TP3 Java Avancé - Slices of bread

Max Ducoudré - INFO2

Exercice 2 - The Slice and The furious

Un slice est une structure de données qui permet de "virtuellement" découper un tableau en gardant des indices de début et de fin (from et to) ainsi qu'un pointeur sur le tableau. Cela évite de recopier tous les éléments du tableau, c'est donc beaucoup plus efficace. Le concept d'array slicing est un concept très classique dans les langages de programmation, même si chaque langage vient souvent avec une implantation différente.

1. On va dans un premier temps créer une interface Slice avec une méthode array qui permet de créer un slice à partir d'un tableau en Java.

```
    String[] array = new String[] { "foo", "bar" };
    Slice<String> slice = Slice.array(array);
```

On créé une interface Slice qui possède deux méthodes d'instance : `size` et `get` et une méthode statique `array` :

```
public interface Slice<E> {

    int size();
    E get(int index);
    public static <E> Slice<E> array(T[] array) {
        Objects.requireNonNull(array);

        return new ArraySlice<E>(array);
    }

    /* [...] */


}
```

La méthode statique array vas retourner une implantation de Slice : un SliceArray qui est une classe interne à l'interface : (L'interface Slice devient scéllée et permet uniquement la classe interne)

```
public sealed interface Slice<E> permits Slice.ArraySlice<E>  {
    /* [...] */

    final class ArraySlice<E> implements Slice<E> {
        private final E[] array;

        private ArraySlice(E[] array) {
```

```
            Objects.requireNonNull(array);
            this.array = array;
        }

        @Override
        public int size() {
            return array.length;
        }

        @Override
        public E get(int index) {
            if(index < 0 || index > size()) throw new
IndexOutOfBoundsException("Index must be between 0 and " +
String.valueOf(size()-1));
            return array[index];
        }
    }
}
```

Ici, la méthode get ne fabrique pas une copie de la valeur récupérée, ce qui permet de la modifier directement depuis le tableau d'origine

2. On souhaite que l'affichage d'un slice affiche les valeurs séparées par des virgules avec un '[' et un ']' comme préfixe et suffixe.

On ajoute une méthode toString à la classe ArraySlice :

```
public sealed interface Slice<E> permits Slice.ArraySlice<E>  {
    /* [...] */

    final class ArraySlice<E> implements Slice<E> {
        @Override
        public String toString() {
            return Arrays.stream(array)
                .map(e -> e == null ? "null" : e.toString())
                .collect(Collectors.joining(", ", "[", "]"));
        }

        /* [...] */
    }
}
```

3. On souhaite ajouter une surcharge à la méthode array qui, en plus de prendre le tableau en paramètre, prend deux indices from et to et montre les éléments du tableau entre from inclus et to exclus.

On ajoute une nouvelle classe interne à l'interface `Slice` `SubArraySlice<E>` puis on implémente les méthodes de `Slice` à l'intérieure. Ensuite, on ajoute une autre méthode `array` qui vas nous permettre d'instancier l'implémentation de `Slice` avec notre classe `SubArraySlice` :

```java
public sealed interface Slice<E> permits Slice.ArraySlice<E>,
Slice.SubArraySlice<E>  {
    /* [...] */


    // Méthode permettant de créer une instance de l'interface Slice avec
SubArraySlice
    public static <E> Slice<E> array(E[] array, int from, int to) {
        Objects.requireNonNull(array);
        return new SubArraySlice<E>(array, from, to);
    }

    // Classe interne SubArraySlice utilisant from/to
    final class SubArraySlice<E> implements Slice<E> {
        private final E[] array;
        private final int from;
        private final int to;

        private SubArraySlice(E[] array, int from, int to) {
            Objects.requireNonNull(array);
            if(from < 0 || from > array.length) throw new
IndexOutOfBoundsException("From is not valid");
            if(to < from || to < 0 || to > array.length) throw new
IndexOutOfBoundsException("To is not valid");

            this.array = array;
            this.from = from;
            this.to = to;
        }

        @Override
        public int size() {
            return to - from;
        }

        @Override
        public E get(int index) {
            if(index < 0 || index >= size()) throw new
IndexOutOfBoundsException("Invalid index");
            return array[from + index];
        }

        @Override
        public String toString() {
            return Arrays.stream(array, from, to)
                .map(e -> e == null ? "null" : e.toString())
                .collect(Collectors.joining(", ", "[", "]"));
        }
    }
}
```

4. On souhaite enfin ajouter une méthode subSlice(from, to) à l'interface Slice qui renvoie un sous-slice restreint aux valeurs entre from inclus et to exclu. Par exemple,

```
String[] array = new String[] { "foo", "bar", "baz", "whizz" };
Slice<String> slice = Slice.array(array);
Slice<String> slice2 = slice.subSlice(1, 3);
```

Bien sûr, cela veut dire implanter la méthode subSlice(from, to) dans les classes ArraySlice et SubArraySlice.

