TP2 Java Avancé - Sed, the stream editor

Max Ducoudré - INFO2

Exercice 2 - Astra inclinant, sed non obligant

1. On va dans un premier temps définir une interface Rule qui va représenter une règle. Une règle prend en entrée une ligne (une String) et renvoie soit une nouvelle ligne soit rien (on peut supprimer une ligne). À l'intérieur de la classe StreamEditor, créer l'interface Rule avec sa méthode rewrite. Rappeler comment on indique, en Java, qu'une méthode peut renvoyer quelque chose ou rien ?

En java, pour qu'une méthode puisse renvoyer quelque chose ou rien, on la lui fait renvoyer un objet de type Optionnal

On créer la classe StreamEditor avec l'interface fonctionelle Rule comme il suit :

```
public final class StreamEditor {
    @FunctionalInterface // L'annotation @FunctionalInterface permet de vérifier
que l'interface ne contient qu'une seule méthode abstraite
    interface Rule {
        Optional<String> rewrite(String value); // -> UnaryOperator
    }
}
```

2. Avant de créer, dans StreamEditor, la méthode rewrite qui prend deux fichiers, on va créer une méthode rewrite intermédiaire qui travaille sur des flux de caractères. On souhaite écrire une méthode rewrite(reader, writer) qui prend en paramètre un BufferedReader (qui possède une méthode readLine()) ainsi qu'un Writer qui possède la méthode write(String). Comment doit-on gérer l'IOException ?

On ajoute un constructeur à la classe StreamEditor pour stocker la règle. Puis on ajoute la méthode `rewrite` qui vas lire le `reader` par ligne et appliquer la règle dessus via une lambda puis la stocker dans le `writer`. Si la règle ne renvoie rien (une optional vide) alors on écrit rien dans le `writer` :

```
public final class StreamEditor {

    @FunctionalInterface
    interface Rule {
        Optional<String> rewrite(String value);
    }

    private final Rule rule;

    public StreamEditor(Rule rule) {
        Objects.requireNonNull(rule);
        this.rule = rule;
```

```
    }

    public void rewrite(BufferedReader reader, Writer writer) throws IOException {
        Objects.requireNonNull(reader);
        Objects.requireNonNull(writer);

        for(var line = reader.readLine(); line != null; line = reader.readLine())
{
            var optional = rule.rewrite(line);
            if(optional.isPresent()) writer.write(optional.get() + "\n") ;
        }
    }
}
```

Ici, la méthode peut renvoyer une IOException, elle doit donc être appellée autour d'un `try/catch`.

3. On souhaite créer la méthode rewrite(input, output) qui prend deux fichiers (pour être exact, deux chemins vers les fichiers) en paramètre et applique la règle sur les lignes du fichier input et écrit le résultat dans le fichier output. Comment faire en sorte que les fichiers ouverts soit correctement fermés ? Comment doit-on gérer l'IOException ? Écrire la méthode rewrite(input, output).

Pour s'assurer que les fichiers soient correctement fermés, on appelle les méthodes qui les ouvre dans un `try`. Ce qui certifie qu'à la fin du bloc, la ressource soit désalouée. Pour gérer l'IOException, l'appelle à la méthode doit être dans un `try/catch`. Voci la méthode `rewrite` qui prend des `Path` en argument:

```
public final class StreamEditor {
    /* [...] */
    public void rewrite(Path inputPath, Path outputPath) throws IOException {
        Objects.requireNonNull(inputPath);
        Objects.requireNonNull(outputPath);

        try(var writer = Files.newBufferedWriter(outputPath)) {
            try(var reader = Files.newBufferedReader(inputPath)) {
                rewrite(reader, writer); // On appelle la méthode rewrite
précédente
            }
        }
    }
}
```

4. On va écrire la méthode createRules qui prend en paramètre une chaîne de caractères et qui construit la règle correspondante. Pour l'instant, on va considérer qu'une règle est spécifiée par un seul caractère :

- "s" veut dire strip (supprimer les espaces),
- "u" veut dire uppercase (mettre en majuscules),
- "l" veut dire lowercase (mettre en minuscules) et
- "d" veut dire delete (supprimer).

