TP1 : Liste, table de hachage, entrées/sorties, stream, lambdas

Exercice 1 - Path, Stream et try-with-resources

1. On représente un chemin par la classe **java.nio.file.Path** car la classe **java.nio.File** contient beaucoup de méthodes qui ne capturent pas d'exceptions.

2. ```
   var path = Path.of("./movies.txt");
   ```

3. ```
   var path = Path.of("./movies.txt");
   try {
       lines = Files.lines(path);
       var numberLines = lines.count();
   } catch (Exception e) {
       System.out.println("Error");
       return ;
   }
   ```

4. ```
   var path = Path.of("./movies.txt");
   try {
       var lines = Files.lines(path);
       var numberLines = lines.count();
       lines.close();
   } catch (Exception e) {
       System.out.println("Error");
       return;
   }
   ```

5. ```
   var path = Path.of("./movies.txt");
   Stream<String> lines = null;
   try {
       lines = Files.lines(path);
       var numberLines = lines.count();
   } finally {
       lines.close();
   }
   ```

6. ```
   var path = Path.of("./movies.txt");
   var lines = Files.lines(path);
   try {
       var numberLines = lines.count();
   } finally {
       lines.close();
   }
   ```

7. **throws** permet de spécifier que la méthode peut lever une exception, cela nous évite de faire plusieurs **try-catch**.
   ```

```java
public static void main(String[] args) throws IOException {
   var path = Path.of("./movies.txt");
   var lines = Files.lines(path);
   try {
      var numberLines = lines.count();
   } finally {
      lines.close();
   }
}
```

8. ````java public static void main(String[] args) throws IOException {

```java
try (var lines = Files.lines(path)) {
    var numberLines = lines.count();
}
```

}

9. En utilisant **try-with-ressources**, on est sur de close les descripteurs de fichier à la fin.

Exercice 2 - Movie Stars

1. ```java
   public record Movie(String title, List<String> actors) {
       public Movie {
           Objects.requireNonNull(title);
           Objects.requireNonNull(actors);
           actors = List.copyOf(actors);
       }
   }
   ```

2. ```java
    public static List<Movie> movies(Path path) throws IOException {
        try (var lines = Files.lines(path)) {
            return lines.map(line -> {
                var tokens = line.split(";");
                var title = tokens[0];
                var actors = Arrays.stream(tokens).skip(1).toList();
                return new Movie(title, actors);
            }).toList();
        }
    }
   ```

3. On choisis le Collector **toUnmodifiableMap()**

```java
public static Map<String, Movie> movieMap(List<Movie> movies) {
    return movies.stream()
        .collect(Collectors.toUnmodifiableMap(Movie::title, movie -> movie));
}
```

4. On peut remplacer le **movie -> movie par Function.identity()**.

```java
public static Map<String, Movie> movieMap(List<Movie> movies) {
    return movies.stream()
```

```
        .collect(Collectors.toUnmodifiableMap(Movie::title,
Function.identity()));
 }
```

5.
```
 public static void numberOfUniqueActors(List<Movie> movies) {
     return movies.stream()
         .flatMap(movie -> movie.actors().stream())
         .limit(20)
         .forEach(actor -> System.out.println(actor));
}
```

6.
```
 public static long numberOfUniqueActors(List<Movie> movies) {
     return movies.stream()
         .flatMap(movie -> movie.actors().stream())
         .count();
 }
```

7. On peut utiliser l'inteface **Set** pour éviter les doublons.

```
public static long numberOfUniqueActors(List<Movie> movies) {
     return movies.stream()
         .flatMap(movie -> movie.actors().stream())
         .collector(Collectors.toSet())
         .count();
 }
```

8.
```
 public static long numberOfUniqueActors(List<Movie> movies) {
     return movies.stream()
         .flatMap(movie -> movie.actors().stream())
         .distinct()
         .count();
 }
```

9. Le type de retour de **numberOfMoviesByActor** est **Map<String, Long>**. collect est une opération terminale, elle fonctionne avec un **Collector**.

```
 public static Map<String, Long> numberOfMoviesByActor(List<Movie> movies) {
      return movies.stream()
          .flatMap(movie -> movie.actors().stream())
          .collect(Collectors.groupingBy(
              Function.identity(),
              Collectors.counting()
          ));
 }
```

10. Le type de retour doit etre un **Optional** au cas ou la structure de données passé en paramètre est vide.

```
public record ActorMovieCount(String actor, long movieCount) {
  public ActorMovieCount {
     Objects.requireNonNull(actor);
  }
}
```

```
public static Optional<ActorMovieCount> actorInMostMovies(Map<String, Long>
numberOfMoviesByActor) {
    return numberOfMoviesByActor.entrySet().stream()
        .collect(Collectors.maxBy(Comparator.comparing(Map.Entry::getValue)))
        .map(elem -> new ActorMovieCount(elem.getKey(), elem.getValue()));
}
```

Steve Chen M1-Info 09/10/2021