Comprendre l'API des collections
Corrigé

Exercice 1 : Généricité et sous-typage : cas des classes *1*
Exercice 2 : Généricité et sous-typage : cas des tableaux *2*
Exercice 3 : Afficher une liste *2*
Exercice 4 : Manipuler des collections *4*
Exercice 5 : Comprendre `Collections.binarySearch` *7*
Exercice 6 : Comprendre List et Collections.unmodifiableList *8*
Exercice 7 : range de Python en Java *16*

1 Généricité et sous-typage

Exercice 1 : Généricité et sous-typage : cas des classes
Intéressons nous au lien entre généricité et sous-typage dans le cas d'une interface ou classe.
1.1. Y a-t-il sous-typage entre List<String> et ArrayList<String> ? Si oui, dans quel sens ?
Solution : Oui, `ArrayList<String>` est une sous-type de `List<String>`.
1.2. Y a-t-il sous-typage entre ArrayList<Object> et ArrayList<String> ? Si oui, dans quel sens ?

Solution : Non, il n'y a pas de relation de sous-typage. On pourrait penser (à tord !) que `ArrayList<String>` est une sous-type de `ArrayList<Object>`. Mais ce n'est pas le cas. Nous allons le voir dans la question suivante.
1.3. Que penser du programme suivant ?

```java
import java.util.ArrayList;

public class GenericiteSoustypageExempleClasse {

    public static void main(String[] args) {
        ArrayList<String> ls = new ArrayList<String>();
        ArrayList<Object> lo = ls;
        lo.add("texte");
        lo.add(15.5);
        System.out.println("premier élément  : " + lo.get(0));
        System.out.println("deuxième élément : " + lo.get(1));
    }

}
```

Solution : Il y a une incohérence car on peut ajouter un nombre réel dans une liste de String : lo autorise à ajouter n'importe quel objet, mais l'objet sera ajouté dans la liste associée qui a été créée comme une liste de String. Ceci n'est pas cohérent !

Est-ce qu'il compile ?

Solution : Il y a une erreur de compilation car `ArrayList<String>` n'est pas un compatible avec `ArrayList<Object>`. Ce n'est donc pas un sous-type !

Que donne l'exécution ?

Solution : Erreur de compilation. On ne peut donc pas l'exécuter.

Que peut-on en conclure ?

Solution : Qu'il n'y a pas sous-typage (voir erreur de compilation). Notons que ceci est logique car s'il y avait sous-typage, on arriverait à une incohérence : on pourrait mettre un nombre réel (`Double`) dans une liste de de chaîne (`String`).

Exercice 2 : Généricité et sous-typage : cas des tableaux

Intéressons sous au lien entre généricité et sous-typage dans le cas des tableaux.

2.1. Y a-t-il sous-typage entre un tableau d'objets `Object[]` et un tableau de chaînes `String[]` ? Si oui, dans quel sens ?

Solution : Il ne devrait pas y avoir sous-typage. C'est ce que nous avons vu dans l'exercice précédent. Mais en Java, on a bien sous-typage : `String[]` est un sous-type de `Object[]`.

Ce sont des raisons historiques, d'avant l'introduction de la généricité.

2.2. Est-ce que le programme suivant compile ? Que donne l'exécution ?

```java
import java.util.ArrayList;

public class GenericiteSoustypageExempleTableau {

    public static void main(String[] args) {
        String[] ts = new String[2];
        Object[] to = ts;
        to[0] = "texte";
        to[1] = 15.5;
        System.out.println("premier élément  : " + to[0]);
        System.out.println("deuxième élément : " + to[1]);
    }

}
```

Solution : Il compile. Mais on a une erreur a l'exécution. Le typage de Java n'est pas complet mais ce qui n'a pas pu être vérifié à la compilation l'est à l'exécution. Ici, il y a la levée de l'exception : `java.lang.ArrayStoreException`.

Exercice 3 : Afficher une liste

On souhaite écrire une méthode qui permettra d'afficher une liste, quelque soit le type de ses éléments. Voici le programme qui est proposé.

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class AfficherListe {

```

```
7       /** Afficher les éléments de liste, un par ligne... */
8       public static void afficher(List<Object> liste) {
9           for (Object o : liste) {
10              System.out.println("  - " + o);
11          }
12      }
13
14      public static void main(String[] args) {
15          List<Object> lo = new ArrayList<>();
16          Collections.addAll(lo, "un", "deux", 3);
17          afficher(lo);
18
19          List<String> ls = new ArrayList<>();
20          Collections.addAll(ls, "un", "deux", "trois");
21          afficher(ls);
22      }
23  }
```

3.1. Est-ce que ce programme compile ? Que donne alors son exécution ?
Solution : Le programme ne compile pas et signale une erreur sur la ligne : `afficher(ls);` car `ls` est du type `List<String>` qui n'est pas compatible avec le type du paramètre formel de afficher qui est `List<Object>`. C'est ce que l'on a vu à l'exercice 1.
3.2. Modifier le programme pour qu'il fonctionne. On utilisera la généricité.
Solution : Une solution est d'utiliser la généricité. On définira ainsi une méthode afficher pour une List<E>. La valeur de E sera choisie en fonction du type du paramètre effectif : Object pour lo et String pour ls.