On créer la méthode `createRules` qui prend une `String` en argument et renvoie une `Rule` appliquant la bonne règle via les méthodes déjàs implémentés de `String`

```
public final class StreamEditor {
    /* [...] */
    public static Rule createRules(String rules) {
        Objects.requireNonNull(rules);
        switch(rules) {
            case("s") :
                return s -> Optional.of(s.replaceAll(" ", ""));
            case("u") :
                return s -> Optional.of(s.toUpperCase(Locale.ROOT));
            case("l") :
                return s -> Optional.of(s.toLowerCase(Locale.ROOT));
            case("d") :
                return s -> Optional.empty();
            default :
                throw new IllegalArgumentException("This rule doesnt exist");
        }
    }
}
```

5. On veut pouvoir composer les règles, par exemple, on veut que "sl" strip les espaces puis mette le résultat en minuscules. Pour cela, dans un premier temps, on va écrire une méthode statique andThen dans Rule, qui prend en paramètre deux règles et renvoie une nouvelle règle qui applique la première règle puis applique la seconde règle sur le résultat de la première.

On ajoute la méthode static `andThen` à l'interface `Rule`. Cette méthode vas renvoyer un `Rule` qui applique la première règle sur la `String` puis, qui, avec `flatMap`, applique la seconde règle si l'optional renvoyée par la première n'est pas vide.

```
public final class StreamEditor {
    /* [...] */
    @FunctionalInterface
    interface Rule {
        Optional<String> rewrite(String value);

        static Rule andThen(Rule first, Rule second) {
            Objects.requireNonNull(first);
            Objects.requireNonNull(second);
            return s -> first.rewrite(s).flatMap(second::rewrite);
        }
    }
}
```

On modifie ensuite la méthode `createRules` pour qu'elle puisse prendre en argument plusieurs règles. On utilise pour cela la méthode `andThen` de l'interface `Rule`.

```
public final class StreamEditor {
    /* [...] */
    public static Rule createRules(String rules) {
        Objects.requireNonNull(rules);
        var rule = Rule::rewrite;
        for(var c : rules.split("")) {
            rule = Rule.andThen(rule, createRule(c));
        }
        return rule;
    }
}
```

On créé une méthode privée, qui, pour 1 caractère donné renvoie une Rule :

```
/* [...] */
private static Rule createRule(char rule) {
    switch(rule) {
        case('s') : return s -> Optional.of(s.replaceAll(" ", ""));
        case('u') : return s -> Optional.of(s.toUpperCase(Locale.ROOT));
        case('l') : return s -> Optional.of(s.toLowerCase(Locale.ROOT));
        case('d') : return s -> Optional.empty();
        default : throw new IllegalArgumentException("This rule doesn't exist");
    }
}
/* [...] */
```

On modifie ensuite la méthode createRule en ajoutant une règle pour chaque caractère de la chaîne donnée en argument :

```
/* [...] */
public static Rule createRules(String rules) {
    Objects.requireNonNull(rules);
    Rule result = Optional::of; // On initialise la règle avec une règle qui ne
modifie pas la String

    for (char rule: rules.toCharArray()) {
        result = Rule.andThen(result, createRule(rule));
    }

    return result;
}
/* [...] */
```

6. En fait, déclarer andThen en tant que méthode statique n'est pas très "objet" ... En orienté objet, on préfèrerait écrire rule1.andThen(rule2) plutôt que Rule.andThen(rule1, rule2). On va donc implanter une nouvelle méthode andThen dans Rule, cette fois-ci comme une méthode d'instance. Écrire la méthode d'instance andThen dans Rule et modifier createRules pour utiliser cette nouvelle méthode.

