TP8 Java Avancé - Query (fonctionnelle)

Max Ducoudré - INFO2

Exercice 2 - Query

1. implantation possible. Ce n'est pas très beau comme design, mais cela fait un seul fichier ce qui est plus pratique pour la correction. L'interface Query doit posséder une méthode fromList qui permet de créer une Query comme expliqué ci-dessus. De plus, il doit être possible d'afficher les éléments d'une Query avec la méthode toString() qui effectue le calcul des éléments et les affiche. L'affichage contient tous les éléments présents (ceux pour qui la fonction prise en second paramètre renvoie un élément présent) séparés par le symbole " |> ". Attention : il ne faut pas faire le calcul des éléments (savoir si ils sont présent ou non) à la création du Query, mais uniquement lorsque l'affichage est demandé. Écrire le fichier Query.java avec l'interface et la classe d'implantation. Vérifier que les tests unitaires "Q1" passent.

On écrit une interface scéllée `Query<E>` qui permet uniquement `QueryImpl`. `QueryImpl<E, T>` prend 2 types paramétrées : E est le type des données qui est stocké et T est le type des données qui sera renvoyé avec l'interface. On stocke également une fonction depuis le constructeur qui permet de passer d'un type à un autre.

```
public sealed interface Query<E> {

    public static <E, T> Query<T> fromList(
            List<E> list,
            Function<? super E, ? extends Optional<? extends T>> mapper) {

        Objects.requireNonNull(list);
        Objects.requireNonNull(mapper);

        return new QueryImpl<E, T>(list, mapper);
    }

    final class QueryImpl<E, T> implements Query<T> {
        private final List<E> elements;
        private final Function<? super E, ? extends Optional<? extends T>> mapper;

        private QueryImpl(List<E> elements, Function<? super E, ? extends
Optional<? extends T>> mapper) {
            Objects.requireNonNull(elements);
            Objects.requireNonNull(mapper);
            this.elements = Collections.unmodifiableList(elements);
            this.mapper = mapper;
        }


        @Override
        public String toString() {
            return Arrays.stream(elements)
                    .mapper(e -> mapper.apply(e).isPresent())
                    .map(e -> mapper.apply(e).get().toString())
```

```
                    .collect(Collectors.joining(" |> "));
        }
    }
}
```

Le constructeur est en privé car on souhaite que `QueryImpl` soit uniquement appellée depuis une méthode `from` de l'interface. On remarque également que le calcul du type de retour de l'interface est fait qu'au moment au `toString`.

2. On souhaite ajouter une méthode toList à l'interface Query dont le but est de renvoyer dans une liste non-modifiable les éléments présents. Écrire la méthode toList.

La méthode toList vas renvoyer une liste non-mutable des éléments de du Query. La méthode vas donc utiliser la fonction permettant de transformer les types E (stockés) vers T. Pour cela, il faut penser a retirer les Optionals vides.

```
public sealed interface Query<E> {

    public List<E> toList();

    /* [...] */

    final class QueryImpl<E, T> implements Query<T> {

        /* [...] */

        @Override
        public List<T> toList() {
            var list = new ArrayList<T>();
            elements.forEach(e -> mapper.apply(e).ifPresent(list::add));
            return Collections.unmodifiableList(list);
        }
    }
}
```

3. On souhaite maintenant ajouter une méthode toStream qui renvoie un Stream des éléments présents dans une Query. Note : ici, on ne vous demande pas de créer un Spliterator, il existe déjà une méthode stream() sur l'interface List. Écrire la méthode toStream.

```
public sealed interface Query<E> {

    public Stream<E> toStream();

    /* [...] */

    final class QueryImpl<E, T> implements Query<T> {

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

```
        @Override
        public Stream<T> toStream() {
            return elements.stream().flatMap(e -> mapper.apply(e).stream());

        }
    }
}
```

4. On souhaite ajouter une méthode toLazyList qui renvoie une liste non-modifiable dont les éléments sont calculés dans une liste modifiable sous-jacente uniquement si on demande la taille et/ou les éléments de la liste. Note : il existe une classe java.util.AbstractList qui peut vous servir de base pour implanter la liste paresseuse demandée. Attention : vous veillerez à ne pas demander plusieurs fois si un même élément est présent, une seule fois devrait suffire. Écrire la méthode toLazyList.

La méthode `toLazyList` renvoie une AbstractList qui utilisera un cache et un iterator pour calculer les valeur au fur et à mesure des appels à `get` et `size` :

```
public sealed interface Query<E> {

    public List<E> toLazyList();

    /* [...] */

    final class QueryImpl<E, T> implements Query<T> {

        /* [...] */

        @Override
        public List<T> toLazyList() {
            return new AbstractList<T>() {
                private final Iterator<E> iterator = elements.iterator();
                private final List<T> list = new ArrayList<>();