```
1   import java.util.List;
2   import java.util.ArrayList;
3   import java.util.Collections;
4
5   public class AfficherListe {
6
7       /** Afficher les éléments de liste, un par ligne... */
8       public static <E> void afficher(List<E> liste) {
9           for (E o : liste) {
10              System.out.println("  - " + o);
11          }
12      }
13
14      public static void main(String[] args) {
15          List<Object> lo = new ArrayList<>();
16          Collections.addAll(lo, "un", "deux", 3);
17          afficher(lo);
18
19          List<String> ls = new ArrayList<>();
20          Collections.addAll(ls, "un", "deux", "trois");
21          afficher(ls);
22      }
23  }
```

3.3. Est-ce que nommer le paramètre de généricité est nécessaire ?
Solution : Non, car on n'utilise pas vraiment E dans le texte de la mêthode. On n'a pas besoin de connaître sa valeur précise. On peut donc remplacer E par le type joker « `?` ».

Modifier le programme pour utiliser le type joker.
Solution :

```java
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class AfficherListe {

    /** Afficher les éléments de liste, un par ligne... */
    public static void afficher(List<?> liste) {
        for (Object o : liste) {
            System.out.println("  - " + o);
        }
    }

    public static void main(String[] args) {
        List<Object> lo = new ArrayList<>();
        Collections.addAll(lo, "un", "deux", 3);
        afficher(lo);

        List<String> ls = new ArrayList<>();
        Collections.addAll(ls, "un", "deux", "trois");
        afficher(ls);
    }
}
```

Notons que pour le type de la variable du `for`, nous avons utilisé `Object` car c'est la borne supérieure de `E`. Si on avait défini une borne supérieure avec `extends` (par exemple on écrira `List<? extends Dessinable>` si on veut que le paramètre effectif de généricité soit un sous-type de `Dessinable`) on aurait utilisé cette borne (`for (Dessinable o : liste)`).

2 Bornes d'un paramètre de généricité

Exercice 4 : Manipuler des collections
Intéressons nous à deux opérations sur des listes qui s'appuient sur un critère. Un critère fournit une méthode qui s'applique sur un élément et retourne vrai ou faux suivant que le critère est vérifié ou non. La première opération s'appelle `tous`. Elle retourne vrai si et seulement si tous les éléments d'une collection satisfont un critère. La deuxième opération s'appelle `filtrer`. Elle permet d'ajouter dans une liste résultat tous les éléments d'une première liste qui respectent un critère donné. Ces méthodes sont définies comme méthodes de classe dans la classe `Outils`.

Notons qu'il seraient possible de les généraliser pour qu'elles travaillent sur des collections plutôt que des listes.

Remarque : Sous eclipse ou autre IDE, des erreurs seront certainement signalées sur les classes de test. Il ne faut pas chercher à les corriger avant que ce soit demandé dans les exercices.

4.1. Compléter le code de la méthode `tous` de la classe `Outils`. On la testera en utilisant la classe de test `TestOutilsTousString`.

Solution :

```
1       public static <E> boolean tous(
2               List<E> elements,
3               Critere<E> critere)
4       {
5           for (E x : elements) {
6               if (! critere.satisfaitSur(x)) {
7                   return false;
8               }
9           }
10          return true;
11      }
```

Nous avons utilisé un `return` dans une boucle alors que ceci était interdit au premier semestre. Pourquoi ?

Car ici le code est beaucoup plus simple et lisible ainsi. Pour se passer du `return` (ou d'un `break`) il faudrait remplacer le `for` par un `while`, expliciter l'itérateur, gérer un booléen pour savoir si le critère est satisfait ou non. Au total un code beaucoup plus compliqué à écrire... Et à lire !

4.2. La classe `TestOutilsTousNumber` signale des erreurs.

4.2.1. Expliquer ces erreurs.

Solution : Nous n'avons pas de solution pour la valeur du paramètre de généricité E. Le premier paramètre impose E = Integer ou E = Double (suivant les tests) et le deuxième paramètre impose E = Number. Il n'y a donc pas de solution pour donner une valeur à E.

4.2.2. Corriger la méthode `tous` pour réussir les tests de `TestOutilsTousNumber`.

Solution : La contrainte sur le paramètre de généricité de la liste est trop forte. En fait, il suffit que les éléments soient des E ou un sous-type de E. On peut donc prendre un type inconnu (type joker) avec E comme borne supérieure (`List<? extends E>`). Dans ce cas, E sera Number et le `?` sera Integer (ou Double).

Remarque : Il serait possible de prendre un autre paramètre de généricité, F, en plus de E en imposant une contrainte entre ces deux paramètres de généricité. Par exemple :

