthodes suivantes :

- `public E get(int index)` : Qui est abstraite dans la classe `AbstractList<E>` et permet d'obtenir l'élément d'indice `i` de la liste.

  - Cette méthode lèvera une `IndexOutOfBoundsException` si `index` est invalide (en dehors de [0...size() - 1]).
- `public E set(int index, E element)` : Qui doit permettre de remplacer l'élément situé à l'indice `index` par l'élément `element` et de renvoyer l'élément qui a été remplacé.
  - L'implémentation fournie par `AbstractList<E>` ne fait que lever une `UnsupportedOperationException`, c'est pourquoi il sera nécessaire de surcharger cette méthode.
  - Cette méthode doit lever une `IndexOutOfBoundsException` si `index < 0` ou `index >= size()`.
- `public void add(int index, E element)` : qui doit permettre d'ajouter un élément à l'index spécifié.
  - L'implémentation fournie par `AbstractList<E>` ne fait que lever une `UnsupportedOperationException`, c'est pourquoi il sera nécessaire de surcharger cette méthode.
  - Cette méthode lèvera une `IndexOutOfBoundsException` si `index < 0` ou `index > size()`.
- `public E remove(int index)` : Qui doit permettre de supprimer de la liste l'élément à la position `index` et de renvoyer l'élément supprimé.
  - L'implémentation fournie par `AbstractList<E>` ne fait que lever une `UnsupportedOperationException`, c'est pourquoi il sera nécessaire de surcharger cette méthode.
  - Cette méthode lèvera une `IndexOutOfBoundsException` si `index < 0` ou `index >= size()`.

`ListIterator<E>`

Un `ListIterator<E>` est un itérateur spécialement conçu pour itérer sur les listes. L'interface `ListIterator<E>` hérite de l'interface `Iterator<E>`.

- A la différence d'un `Iterator<E>` qui ne peut progresser que vers l'avant (avec les méthodes `hasNext()` et `next()`) un `ListIterator<E>` peut aussi progresser vers l'arrière (avec les méthodes `hasPrevious()` et `previous()`).
- La méthode `nextIndex()` peut indiquer l'index dans la liste de l'élément qui serait renvoyé par `next()` s'il était appelé.
- La méthode `previousIndex()` peut indiquer l'index dans la liste de l'élément qui serait renvoyé par `previous()` s'il était appelé.
- La méthode `remove()` peut supprimer de la liste l'élément qui vient d'être renvoyé par `next()` ou par `previous()`.
- La méthode `set(E)` permet de remplacer dans la liste l'élément qui vient d'être renvoyé avec `next()` ou bien celui qui vient d'être renvoyé avec `previous()` par celui fournit en argument.
- La méthode `add(E)` ajoute l'élément passé en argument dans la liste immédiatement *avant* l'élément qui serait retourné par un appel à `next()` (s'il existe) et *après* l'élément qui serait retourné par un appel à `previous()` (s'il existe).