                @Override
                public T get(int index) {
                    if (index >= list.size()) {
                        while (iterator.hasNext()) {
                            mapper.apply(iterator.next())
                                .ifPresent(list::add);
                            if(index < list.size()) {
                                break;
                            }
                        }
                    }

                    Objects.checkIndex(index, list.size());
                    return list.get(index);
                }

                @Override
```

```
                public int size() {
                    while(iterator.hasNext()) {
                        mapper.apply(iterator.next())
                        .ifPresent(list::add);
                    }

                    return list.size();
                }
            };
        }
    }
}
```

5. On souhaite pouvoir créer une Query en utilisant une nouvelle méthode fromIterable qui prend un Iterable en paramètre. Dans ce cas, tous les éléments de l'Iterable sont considérés comme présents. Note : une java.util.List est un Iterable et Iterable possède une méthode spliterator(). Écrire la méthode fromIterable et modifier le code des méthodes existantes si nécessaire.

Au lieu de stocker une `List` dans `QueryImpl`, on vas stocker un `Iterable` et utiliser son splititerator pour générer les streams :

```
public sealed interface Query<E> {

    /* [...] */
    public static <E> Query<E> fromIterable(Iterable<? extends E> iterable) {
        Objects.requireNonNull(iterable);
        var list = new ArrayList<E>();
        return new QueryImpl<E, E>(iterable,(e) -> Optional.of(e));
    }


    final class QueryImpl<E, T> implements Query<T> {
        private final Iterable<? extends E> elements;

        /* [...] */

        private QueryImpl(Iterable<? extends E> elements, Function<? super E, ?
extends Optional<? extends T>> mapper) {
            Objects.requireNonNull(elements);
            Objects.requireNonNull(mapper);
            this.elements = elements;
            this.mapper = mapper;
        }


        @Override
        public Stream<T> toStream() {
            return StreamSupport.stream(elements.spliterator(), false)
                    .flatMap(e -> mapper.apply(e).stream());
        }
```

```
        @Override
        public String toString() {
            return toStream()
                    .map(e -> e.toString())
                    .collect(Collectors.joining(" |> "));
        }

        @Override
        public List<T> toList() {
            return toStream().toList();
        }
    }
}
```

6. On souhaite écrire une méthode filter qui permet de sélectionner uniquement les éléments pour lesquels un appel à la fonction prise en paramètre de filter renvoie vrai.

On ajoute un champs `filter` correspondant à une liste de `Predicate` à appliquer sur les éléments lors du calcul de `Query`. On modifie également la méthde `toStream` pour filtrer les éléments qui correspondent à tous les filtres dans la liste de `Predicate`

```
public sealed interface Query<E> {

    /* [...] */

    public Query<E> filter(Predicate<? super E> filter);


    final class QueryImpl<E, T> implements Query<T> {

        /* [...] */

        private final List<Predicate<? super T>> filters = new ArrayList<>();

        @Override
        public Query<T> filter(Predicate<? super T> filter) {
            filters.add(filter);
            return this;
        }

        @Override
        public Stream<T> toStream() {
            return StreamSupport.stream(elements.spliterator(), false)
                .flatMap(e -> {
                    var optional = mapper.apply(e);
                    if(optional.isPresent() && filters.stream().allMatch(f ->
f.test(optional.get()))) {
                            return optional.stream();
                    }
                    return Stream.<T>empty();
```

```
                });
        }

    }

}
```

8. On souhaite écrire une méthode map qui renvoie une Query telle que chaque élément est obtenu en appelant la fonction prise en paramètre de la méthode map sur un élément d'une Query d'origine. Écrire la méthode map.

La méthode map vas prendre un type nouveau type paramétré `R` et renvoyer une nouvelle QueryImpl avec ce nouveau type.

```
public sealed interface Query<E> {

    /* [...] */

    public <R> Query<R> map(Function<? super E, ? extends R> function);

    final class QueryImpl<E, T> implements Query<T> {
        /* [...] */

        @Override
        public <R> Query<R> map(Function<? super T, ? extends R> function) {
            return new QueryImpl<E, R>(elements, mapper.andThen(o ->
o.map(function)));
        }
    }
}
```

9. Enfin, on souhaite écrire une méthode reduce sur une Query qui marche de la même façon que la méthode reduce à trois paramètres sur un Stream et sachant que comme notre Query n'a pas d'implantation parallel, le troisième paramètre est superflu. Écrire la méthode reduce. Note: quelle early preview feature peut-on utiliser ici ?

```
public sealed interface Query<E> {

    /* [...] */

    public <U> U reduce(U identity, BiFunction<U, ? super E, U> accumulator);

    final class QueryImpl<E, T> implements Query<T> {
        /* [...] */

        @Override
        public <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator) {
            Objects.requireNonNull(accumulator);
            var result = identity;
```

```
            for(var e : toList()) {
                 result = accumulator.apply(result, e);
            }
            return result;
        }
    }
}
```