```
1       public static <E, F extends E> boolean tous(
2               List<F> elements,
3               Critere<E> critere)
```

Il n'est cependant pas nécessaire ici de connaître F et un type joker est donc plus adapté.

Alternativement, on pourrait affaiblir la contrainte sur le paramètre de généricité du critère : ce pourrait être tout type plus général que E. On le noterait donc `Critere<? super E> critere`. Dans ce cas, E sera Integer (ou Double) et le `?` sera Number.

Bien sûr, on pourrait combiner les deux :

```
<E> boolean tous(List<? extends E> elements, Critere<? super E> critere)
```

Dans ce cas, on peut avoir plusieurs solutions possibles pour E. Prenons l'exemple de l'appel `Outils.tous(entiers, nonNul)` avec `entiers` de type `List<Integer>` et `nonNul` de type `Critere<Integer>`. Notons `?2` le premier type joker et `?2` le second. On a alors :

```
?1 = Integer    // imposé par le premier paramètre effectif
?2 = Number     // imposé par le deuxième paramètre effectif
?1 extends E    // voir paramètre de généricité du premier paramètre formel
?2 super E      // voir paramètre de généricité du deuxième paramètre formel
```

donc

```
Integer extends E    // ou Integer <= E
Number super E       // ou Number >= E
```

donc

```
Integer <= E <= Number
```

Il y a donc ici deux solutions : E = Integer ou E = Number.

4.3. Compléter le code de la méthode `filtrer` de la classe `Outils`. On la testera en utilisant la classe de test `TestOutilsFiltrerString`.

Solution :