`AbstractSequentialList<E>`

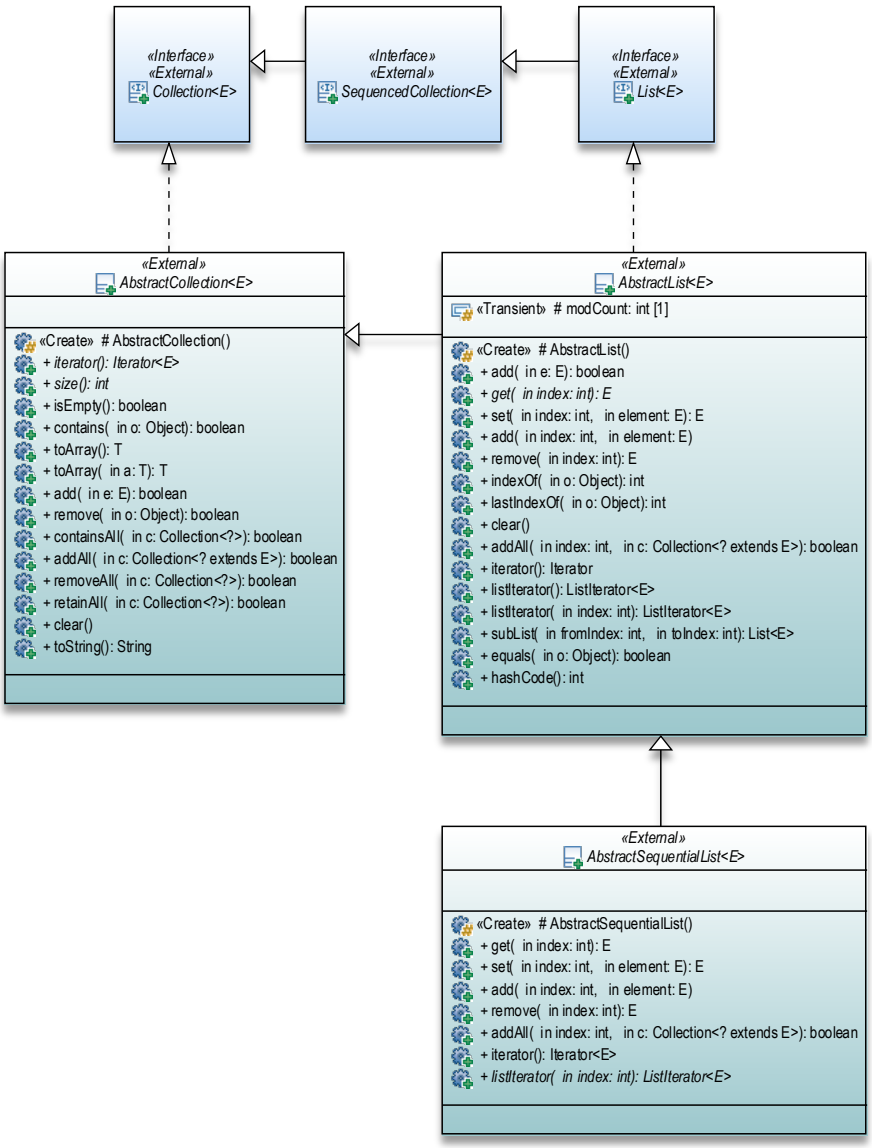
La classe abstraite `AbstractSequentialList<E> extends AbstractList<E>` est un **raffinement** de l'implémentation partielle des listes apportée par la classe abstraite `AbstractList<E>`. Les opérations surchargées par la classe abstraite `AbstractSequentialList<E>` reposent principalement sur l'utilisation du `ListIterator<E>` fournit par la "factory method" `public ListIterator<E> listIterator()` qui reste abstraite dans cette classe et qu'il faudra donc implémenter dans une classe fille concrète.

Les opérations suivantes sont donc concrètement implémentées en utilisant l'itérateur fournit par `public ListIterator<E> listIterator()` :

- `public E get(int index)`

- `public E set(int index, E element)`
- `public void add(int index, E element)`
- `public E remove(int index)`
- `public boolean addAll(int index, Collection<? extends E> c)`
- `public Iterator<E> iterator()`

La figure ci-dessous montre les relations de la classe abstraite `AbstractSequentialList<E>` :



2. Travail à effectuer

Nous allons donc implémenter 2 listes :

- Une classe `NodeList<E>` qui va hériter de la classe abstraite `AbstractList<E>` et utiliser des nœuds doublement chaînés (`Node<E>`) pour stocker les éléments.
- Une classe `NodeSequentialList<E>` similaire à la précédente mais qui va hériter de la classe abstraite `AbstractSequentialList<E>` (héritière de la classe `AbstractList<E>`) et utiliser un `ListIterator<E>` implémenté par un itérateur de liste utilisant des `Node<E>` : `NodeListIterator<E>`.

  - Il faudra donc d'abord implémenter la classe `NodeListIterator<E>` pour pouvoir ensuite l'utiliser dans la "factory method" `ListIterator<E> listIterator()` de la classe `NodeSequentialList<E>`.

Vos classes implémentant l'interface `List<E>` devront aussi fournir les constructeurs suivants :

- Un constructeur par défaut (sans arguments pour être exact).
- Un constructeur de copie avec en argument une `Collection<E>`.