On ajoute la méthode `andThen` à l'interface `Rule` avec l'ancienne méthode en utilisant `this` et la nouvelle règle :

```
    public default Rule andThen(Rule other) {
        Objects.requireNonNull(other);
        return Rule.andThen(this, other);
    }
```

On modifie ensuite la méthode `createRules` pour qu'elle utilise la nouvelle méthode `andThen` :

```
    public static Rule createRules(String rules) {
        Objects.requireNonNull(rules);
        Rule result = Optional::of;

        for (char rule: rules.toCharArray()) {
            result = result.andThen(createRule(rule));
        }

        return result;
    }
```

7. On souhaite implanter la règle qui correspond au if, par exemple, "i=foo;u", qui veut dire si la ligne courante est égal à foo (le texte entre le '=' et le ';') alors, on met en majuscules sinon on recopie la ligne. Avant de modifier createRules(), on va créer, dans Rule, une méthode statique guard(function, rule) qui prend en paramètre une fonction et une règle et crée une règle qui est appliquée à la ligne courante si la fonction renvoie vrai pour cette ligne. Autrement dit, on veut pouvoir créer une règle qui s'applique uniquement aux lignes pour lesquelles la fonction renvoie vrai. Quelle interface fonctionnelle correspond à une fonction qui prend une String et renvoie un boolean ? Écrire la méthode statique guard(function, rule).

L'interface fonctionnelle qui correspond à une fonction qui prend une String et renvoie un boolean est `Predicate<String>` On ajoute la méthode `guard` à l'interface `Rule` qui applique une `Rule` si le `Predicate<String>` est validée :

```
@FunctionalInterface
interface Rule {
    Optional<String> rewrite(String value);

        /* [...] */

    static Rule guard(Predicate<String> predicate, Rule rule) {
        Objects.requireNonNull(predicate);
        Objects.requireNonNull(rule);
        return s -> predicate.test(s) ? rule.rewrite(s) : Optional.of(s);
    }
}
```

On modifie ensuite la méthode `createRules` pour qu'elle utilise la nouvelle méthode `guard` :

```
    // Méthode pour ajouter des règles à une règle existante à partir d'une chaine
de caractère
    private static Rule addRules(Rule providerRule, String rules) {
        var result = providerRule;
        for (char rule: rules.toCharArray()) {
            result = result.andThen(createRule(rule));
        }
        return result;
    }

    // On utilise une expression régulière pour séparer les règles conditionnelles
des règles normales
    public static Rule createRules(String rules) {
      Objects.requireNonNull(rules);
      Rule result = Optional::of;

        var pattern = Pattern.compile("(.*)i=(.*);(.*)");
        var matcher = pattern.matcher(rules);

        if(matcher.matches()) {
            // Règles normales
            var normalRules = matcher.group(1);
            result = StreamEditor.addRules(result, normalRules);

            // Règle conditionelles
            var text = matcher.group(2);
            var conditionalRules = matcher.group(3);

            result = Rule.guard(s -> s.equals(text),
createRules(conditionalRules));

        } else {
            result = StreamEditor.addRules(result, rules);
        }
        return result;
    }
```

8. On souhaite que le test du if puisse être non seulement une String mais aussi une expression régulière.

On modifie la méthode `createRule` en modifiant la lambda qu'on passe dans `guard` en cherchant si sont paramètre `s` match avec l'expression régulière :

```
public static Rule createRules(String rules) {
      Objects.requireNonNull(rules);
      Rule result = Optional::of;

        var pattern = Pattern.compile("(.*)i=(.*);(.*)");
```

```
        var matcher = pattern.matcher(rules);


        if(matcher.matches()) {
            var previousRules = matcher.group(1);
            var text = matcher.group(2);
            var conditionalRules = matcher.group(3);

            result = StreamEditor.addRules(result, previousRules);
            result = Rule.guard(s -> Pattern.matches(text,
s),createRules(conditionalRules));

        } else {
            result = StreamEditor.addRules(result, rules);
        }
        return result;
    }
```