```
1   public static <E> void filtrer(
2           List<E> source,
3           Critere<E> aGarder,
4           List<E> resultat)
5   {
6       for (E e : source) {
7           if (aGarder.satisfaitSur(e)) {
8               resultat.add(e);
9           }
10      }
11  }
```

4.4. La classe `TestOutilsFiltrerStringObject` signale des erreurs.

4.4.1. Expliquer ces erreurs.

Solution : Nous n'avons pas de solution pour la valeur du paramètre de généricité E. Les deux premiers paramètres imposent E = String et le dernier impose E = Object. E ne peut pas être les deux à la fois. D'où l'erreur de compilation.

4.4.2. Corriger la méthode `filtrer` pour réussir les tests de `TestOutilsFiltrerStringObject`.

Solution : La contrainte sur le paramètre de généricité de la liste résultat est trop forte. En fait, il suffit que les éléments soient des E ou un super-type de E. On peut donc prendre un type inconnu (type joker) avec E comme borne inférieure (`List<? super E>`).

4.5. Tester (et corriger) la méthode `filtrer` avec la classe `TestOutilsFiltrerNumber`.

Solution : Comme pour la méthode `Outils.tous`, il faut diminuer la contrainte sur le premier paramètre en utilisant un type inconnu avec E comme borne supérieure.

On pourrait aussi affaiblir la contrainte du paramètre de généricité du critère en disant que c'est un type quelconque avec E comme borne inférieure.

Remarque : Plutôt que d'utiliser le type joker, on aurait pu donner des noms explicites aux paramètres de généricité (ici 3 noms). Cependant, nous n'avons pas besoin de les connaître précisément. Il est donc préférable ici d'utiliser le type joker plutôt que de choisir des noms qui ne seront pas forcément significatifs. On pourrait prendre E le type des éléments, F, un type fils de E et P un type parent de E. Est-ce plus vraiment lisible ? Pas vraiment, d'autant plus que les contraintes sur les bornes seront données dans la définition des types et pas directement sur les paramètres formels comme avec les types joker).

3 API Collection

Exercice 5 : Comprendre `Collections.binarySearch`
On s'intéresse à la méthode `binarySearch` qui prend deux paramètres, une liste et l'élément cherché. On peut consulter sa documentation.
5.1. Quelles sont les contraintes sur la liste ?
Solution : Les éléments de la liste doivent être triés dans l'ordre croissant.
5.2. S'agit-il de programmation défensive ou offensive ?
Solution : De programmation offensive. La condition à respecter est donnée dans la documentation mais n'est pas vérifiée explicitement puisqu'il n'y a pas d'exception correspondante. C'est bien ce qu'on constatera quand on regardera le code.

Ceci est logique puisque une recherche dichotomique est efficace (en log(n)) alors que vérifier si une liste est triée est linéaire. Vérifier la précondition annulerait tout le bénéfice de la recherche par dichotomie ! Le programmation défensive n'aurait donc pas de sens ici.

En revanche, exprimer la précondition en JML et l'instrumenter pendant la phase de mise au point du programme permettrait de mettre en évidence le non respect de ces préconditions et de localiser plus rapidement les erreurs dans les programmes.
5.3. À quoi correspond RandomAccess ?
Solution : Il s'agit d'une interface de marquage : elle ne spécifie pas de méthode mais correspond à une propriété. Une classe qui est sous-type de l'interface possède la propriété. Une classe qu'y n'en est pas un sous-type ne la possède pas.

La propriété qui correspond à RandomAccess est "est-ce que l'accès au ième élément est efficace ?". C'est par exemple le cas de ArrayList qui est un sous-type de RandomAccess et pas de LinkedList qui n'en est pas un sous-type.
5.4. Expliquer le code de cette méthode `binarySearch`. Voir le source de la classe Collections.
Solution : Cette méthode commence par tester si la liste a un accès efficace au ième élément (via `instanceof` sur `RandomAccess`). Si c'est le cas, c'est une recherche par dichotomie classique qui est faite en utilisant la méthode get de List.

Sinon, la recherche par dichotomie utilise un itérateur de liste pour passer au nouvel élément milieu : en avançant ou en reculant suivant qu'il est après ou avant le milieu précédent. Ceci permet de garantir que l'on parcourra au plus n éléments (contrairement à une utilisation directe

de get qui pourrait conduire à beaucoup de déplacement (suivant ou précédent) si l'élément cherché est vers le milieu [1] de la liste). La recherche reste efficace en terme de comparaison (log(n)).

Notons qu'il y a une deuxième condition : si la taille de la liste est petite (inférieure à `BINARYSEARCH_THRESHOLD`) une recherche classique est faite car le coût de création de l'itérateur ne serait pas rentabilisé. La limite est à 5000 !

Exercice 6 : Comprendre List et Collections.unmodifiableList

Intéressons nous à `Collections.unmodifiableList`.

6.1. *Exemple1*. Compléter le code de la méthode `consulter` de `ExempleUnmodifiableList`. Il s'agit de remplacer les `TODO()` et les `XXX*` par le code attendu. La méthode `exemple1()` doit alors s'exécuter sans erreur.

Si des erreurs sont signalées, il faut comprendre les messages qui sont affichés car ils devraient vous permettre de comprendre le problème et comment le corriger.

Solution : L'objectif était de retrouver les opérations classique des opérations sur les collections. La version complétée de la classe est donnée au listing 1.

L'appel de la méthode `remove` est ici un piège car il existe deux méthodes nommées `remove` sur `List` :

1. `remove(Object o);`
2. `remove(int index);`

C'est la première qui nous intéresse ici mais l'appel avec l'entier fait que c'est la seconde qui est appelée. Les deux signatures fonctionnent mais la première nécessite une conversion. C'est donc la deuxième qui est retenue.

Pour appeler la première, il faut construire un Integer à partir de `entier`. On peut faire `Integer.valueOf(entier)` ou `(Integer) entier`.

Est-ce que la méthode consulter se contente de consulter la liste sans la modifier ?

Solution : Non, puisqu'elle appelle `remove`.

Est-ce que mettre le premier paramètre en **`final`** résoudrait le problème ?

Solution : Non, car **`final`** s'appliquerait ici au paramètre : on ne pourrait pas réaffecter le paramètre mais on pourrait toujours appliquer `remove`. Le mot-clé **`final`** garantit juste, ici, que l'on référencera toujours la même liste.

Ce serait une bonne pratique de mettre **`final`** pour tous les attributs, variables et paramètres que l'on ne change pas...

6.2. Dans la question précédente, on s'est rendu compte que l'on ne pouvait pas faire confiance à la méthode qui dit ne pas modifier la liste. Elle a accès à notre liste et donc à toutes ses opérations, y compris celles de modification.

Proposer une manière pour garantir que la méthode `consulter` ne modifiera pas la liste qu'elle reçoit. Cette solution ne doit pas induire un surcout trop important, en particulier si la liste

1. Dans `LinkedList` la recherche part du début ou de la fin suivant que l'indice de l'élément cherché est avant ou après le milieu. Il s'agit en effet d'une liste avec double chaînage (vers le précédent et vers le suivant).

contient un grand nombre d'éléments, surtout qu'en général, la méthode ne devrait pas modifier la liste (puisqu'elle s'y est engagée). On ne mettra pas en œuvre cette solution. Il faut juste donner le principe.

Solution : Bien sûr mettre le paramètre en `final` ne sert à rien : c'est le paramètre, la poignée qui est constant, pas l'objet associé !

Une autre solution serait de transmettre une copie de la liste. Ainsi, si la liste est modifiée par l'opération utilisée, la liste originale ne sera pas modifiée. Cette solution est coûteuse puisqu'il faut faire une copie de la liste. C'est d'autant plus coûteux qu'en général c'est inutile puisque l'opération ne modifie normalement pas la liste qu'elle a reçue.

La bonne solution ici consiste à fournir à l'opération non pas notre liste, mais une procuration sur cette liste. La procuration (ou proxy) interdira les opérations interdites (les opérations de modifications de la liste) en levant une exception, par exemple UnsupportedOperationException.

Le coût est limité : création d'un nouvel objet proxy mais pas de copie de la liste, et une indirection supplémentaire lors de l'appel des méthodes de la liste. Cette solution permet aussi de savoir simplement si l'opération a essayé de modifier la liste : l'exception se produit.

6.3. *Exemple2.* Dans la méthode `exemple2`, on veut utiliser `Collections.unmodifiableList` pour rendre une liste non modifiable et ainsi répondre à la question 6.2. Compléter le code de exemple2().

Solution : Voir le listing 1.

6.4. *Code source.* Comprenons le fonctionnement de la méthode `unmodifiableList`.

6.4.1. Consulter le code source de la classe Collections de l'API Java 8 dans le fichier `/mnt/n7fs/ens/tp_cregut/src/Collections.java` ou sur `http://cregut.perso.enseeiht.fr/ens/src/Collections.java` pour répondre aux questions qui suivent.

Solution : Il faut retrouver la méthode, lire son code et les éléments qu'elle utilise...

6.4.2. Quel est le patron de conception mis en œuvre ?

Solution : Procuration (ou Proxy).

On le retrouve bien dans le code. La liste réelle est reçue en paramètre du constructeur. Les opérations appellent les opérations correspondantes de la liste réelle sauf si elles modifient la liste. Dans ce cas, une exception est levée.

On remarque qu'il y a deux classes qui réalisent la procuration, dont l'une hérite de l'autre. C'est le point abordé dans la question suivante.

Dans le code de `UnmodifiableList`, on retrouve bien le principe du patron Proxy protection : un attribut qui correspond à la vraie liste (`list`) fournie en paramètre du constructeur. Les méthodes de consultation appellent la méthode correspondante de la vraie liste. Les opérations de modification lèvent l'exception `UnsupportedOperationException`.