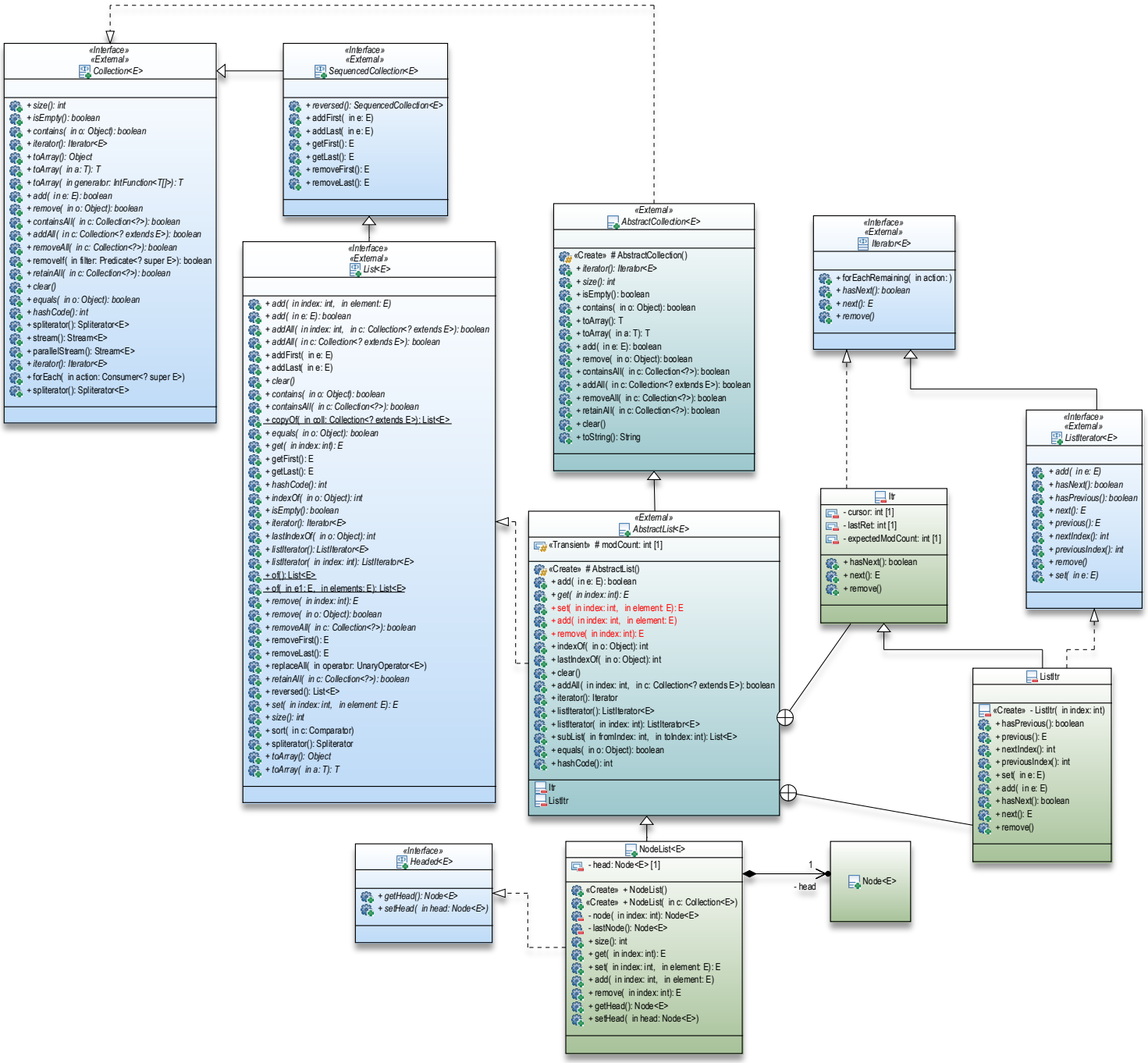
3. `NodeList<E>` dans le package `collections.lists`

La classe *à compléter* `public NodeList<E> extends AbstractList<E> implements Headed<E>` du package `collections.lists` hérite de la classe abstraite `AbstractList<E>` et implémente les méthodes décrites dans l'introduction. Cette classe utilise comme conteneur interne la tête d'une liste chaînée (un nœud de la liste): `Node<E> head`.