On ajoute la méthode `subSlice` dans l'interface `slice` :

```
public sealed interface Slice<E> permits Slice.ArraySlice<E>,
Slice.SubArraySlice<E> {
    /* [...] */
    Slice<E> subSlice(int from, int to);
    /* [...] */
}
```

Puis on l'implémente dans `ArraySlice` :

```
public sealed interface Slice<E> permits Slice.ArraySlice<E>,
Slice.SubArraySlice<E> {
    /* [...] */

    final class ArraySlice<E> implements Slice<E> {
        /* [...] */
        @Override
        public Slice<E> subSlice(int from, int to) {
            if(from < 0 || from > size()) throw new
IndexOutOfBoundsException("From is not valid");
            if(to < from || to < 0 || to > size()) throw new
IndexOutOfBoundsException("To is not valid");
            return new SubArraySlice<E>(array, this.from + from, this.from + to);
        }
    }
}
```

Enfin, dans `SubArraySlice` :

```
public sealed interface Slice<E> permits Slice.ArraySlice<E>,
Slice.SubArraySlice<E> {
    /* [...] */

    final class SubArraySlice<E> implements Slice<E> {
        /* [...] */
        @Override
```

```
        public Slice<E> subSlice(int from, int to) {
            return new SubArraySlice<E>(array, from, to);
        }
    }
}
```

Exercice 2 - The Slice and The furious

1. Recopier l'interface Slice de l'exercice précédent dans une interface Slice2. Vous pouvez faire un copier-coller de Slice dans même package, votre IDE devrait vous proposer de renommer la copie. Puis supprimer la classe interne SubArraySlice ainsi que la méthode array(array, from, to) car nous allons les réimplanter et commenter la méthode subSlice(from, to) de l'interface, car nous allons la ré-implanter aussi, mais plus tard.

L'interface `Slice2` :

```
public sealed interface Slice2<E> permits Slice2.ArraySlice<E> {

    int size();
    E get(int index);

    public static <E> Slice2<E> array(E[] array) {
        Objects.requireNonNull(array);
        return new ArraySlice<E>(array);
    }

    final class ArraySlice<E> implements Slice2<E> {
        private final E[] array;

        private ArraySlice(E[] array) {
            Objects.requireNonNull(array);
            this.array = array;
        }

        @Override
        public int size() {
            return array.length;
        }

        @Override
        public E get(int index) {
            if(index < 0 || index > size()) throw new
IndexOutOfBoundsException("Index must be between 0 and " +
String.valueOf(size()-1));
            return array[index];
        }

        @Override
        public String toString() {
            return Arrays.stream(array)
                .map(e -> e == null ? "null" : e.toString())
```

```
                .collect(Collectors.joining(", ", "[", "]"));
        }
    }
}
```

2. Déclarer une classe SubArraySlice à l'intérieur de la classe ArraySlice comme une inner class donc pas comme une classe statique et implanter cette classe et la méthode array(array, from, to).

On ajoute la classe `SubArraySlice` dans la classe `ArraySlice` puis on la fait implémenter l'interface `Slice` comme aux questions précédentes. Néamoins, ici, elle n'aura pas besoin de stocker un tableau `E[]` car elle n'est pas statique et peux accéder aux champs de son parent `ArraySlice` :

```
public sealed interface Slice2<E> permits Slice2.ArraySlice<E>,
Slice2.ArraySlice<E>.SubArraySlice {
    /* [...] */

    final class ArraySlice<E> implements Slice2<E> {

        /* [...] */

        public final class SubArraySlice implements Slice2<E> {
            private final int from;
            private final int to;

            private SubArraySlice(int from, int to) {
                Objects.requireNonNull(array);
                if(from < 0 || from > array.length) throw new
IndexOutOfBoundsException("From is not valid");
                if(to < from || to < 0 || to > array.length) throw new
IndexOutOfBoundsException("To is not valid");
                this.from = from;
                this.to = to;
            }

            @Override
            public int size() {
                return to - from;
            }

            @Override
            public E get(int index) {
                if(index < 0 || index >= size()) throw new
IndexOutOfBoundsException("Invalid index");
                return array[from+index];
            }

            @Override
            public String toString() {
                return Arrays.stream(array, from, to)
                    .map(e -> e == null ? "null" : e.toString())
```

```
                    .collect(Collectors.joining(", ", "[", "]"));
            }
        }
    }
}
```

On ajoute également une nouvelle méthode static `array` pour utiliser cette nouvelle implémentation :

```
public sealed interface Slice2<E> permits Slice2.ArraySlice<E>,
Slice2.ArraySlice<E>.SubArraySlice {
    /* [...] */
    public static <E> Slice2<E> array(E[] array, int from, int to) {
        Objects.requireNonNull(array);
        return new ArraySlice<E>(array).new SubArraySlice(from, to);
    }

}
```

En effet, pour instancier un une classe interne (`SubArraySlice`), il faut d'abord instancier la classe parent (`ArraySlice`).

3. Dé-commenter la méthode subSlice(from, to) de l'interface et fournissez une implantation de cette méthode dans les classes ArraySlice et SubArraySlice.