```
1  static class UnmodifiableList<E> extends UnmodifiableCollection<E>
2                                    implements List<E> {
3      private static final long serialVersionUID = -283967356065247728L;
4
5      final List<? extends E> list;
6
7      UnmodifiableList(List<? extends E> list) {
```

```
8           super(list);
9           this.list = list;
10      }
11
12      public boolean equals(Object o) {return o == this || list.equals(o);}
13      public int hashCode()           {return list.hashCode();}
14
15      public E get(int index) {return list.get(index);}
16      public E set(int index, E element) {
17          throw new UnsupportedOperationException();
18      }
19      public void add(int index, E element) {
20          throw new UnsupportedOperationException();
21      }
22      public E remove(int index) {
23          throw new UnsupportedOperationException();
24      }
25
26      ...
27
28  }
```

6.4.3. Pourquoi un test est-il fait sur `RandomAccess` ?

Solution : Il faut préserver la propriété `RandomAccess` de la liste passée en paramètre. Ainsi, si la liste initiale est `RandomAccess`, la procuration doit aussi être `RandomAccess`.

Voici la définition de la méthode `Collections.unmodifiableList` :

```
1       /**
2        * Returns an unmodifiable view of the specified list.  This method allows
3        * modules to provide users with "read-only" access to internal
4        * lists.  Query operations on the returned list "read through" to the
5        * specified list, and attempts to modify the returned list, whether
6        * direct or via its iterator, result in an
7        * <tt>UnsupportedOperationException</tt>.<p>
8        *
9        * The returned list will be serializable if the specified list
10       * is serializable. Similarly, the returned list will implement
11       * {@link RandomAccess} if the specified list does.
12       *
13       * @param  <T> the class of the objects in the list
14       * @param  list the list for which an unmodifiable view is to be returned.
15       * @return an unmodifiable view of the specified list.
16       */
17      public static <T> List<T> unmodifiableList(List<? extends T> list) {
18          return (list instanceof RandomAccess ?
19                  new UnmodifiableRandomAccessList<>(list) :
20                  new UnmodifiableList<>(list));
21      }
```

Elle s'appuie sur les proxies `UnmodifiableList` et `UnmodifiableRandomAccessList`.

La classe `UnmodifiableRandomAccessList` est aussi un patron Proxy. Il hérite de `UnmodifiableList` et réalise `RandomAccess`.

`RandomAccess` est une interface de marquage (interface vide) qui modélise la propriété « la liste fournit un accès en temps constant au ième élément ». C'est par exemple le cas de `ArrayList` (qui la réalise) mais pas de `LinkedList` (qui ne la réalise donc pas). Voir le cours sur les Collections.

```
static class UnmodifiableRandomAccessList<E> extends UnmodifiableList<E>
                          implements RandomAccess
{
    UnmodifiableRandomAccessList(List<? extends E> list) {
    super(list);
    }

    public List<E> subList(int fromIndex, int toIndex) {
    return new UnmodifiableRandomAccessList<>(
        list.subList(fromIndex, toIndex));
    }

    private static final long serialVersionUID = -2542308836966382001L;

    /**
    * Allows instances to be deserialized in pre-1.4 JREs (which do
    * not have UnmodifiableRandomAccessList).  UnmodifiableList has
    * a readResolve method that inverts this transformation upon
    * deserialization.
    */
    private Object writeReplace() {
    return new UnmodifiableList<>(list);
    }
}
```

À noter que `subList` est redéfinie car elle doit retourner un objet du type `UnmodifiableRandomAccessList`.

Enfin, la méthode `writeReplace` utilisée pour la sérialisation retourne une `UnmodifiableList` pour des raisons de compatibilité avec les versions de Java qui n'avaient pas encore `UnmodifiableRandomAcces` Lors de la désérialisation l'objet sera testé et s'il est sous-type de `RandomAccess`, le proxy `UnmodifiableRandomAccessList` sera ajouté au-dessus de `UnmodifiableList` pour bien avoir le type `RandomAccess`. Voir le code de `readResolve` ci-après (présent dans `UnmodifiableList`).