Les attributs et méthodes à implémenter dans la classe `NodeList<E>` sont donc les suivantes :

- Un `Node<E>` privé représentant la tête de liste.
- Un constructeur par défaut qui crée une liste vide.
- Un constructeur de copie à partir d'une `Collection<E>`
- Deux méthodes protégées ou privées pour accéder à certains nœuds et que vous pourrez réutiliser dans les autres méthodes à implémenter :
  - `private Node<E> node(int index)` : Qui permet d'accéder au nœud d'index `index` (à partir du nœud de tête `head`) s'il existe.
  - `private Node<E> lastNode()` : Qui permet d'accéder directement au dernier nœud non null de la liste (s'il existe).
- `public int size()` : Renvoie le nombre d'éléments de la liste.
- `public E get(int index)` : Renvoie la valeur contenue dans l'élément d'index `index` dans la liste ou bien lève une `IndexOutOfBoundsException` si l'index est invalide, lorsque `index < 0` ou `index >= size()`.
- `public E set(int index, E element)` : Met en place une nouvelle valeur `element` sur l'élément d'index `index` dans la liste.
  - Lève une `NullPointerException` si `element` est null car on n'autorise pas les éléments nuls dans nos collections.
  - Lève une `IndexOutOfBoundsException` si l'index est invalide.
- `public void add(int index, E element)` : Ajoute un nouvel élément contenant la valeur `element` à l'index `index` dans la liste.
  - Lève une `NullPointerException` si `element` est null car on n'autorise pas les éléments nuls dans nos collections.
  - Lève une `IndexOutOfBoundsException` si l'index est invalide lorsque `index < 0` ou `index > size()` car on considère que l'on peut ajouter un nouvel élément après de dernier élément présent dans la liste.
- `public E remove(int index)` : Retire de la liste des éléments, l'élément à l'index `index`, si et seulement si l'indice est valide. L'index sera invalide lorsque `index < 0` ou `index >= size()`.
  - Lève une `IndexOutOfBoundsException` si l'index est invalide.
- Les méthodes de l'interface `Headed<E>` qui permettent d'accéder au nœud en tête de liste.
  - `public Node<E> getHead()`
  - `public void setHead(Node<E> head)`

La figure ci-dessous décrit les relations de la classe `NodeList<E>` :



➽ Testez la classe `NodeList<E>` avec les classes de test `ListTest` et `CollectionTest`

4. `NodeListIterator<E>` dans le package `collections.nodes`

Avant de pouvoir implémenter la classe `NodeList<E>` il faut tout d'abord implémenter le `ListIterator<E>` sur lequel s'appuie sa classe mère `AbstractSequentialList<E>`.

La classe `NodeListIterator<E> extends NodeIterator<E> implements ListIterator<E>` implémente un itérateur bidirectionnel sur des `Node<E>` utilisable dans des `List<E>` en héritant de la classe `NodeIterator<E>` que nous avons implémentée lors du dernier TP.

La classe `NodeIterator<E>` implémentée lors du dernier TP est constituée des éléments suivants :

- `protected Headed<E> headed` : Référence vers l'entité porteuse du `Node<E>` en tête de liste.
- `protected Node<E> next` : Référence vers le nœud courant de l'itération. Utilisé pour renvoyer une valeur dans la méthode `next()`.

- `protected Node<E> lastReturned` : Référence au dernier nœud renvoyé par un appel à la méthode `next()` et donc aussi une référence vers le nœud à supprimer lors d'un appel à la méthode `remove()`.
- `protected boolean nextCalled` : Flag booléen indiquant que la méthode `next()` a été appelée et qu'il est donc maintenant légal d'appeler la méthode `remove()`. Ce flag est donc mis à `true` dans `next()` et remis à `false` dans `remove()`.
- `public NodeIterator(Headed<E> headed, int index)` : Constructeur valué pour initialiser le porteur du nœud en tête de liste ainsi que le premier nœud lors de de l'itération.
  - Lève une `NullPointerException` si `headed` est null.
  - Lève une `IndexOutOfBoundsException` si l'index est invalide : `< 0` ou `>= <nombre de nœuds>`.
- `public NodeIterator(Headed<E> headed)` : Constructeur valué appelant le constructeur précédent avec un `index = 0`.
- `protected Node<E> node(int index)` : Méthode permettant d'accéder au nœud d'index `index`.
- `protected E collapse(Node<E> x)` : Méthode pour court-circuiter le nœud `x` en reliant son précédent et son suivant ensembles.
- `public boolean hasNext()` : Indique s'il reste des éléments à itérer dans la collection.
- `public E next()` : Fournit la valeur du prochain élément de l'itération.
  - Lève une `NoSuchElementException` si un tel élément n'existe pas.
- `public void remove()` : Retire de la collection l'élément qui vient d'être renvoyé par `next`.
  - Lève une `IllegalStateException` si `next()` n'a pas été appelé au préalable ou bien si la référence du nœud à supprimer (`lastReturned`) est null.