On ajoute la méthode `subSlice` dans l'interface `Slice2` :

```
public sealed interface Slice2<E> permits Slice2.ArraySlice<E>,
Slice2.ArraySlice<E>.SubArraySlice {
    /* [...] */
    Slice2<E> subSlice(int from, int to);
    /* [...] */
}
```

Puis on l'implémente dans `ArraySlice` en créant une nouvelle instance de `SubArraySlice` puis on l'impémente dans `SubArraySlice` en utilisant l'array du parent :

```
public sealed interface Slice2<E> permits Slice2.ArraySlice<E>,
Slice2.ArraySlice<E>.SubArraySlice {
    /* [...] */

    // Méthode pour une meilleure lisibilité du code dans les préconditions
    public static void checkFromTo(int from, int to, int size) {
        if(from < 0 || from > size) throw new IndexOutOfBoundsException("'from'
value ("+from+") must be > 0 and < " + String.valueOf(size));
        if(to < from || to < 0 || to > size) throw new
IndexOutOfBoundsException("'to' value ("+to+") mist be >= 0 and < " +
String.valueOf(size));
```

```
    }


    final class ArraySlice<E> implements Slice2<E> {
        /* [...] */
        @Override
        public Slice2<E> subSlice(int from, int to) {
            Slice2.checkFromTo(from, to, size());
            return new SubArraySlice(from, to);

        }

        public final class SubArraySlice implements Slice2<E> {
            /* [...] */
            @Override
            public Slice2<E> subSlice(int from, int to) {
                Slice2.checkFromTo(from, to, size());
                return new SubArraySlice(this.from + from, this.from + to);
            }
        }
    }
}
```

4. Dans quel cas va-t-on utiliser une inner class plutôt qu'une classe interne ? *On préfère utiliser une inner class plutôt qu'une classe interne quand on a besoin d'accéder aux champs de la classe parente.*

Exercice 4 - The Slice and The Furious: Tokyo Drift

1. Recopier l'interface Slice du premier exercice dans une interface Slice3. Supprimer la classe interne SubArraySlice ainsi que la méthode array(array, from, to) car nous allons les réimplanter et commenter la méthode subSlice(from, to) de l'interface, car nous allons la réimplanter plus tard. Puis déplacer la classe ArraySlice à l'intérieur de la méthode array(array) et transformer celle-ci en classe anonyme.

On garde uniquement la méthode statique `array` qui renvoie une classe anonyme implémentant `Slice3` :

```
public interface Slice3<E> {

    int size();
    E get(int index);

    public static <E> Slice3<E> array(E[] array) {
        Objects.requireNonNull(array);
        return new Slice3<E>() {
            @Override
            public int size() {
                return array.length;
            }
            @Override
            public E get(int index) {
                if(index < 0 || index > size()) throw new
IndexOutOfBoundsException("Index must be between 0 and " +
```

```
String.valueOf(size()-1));
                return array[index];
            }
            @Override
            public String toString() {
                return Arrays.stream(array)
                    .map(e -> e == null ? "null" : e.toString())
                    .collect(Collectors.joining(", ", "[", "]"));
            }
        };
    }
}
```

2. On va maintenant chercher à implanter la méthode subSlice(from, to) directement dans l'interface Slice3. Ainsi, l'implantation sera partagée. Écrire la méthode subSlice(from, to) en utilisant là encore une classe anonyme.

On ajoute la méthode `subSlice` dans l'interface `Slice3` puis une méthode `array(array, form, to)` qui créé une classe anonyme. Enfin, on implémente `subSlice` dans nos deux méthodes anonymes

```
public interface Slice3<E> {
    /* [...] */

    Slice3<E> subSlice(int form, int to);

    public static <E> Slice3<E> array(E[] array, int from, int to) {
        var sliceArray = Slice3.array(array);
        return new Slice3<E>() {

            @Override
            public int size() {
                return to - from;
            }

            @Override
            public E get(int index) {
                if(index < 0 || index >= size()) throw new
IndexOutOfBoundsException("Invalid index");
                return array[from+index];
            }

            @Override
            public String toString() {
                return Arrays.stream(array, from, to)
                    .map(e -> e == null ? "null" : e.toString())
                    .collect(Collectors.joining(", ", "[", "]"));
            }

            @Override
            public Slice<E> subSlice(int fromParam, int toParam) {
                Slice3.checkFromTo(fromParam, toParam, size());
```

```
                    return Slice3.array(array, from + fromParam, from + toParam);
                }
            }
        }

        public static <E> Slice3<E> array(E[] array) {
            Objects.requireNonNull(array);
            return new Slice3<E>() {
                /* [...] */

                @Override
                public Slice3<E> subSlice(int form, int to) {
                    Slice3.checkFromTo(form, to, array.length);
                    return Slice3.array(array, form, to);
                }
            };
        }
    }
```

3. Dans quel cas va-t-on utiliser une classe anonyme plutôt qu'une classe interne ? *Utiliser une classe anonyme permet d'utiliser l'équivalent d'une interface fonctionnelle avec plusieurs méthodes.*