```
    /**
     * UnmodifiableRandomAccessList instances are serialized as
     * UnmodifiableList instances to allow them to be deserialized
     * in pre-1.4 JREs (which do not have UnmodifiableRandomAccessList).
     * This method inverts the transformation.  As a beneficial
     * side-effect, it also grafts the RandomAccess marker onto
     * UnmodifiableList instances that were serialized in pre-1.4 JREs.
     *
     * Note: Unfortunately, UnmodifiableRandomAccessList instances
     * serialized in 1.4.1 and deserialized in 1.4 will become
     * UnmodifiableList instances, as this method was missing in 1.4.
     */
    private Object readResolve() {
    return (list instanceof RandomAccess
```

```
15            ? new UnmodifiableRandomAccessList<>(list)
16            : this);
17        }
```

6.5. Est-ce que l'utilisation de `Collections.unmodifiableList` garantit que les objets de la liste ne seront pas modifiés ?

Solution : Non !

On pourra par exemple utiliser la méthode `get` pour récupérer un élément de la liste puis lui appliquer toutes les opérations que sa classe propose, y compris les opérations de modification.

Le proxy créé par `unmodifiableList` garantit que la liste référencera toujours les mêmes éléments (elle ne sera pas modifiée). En revanche, elle ne garantit pas que ces éléments ne seront pas modifiés.

Si on prend une liste d'entier, ils ne seront pas modifiés mais ce n'est pas du fait du proxy, c'est parce que les Integer sont des objets immuables (non modifiables) en Java.

6.6. *Synthèse.* Répondre de manière concise aux questions suivantes :

1. Peut-on déclarer une liste d'entiers en faisant `List<int>` ? Pourquoi ?

   Solution : On ne peut pas car le paramètre de généricité doit nécessairement être un type référence (classe, interface, tableau ou type énuméré), pas un type valeur. Pour ces types, il faut utiliser les classes enveloppes (`Integer` pour **`int`**, `Double` pour **`double`**, etc.).

2. Combien y a t-il de méthodes `remove` sur List ?

   Solution : Il y a deux méthode `remove` :

   ```
   1  boolean remove(Object o); // spécifiée dans Collection.
   2  E remove(int index);      // ajoutée par List (élément repéré par sa position)
   ```

3. Est-ce que `remove` retourne une information ? Si oui, quelle est sa signification ?

   Solution : La première retourne un booléen qui indique si le modification a été modifiée (vrai) ou pas (faux) suite à l'exécution de la méthode.

   La seconde retourne l'élément qui a été supprimé de la liste (qui était avant à l'indice `index`). On sait que l'élément a été supprimé car sinon on aurait eu `IndexOutOfBoundsException`. En revanche, contrairement à la précédente méthode on ne sait pas l'élément qui est supprimé d'où l'intérêt de l'avoir en valeur de retour de la méthode.

4. Peut-on faire `remove("abc")` sur une liste d'entier (`List<Integer>`) ?

   Solution : Oui, car le paramètre de `remove` dans `Collection` est `Object`. On peut donc supprimer n'importe quel objet. Bien sûr, s'il n'est pas du bon type, il n'est pas dans la collection, il ne sera pas supprimé et la méthode retournera faux.

5. Dans le code de `unmodifiableList`, à quoi correspond `RandomAccess` ?

   Solution : Voir ci-avant dans la réponse à la question 6.4.

Solution :

Listing 1 – La classe ExempleUnmodifiableList complétée