Les membres à implémenter dans la classe *à compléter* `NodeListIterator<E>` sont donc les suivants. Vous devrez autant que faire se peut réutiliser les méthodes déjà présentes dans la classe mère (`NodeIterator<E>`) :

- Le membre hérité `protected Node<E> lastReturned` de la classe `NodeIterator<E>` pointe maintenant sur le dernier nœud contenant la valeur renvoyée soit par `next()` soit par `previous()`. Mais reste dans tous les cas la référence du nœud à supprimer par l'opération `remove()`.
- `private Node<E> last` : Une référence vers le dernier nœud non null de la liste. Ou bien null si un tel nœud n'existe pas. Ce nœud n'est techniquement pas requis mais nous permettra d'accéder plus rapidement aux nœuds en fin de liste.
- `private int size` : Le nombre de nœuds entre `headed.getHead()` et `last`.
- `private int nextIndex` : Index de l'élément renvoyé par `next()` et valeur renvoyée par `nextIndex()`.
- `private boolean previousCalled` : Flag indiquant que la méthode `previous()` a été appelée et qu'il est donc maintenant légal d'appeler la méthode `remove()`. Ce flag est donc mis à `true` dans `previous()` et remis à `false` dans les méthodes `remove()` et `next()`.
- `public NodeListIterator(Headed<E> headed, int index)` : Un constructeur valué qui met en place le porteur du nœud en tête de liste et l'index où doit commencer cet itérateur.
- `public NodeListIterator(Headed<E> headed)` : Un autre constructeur qui appellera simplement le constructeur précédent avec un `index` à 0.
- `protected Node<E> node(int index)` : Une surcharge de la méthode protégée pour accéder au nœud d'index `index`. Vous pourrez proposer une implémentation plus efficace de cette méthode en profitant du fait que :
  - Si `index < size / 2` il est plus avantageux de partir du début de la liste pour aller chercher le nœud désiré.
  - Si `index >= size / 2` il est plus avantageux de partir de la fin de liste (voir le nœud `last`) pour atteindre le nœud désiré.
- `protected E collapse(Node<E> x)` : Méthode pour court-circuiter le nœud `x` en reliant son précédent et son suivant ensembles. On ne peut pas se contenter ici de l'implémentation fournie par la classe mère `NodeIterator<E>` car on doit ici éventuellement mettre à jour les attributs `last` et `size`.