```
1  import java.util.*;
2
3  public class ExempleUnmodifiableList {
```

```
4
5     /* Consigne :
6      * - remplacer les XXX* par les expressions qui vont bien
7      * - remplacer les TODO() par les instructions qui vont bien
8      */
9
10    final static char NL = '\n', TAB = '\t';
11
12    /*
13       Le nom de cette méthode laisse penser qu'elle se contente de consulter la liste.
14       Peut-elle quand même la modifier ? Pourquoi ?
15    */
16    public static void consulter(List<Integer> nombres, int entier) {
17      System.out.println(NL + "consulter(" + nombres + ", " + entier + ")");
18      System.out.println("=== J'EN M'ENGAGE À NE PAS MODIFIER LA LISTE ===");
19
20      // Afficher la taille de la liste nombres
21      System.out.println("nombres = " + nombres);
22      int taille = nombres.size();    // La taille de nombres
23
24      System.out.println("taille = " + taille);
25      assert taille >= 0 : "Il faut remplacer le XXX_i :";
26
27      // Afficher la présence de l'élément entier dans la liste nombres
28      boolean est_present = nombres.contains(entier);  // entier est-il présent dans nombres ?
29
30      System.out.println(entier + " dans nombres ? " + (est_present ? "oui" : "non"));
31
32      // Afficher la fréquence de entier dans nombres
33      int frequence = Collections.frequency(nombres, entier);  // fréquence de entier dans nombres ?
34
35      System.out.println("fréquence de " + entier + " dans nombres ? " + frequence);
36      assert frequence >= 0 : "Il faut remplacer le XXX_i :";
37
38      // Supprimer une fois l'élément entier de nombres
39      System.out.println("suppression de " + entier);
40      // nombres.remove(entier); // Attention remove(int indice)
41      // nombres.remove(Integer(entier));  // remove(Integer element) mais Interger(n) est deprecated
42      // nombres.remove(Integer.valueOf(entier));  // remove(Integer element)
43      nombres.remove((Integer) entier);  // remove(Integer element)
44      System.out.println("nombres = " + nombres);
45
46      // Afficher la taille de la liste nombres
47      int nouvelle_taille = nombres.size();  // La taille de nombres
48      System.out.println("taille de nombres = " + nouvelle_taille);
49      assert nouvelle_taille >= 0 : "Il faut remplacer le XXX_i :";
50
51      if (nouvelle_taille < taille) {
52        System.out.println("=== J'AI QUAND MÊME MODIFIÉ LA LISTE ===");
53      }
54
55      // Afficher la présence de l'élément entier dans la liste nombres
56      boolean encore_present = nombres.contains(entier);  // entier est-il présent dans nombres ?
57
58      System.out.println(entier + " dans nombres ? " + (encore_present ? "oui" : "non"));
59      assert  frequence > 1 == encore_present
60          // frequence > 1 <=> encore_present
61        : "frequence et/ou encore_present mal définis !";
62
63      // Afficher la fréquence de entier dans nombres
```

```
64      int nouvelle_frequence = Collections.frequency(nombres, entier);  // fréquence de entier dans nomb
65
66      System.out.println("fréquence de " + entier + " dans nombres ? " + nouvelle_frequence);
67
68      assert nouvelle_frequence >= 0 : "Il faut remplacer le XXX_i :";
69
70      // Faire quelques vérifications
71      assert frequence - 1 <= nouvelle_frequence && nouvelle_frequence <= frequence
72        : "Les fréquences semblent avoir été mal calculées :";
73
74      assert ! est_present || nouvelle_taille < taille
75        : "Élément non supprimé : TODO() non remplacé ?";
76
77      assert nouvelle_frequence != frequence || ! est_present
78            // nouvelle_frequence == frequence => ! est_present
79        : "Est-ce que est_present a été calculé correctement ?";
80
81      assert frequence != 1 || ! encore_present
82            // frequence == 1 ==> ! encore_present
83        : "" + NL + TAB +  "Comment se fait-il que l'élément soit encore là ?"
84         + NL + TAB + "Indications :"
85         + NL + TAB + TAB + "- Chercher 'remove' dans la documentation de List"
86         + NL + TAB + TAB + "- Quel élément a été supprimé de la liste ?"
87         + NL + TAB + TAB + "- Quel est l'indice de l'élémet supprimé ?"
88         ;
89    }
90
91    public static void exemple1() {
92      List<Integer> mesNombres = new ArrayList<>();
93      Collections.addAll(mesNombres, 0, 2, 3, 5, 7);
94
95      List<Integer> autresNombres = new ArrayList<>();
96      Collections.addAll(autresNombres, 0, 2, 3, 5, 7, 0, 0);
97
98      consulter(mesNombres, 4);
99      consulter(mesNombres, 0);
100     consulter(autresNombres, 0);
101     consulter(mesNombres, 2);
102     consulter(mesNombres, 7);
103   }
104
105   public static void exemple2() {
106     List<Integer> mesNombres = new ArrayList<>();
107     Collections.addAll(mesNombres, 0, 2, 3, 5, 7);
108
109     // Maintenant, on veut être sûr que consulter ne modifiera pas la liste
110     // Comment faire ?  On utlisera Collections.unmodifiableList()
111     // Gradons un accès sur les listes initiales (pour pouvoir les modifier
112     // si besoin)
113     List<Integer> mesNombresModifiable = mesNombres;
114
115     // Rendons non modifiable la liste mesNombres (abus de langage)
116     // En fait la liste d'origine est toujours modifiable.  Mais à travers
117     // mesNombres, elle ne le sera pas grâce au proxy créé par
118     // unmodifiableList.
119     mesNombres = Collections.unmodifiableList(mesNombres);
120
121     // Remarque : Maintenant mesNombres autresNombres ne peut plus être
122     // modifiée. Cependant, mesNombresModifiable donnent toujours accès à la
123     // liste initiale, sans protection.  À travers mesNombresModifiable on
```