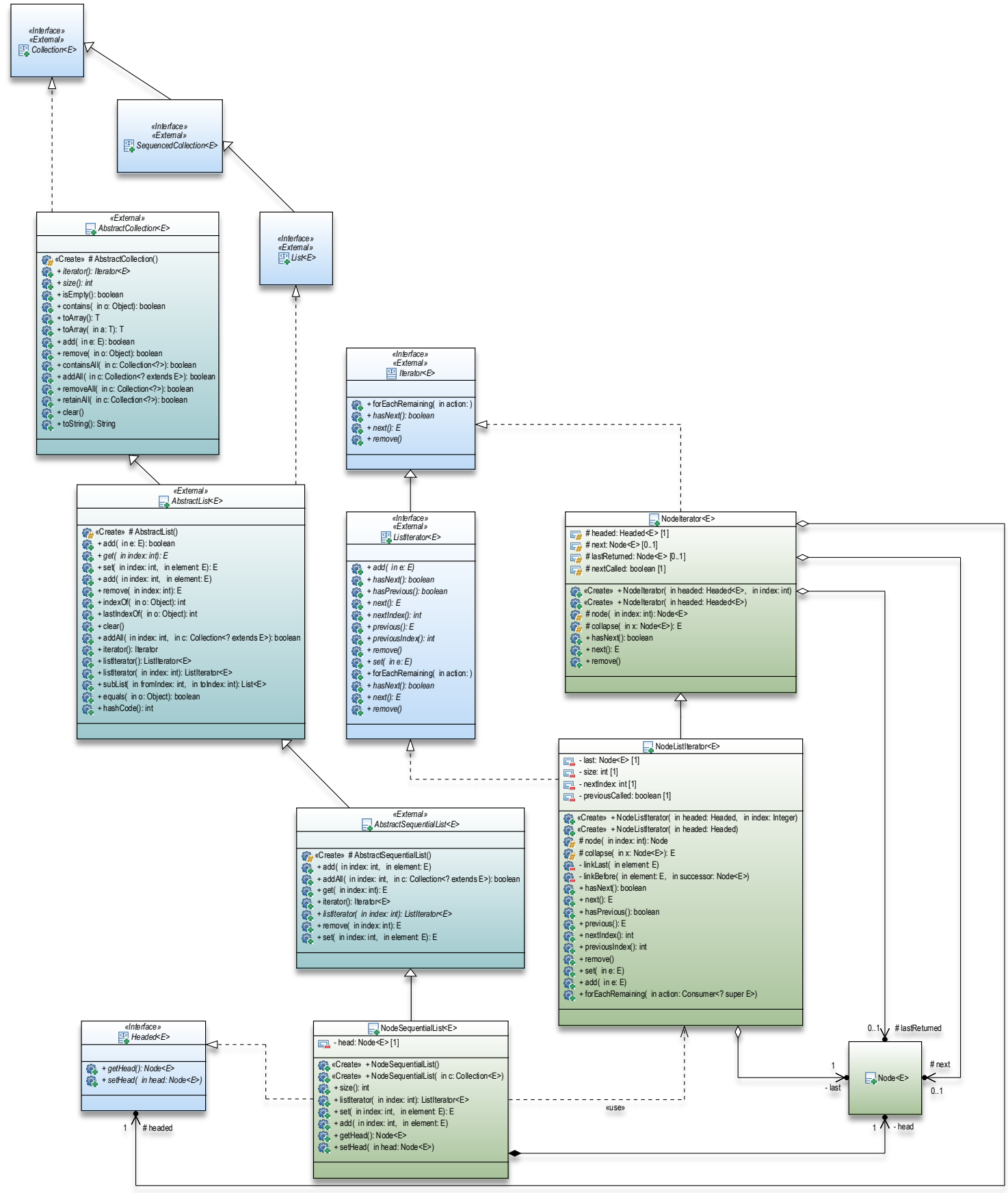
- `public boolean hasNext()` : Indique si cet itérateur a encore des éléments à itérer vers l'avant. Autrement dit si `next()` peut retourner le prochain élément. On pourra utiliser `nextIndex` qui devra rester `< size`.
- `public E next()` : Renvoie la valeur du prochain élément dans la liste. Ou bien lève une `NoSuchElementException` s'il un tel élément n'existe pas.
- `public boolean hasPrevious()` : Indique si cet itérateur a encore des éléments à itérer vers l'arrière. Autrement dit si `previous()` peut retourner l'élément précédent. On pourra utiliser le fait que `nextIndex > 0`.
- `public E previous()` : Renvoie la valeur de l'élément précédent dans la liste. Ou bien lève une `NoSuchElementException` s'il un tel élément n'existe pas.
- `public int nextIndex()` : Index dans la liste du prochain élément renvoyé par un appel à `next()`.
- `public int previousIndex()` : Index dans la liste du prochain élément renvoyé par un appel à `previous()`. En général `nextIndex - 1`.
- `public void remove()` : Retire de la liste l'élément dont la valeur vient d'être renvoyée par `next()` ou bien par `previous()`.
  - Lève une `IllegalStateException` si ni `next()` ni `previous()` n'a été appelé auparavant, ou bien si `lastReturned` est null.
- `public void set(E e)` : Change la valeur de l'élément qui vient d'être renvoyé par `next()` ou par `previous()`.
  - Lève une `IllegalStateException` si un tel élément n'existe pas, lorsque `lastReturned` est null.
  - Lève une `IllegalArgumentException` si la nouvelle valeur `e` est null.
- `public void add(E e)` : Insère un nouvel élément de valeur `e` dans la liste, juste avant l'élément qui serait renvoyé par `next()` (s'il existe) et juste après l'élément qui serait renvoyé par `previous()` (s'il existe).
  - Lève une `IllegalArgumentException` si `e` est null.

➽ Testez la classe `NodeListIterator<E>` avec les classes de test JUnit `ListIteratorTest` et `NodeListIteratorTest`.

5. `NodeSequentialList<E>` dans le package `collections.lists`

La classe *à compléter* `NodeSequentialList<E>` est similaire à class `NodeList<E>` mais hérite de la classe abstraite `AbstractSequentialList<E>` (héritière de la classe `AbstractList<E>`) mais base ses opérations sur un `ListIterator<E>` qui sera fourni par la "factory method" `listIterator()`. Dans notre cas on fournira une instance de `NodeListIterator<E>` nouvellement implémenté.