```
124       // pourra la modifier.
125
126       for (int i : new int[] {4, 0, 2, 7}) {
127         try {
128           consulter(mesNombres, i);
129           throw new AssertionError("La liste est modifiable !");
130         } catch (Exception e) {
131           System.out.println("EXCEPTION : " + e);
132           System.out.println("mesNombres = " + mesNombres + NL);
133         }
134       }
135
136       // Vérifier que les autres opérations de modification de liste ne
137       // peuvent pas être appelées non plus : add, set, etc.
138
139       try {
140         mesNombres.add(7);
141       } catch (Exception e) {  // En fait : UnsupportedOperationException
142         System.out.println("Impossible d'ajouter 7 : " + e);
143         System.out.println("mesNombres = " + mesNombres + NL);
144       }
145
146       try {
147         mesNombres.remove(0);  // remove int
148         System.out.println("mesNombres = " + mesNombres + NL);
149       } catch (Exception e) {  // En fait : UnsupportedOperationException
150         System.out.println("Impossible de supprimer l'élément à l'indice 0 : " + e);
151         System.out.println("mesNombres = " + mesNombres + NL);
152       }
153
154       try {
155         mesNombres.set(0, 5);  // remplacer l'élément à l'indice 0 par 5
156       } catch (Exception e) {  // En fait : UnsupportedOperationException
157         System.out.println("Impossible de supprimer l'élément à l'indice 0 : " + e);
158       }
159
160       System.out.println("mesNombres = " + mesNombres);
161     }
162
163     public static void main(String[] args) {
164       if (! ExempleUnmodifiableList.class.desiredAssertionStatus()) {
165         System.out.println(""
166             + NL + "Il FAUT exécuter ce programme avec l'option -ea : "
167             + NL + TAB + "java -ea ExempleUnmodifiableList"
168             + NL);
169         System.exit(-1);
170       }
171
172       exemple1();
173       exemple2();
174     }
175
176   }
```

4 Itérateurs

Exercice 7 : range de Python en Java
On veut faire l'équivalent du `range` de Python. On se limite à la forme qui prend trois paramètres entiers : début, fin, pas avec un pas positif. Voici un exemple d'utilisation.

```
for n in range(2, 11, 3):
    print(n)
```

Ce programme affiche 2, 5 et 8 : les entiers de 2 (début) inclus à 11 (fin) exclu de 3 en 3 (pas).
7.1. Qu'affichera le programme suivant ?

```
for n in range(5, 12, 4):
    print(n)
```

Solution :

```
5
9
```

7.2. En Java, on peut faire l'équivalent de range. Ainsi, si on définit une méthode de classe `range` sur une classe `Range` qui est dans le paquetage `exercices`, on peut écrire :

```
import static exercices.Range.range;
...
for (int n : range(2, 11, 3)) {
    System.out.println(n);
}
```

7.2.1. Sur quoi peut s'appliquer un `foreach` en Java ?
Solution : Il peut s'appliquer sur un tableau ou un Iterable qui définit la méthode `iterator` qui retourne un Iterator.
Voici ces deux interfaces [2] (sans commentaires).

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
}
```

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

7.2.2. Écrire la classe Java `exercices.Range`.
Solution :

2. L'interface `Iterator` spécifie une autre méthode : **`void`** `remove()`. Cependant, depuis Java8, elle est définie comme méthode par défaut qui lève `UnsupportedOperationException`. Il est donc inutile de la définir si elle doit avoir ce comportement.

```java
package exercices;

import java.util.Iterator;

public class Range implements Iterable<Integer> {
    /*
     * TODO : La classe Range hérite de Object.  Il faudrait donc redéfinir
     * la méthode equals (deux Range sont égaux ssi ils ont même début,
     * même fin et même pas) et la méthode hashCode.
     *
     * TODO : Il faudrait aussi traiter les pas négatifs.
     */

    private int start, end, step;

    protected Range(int start, int end, int step) {
        if (step <= 0) {
            throw new IllegalArgumentException("required: step > 0,"
                    + " found: step = " + step);
        }
        this.start = start;
        this.end = end;
        this.step = step;
    }

    public static Range range(int start, int end, int step) {
        return new Range(start, end, step);
    }

    public static Range range(int start, int end) {
        return range(start, end, 1);
    }

    public static Range range(int end) {
        return range(0, end, 1);
    }


    @Override public Iterator<Integer> iterator() {
        return new RangeIterator();
    }

    private class RangeIterator implements Iterator<Integer> {
        private int next = start;

        @Override public boolean hasNext() {
            return next < end;
        }

        @Override public Integer next() {
            if (! hasNext()) {
                throw new java.util.NoSuchElementException();
            }
```

```
54                int p = next;
55                next += step;
56                return p;
57            }
58        }
59
60    }
```

7.2.3. Reproduire l'exemple du sujet.
Solution :

```
1   package exercices;
2
3   import static exercices.Range.range;
4
5   public class ExempleSujet {
6       public static void main(String[] args) {
7           for (int i : range(2, 11, 3)) {
8               System.out.println(i);
9           }
10      }
11  }
```

7.2.4. Tester votre classe `exercices.Range` avec la classe `exercices.TestRange`.
Solution : Il suffit d'exécuter la classe de test et, éventuellement, corriger la classe Range.
7.2.5. La fonction range de Python peut prendre deux paramètres (dans ce cas le pas vaut 1) ou un seul (il correspond à la fin, le début vaut 0 et le pas vaut 1). Ajouter ces possibiités et les tester avec la classe `exercices.TestRangeParametresOptionnels`.
Solution : Il suffit surcharger la méthode `range`.