La figure ci-dessous décrit les relations de la classe `NodeSequentialList<E>`



Les attributs et méthodes à implémenter dans la classe `NodeSequentialList<E>` sont donc les suivants :

- Un `Node<E>` privé représentant la tête de liste.
- Un constructeur par défaut qui crée une liste vide.
- Un constructeur de copie à partir d'une `Collection<E>`

- `public int size()` renvoie le nombre d'éléments de la liste.
- `public ListIterator<E> listIterator(int index)` Renvoie une nouvelle instance d'un `NodeListIterator<E>` à l'index fourni.
  - Lève une `IndexOutOfBoundsException` si l'index est invalide. C'est à dire lorsque `index < 0` ou `index > size()`.
- Les méthodes suivantes implémentées dans la classe `AbstractSequentialList<E>` fonctionnent très bien (et vous pourrez donc les utiliser) mais ne prennent pas en compte l'interdiction d'insérer des éléments nuls dans nos listes. Il faudra donc les réimplémenter ici pour lever des `NullPointerException` si l'élément que l'on cherche à ajouter est null.
  - `public E set(int index, E element)`
  - `public void add(int index, E element)`
- Les méthodes de l'interface `Headed<E>` qui permettent d'accéder au nœud en tête de liste.
  - `public Node<E> getHead()`
  - `public void setHead(Node<E> head)`

➽ Testez la classe `NodeSequentialList<E>` avec les classes de test `ListTest` et `CollectionTest`

Classes et interfaces fournies

`Node<E>`

La classe `Node<E>` vous est fournie dans le package `collections.nodes` et représente les nœuds doublement chaînés d'une liste doublement chaînée que vous pourrez utiliser dans les classes `NodeList<E>` et `NodeSequentialList<E>`. Ce Nœud contient donc :

- une donnée : `E data`.
- une référence au nœud précédent : `Node<E> previous` qui peut être `null` s'il n'y a pas de nœud précédent.
- une référence au nœud suivant : `Node<E> next` qui peut être `null` s'il n'y a pas de nœud suivant.

`NodeIterator<E>`

La classe `NodeIterator<E> implements Iterator<E>` est un l'itérateur que nous avons développée pour les collections utilisant des `Node<E>` lors du dernier TP.

`NodeCollection<E>`

La classe `NodeCollection<E>` est un exemple de `Collection<E>` basée sur des `Node<E>` et utilisant un `NodeIterator<E>` que nous avons développée lors du dernier TP.

`Objects`

La classe `Objects` du package `java.util` peut vous être utile dans la mesure où elle propose un certain nombre d'algorithmes utilitaires sur les valeurs ou les objets tels que :

- `public static int checkIndex(int index, int length)` : Qui vérifie si `index` est dans les limites de la plage de 0 (inclus) à `length` (exclus) et lève une `IndexOutOfBoundsException` si ce n'est pas le cas.
  - L'index `index` est considéré comme hors des bornes de la plage lorsque :
    - `index < 0`
    - `index >= length`
    - `length < 0` qui est implicite d'après les inégalités précédentes.
- `public static <T> T requireNonNull(T obj)` : Qui vérifie que `obj` est non null et lève une `NullPointerException` si ce n'est pas le cas.

- Exemple :

```
public Foo(Bar bar) throws NullPointerException
{
    this.bar = Objects.requireNonNull(bar);
    ⋮
}
```

Classes de test

Vous disposez dans le package `tests` des classes de test JUnit suivantes :

- `NodeTest` : Pour tester la classe `Node<E>`.
- `NodeIteratorTest` : Pour tester la classe `NodeIterator<E>` en dehors de toute collection.
- `CollectionTest` : Pour tester les collections (par exemple `NodeCollection<E>`) ou bien les listes (en tant que collections) (par exemple `NodeList<E>` & `NodeSequentialList<E>`).
- `ListIteratorTest` : Pour tester plusieurs type de `ListIterator<E>` : Ceux fournis par des `ArrayList<E>`, des `LinkedList<E>` ou bien directement `NodeListIterator<E>`.
- `NodeListIteratorTest` : Pour tester spécifiquement la classe `NodeListIterator<E>`.
- ListTest : Pour tester plusieurs type de `List<E>` : Des `ArrayList<E>`, des `LinkedList<E>` et nos listes `NodeList<E>` et `NodeList<E>`.