Cahier de TP
« Groovy»

Pré-requis :

- Bonne connexion Internet
- JDK11+
- IDE (Eclipse de préférence)
-

Outils recommandés :

- Docker
- Git

Atelier 1 : Mise en place

Installation Groovy

Choisir une installation en fonction de votre OS. (recommandé SDK)

Tester l'installation en testant les différents outils

- Exécuteur de script
- CLI interactif
- Console
- Compilateur

Visualiser les différences entre une compilation ***javac*** et ***groovyc*** avec le dés-assembleur ***javap***

groovyc HelloWorld.groovy
javap HelloWorld.class

Mise en place IDE

Installer le plugins Groovy pour l'IDE de votre choix

Par exemple, pour Eclipse et Java 11 voir :

https://github.com/groovy/groovy-eclipse/wiki

Test de l'IDE

- Création de projet *Groovy*
- Editeur et complétion
- Exécution *Run As Groovy Script, Java Application*

Atelier 2 : Types de base et collections

Reprendre le script *Indexer.groovy*

String, Maps, Regexp

1. Écrire un script qui à partir d'un texte donné, remplit une map avec comme clé les mots du texte et comme valeur leur nombre d'occurrences.

2. Améliorer le programme pour ne pas prendre en compte les mots de 2 caractères ou moins

3. Trier la map par les mots-clés

4. Construire la map inversé : la clé étant le nombre d'occurrence, les valeurs les listes de mots

5. Créer un intervalle avec les valeurs possibles des occurrences. Boucler sur l'intervalle pour afficher les mots clés de cette occurrence

Atelier 3 : POO - Classes, Scripts , traits

3.1 Organisation en classes et packages

Nous réorganisons la logique du TP précédent avec des classes, des packages et un script principal.

Le projet contiendra donc :

- Une classe ***Index*** (package *model*) qui encapsule les données d'un index :
  - Une source (contenu source associé à l'index)
  - Dates de création et d'indexation
  - Une map contenant les mots et leur nombre d'occurrence
- Une classe ***Indexer*** (package *service*) qui permettra de déclencher l'indexation d'un index. Avec comme propriétés :
  - Un *tokenizer* : le délimiteur de mots-clé
  - *minimalSize* ; La taille minimale des mots à mettre dans l'index

  Avec comme méthode
  - *Index buildIndex(Index index)* qui déclenche l'indexation
- Une classe ***MainScript*** utilisant les classes précédentes

3.2 Traits

Définir un trait *Persistent* définissant une méthode boolean save() implémenté comme suit :

```
boolean save() {
      println "Saving ${this.dump()}"
      true
}
```

Utilisé l'index comme un objet persistent dans la classe script, essayer une implémentation dynamique du trait

Atelier 4 : Closure, contexte et délégué

4.1 Utilisation des closures

Nous transformons notre classe *Indexer*.

Un indexeur est désormais modélisé comme :

- Un *Tokenizer* responsable de découper un texte en un ensemble de mots
- Une liste de filtres : Chaque filtre traite une collection de mots et retourne une autre collection de mots. Les filtres sont donc traités comme des closures prenant en entrée une liste de *String* et retournant une autre liste de *String*

Transformer la classe *Indexer* dans ce sens et réécrire la méthode qui renseigne un index.

Dans la classe Main, créer un Index et lui ajouter 3 filtres :

- Un filtre transformants la liste de mots en liste de mots en minuscule
- Un filtre supprimant les mots inférieurs à 3 caractères
- Un filtre supprimant les noms propres Delhi, Londres, Pékin, Chine

4.2 Classe déléguée

Afficher la classe propriétaire et la classe délégué de la Closure

Modifier la logique afin que les filtres puissent mettre à jour une variable dans l'indexer. Utiliser l'opérateur Elvis pour se protéger des NPE

Par exemple, on mettra à jour un tableau stockant les temps d'exécutions des Closures.

Atelier 5 : Programmation Dynamique

5.1 Utilisation d'une méthode de hook

Nous voulons modifier la classe Index afin qu'elle puisse répondre à des méthodes de type *hasTermAndAnotherTerm*

Les règles étant :

- Les méthodes sont préfixées par *has*, suivi d'un terme arbitraire suivi éventuellement du mot clé *And* et un autre terme
- Le nombre de termes n'est pas limité
- Les méthodes renvoient true si l'index contient tous les termes compris dans le nom de la méthode

Pour ce faire on utilisera la méthode de hook *methodMissing*

5.2 Implémenter une méthode de GroovyObject

Toujours dans la classe *Index*, Implémenter la méthode *getProperty(String propertyName)* de telle sorte que si la propriété n'existe pas la méthode *hasPropertyName* est appelée

5.3 Intercepteurs et ProxyMetaClass

- Créer une classe implémentant *Interceptor*.
- Cette classe devra chaîner les 2 intercepteurs fournis par Groovy : *TracingInterceptor* et *BenchmarkInterceptor*.
- Appliquer cet intercepteur à la classe *Indexer* via un *ProxyMetaClass*
- Afficher les informations de ces intercepteurs

5.4 Meta-classe

Ajouter dynamiquement une méthode *export* à l'index qui retourne une String au format CSV de la map interne

5.5 Category

Utiliser une classe annotée pour effectuer la méthode *export* précédente

Atelier 6 : Annotations

6.1 @Builder

Utiliser l'annotation *@Builder* sur la classe ***Indexer*** et modifier dans le script principal la façon de créer l'instance *d'Indexer*

6.2 Trace

Remplacer les *println* de la classe *Indexer* par un logger
Utiliser plusieurs niveau de log et configurer.

Exemple configuration ***logging.properties*** pour *java.util.logging* :

```
handlers= java.util.logging.ConsoleHandler
.level= FINE
java.util.logging.ConsoleHandler.level = FINE
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter
```

6.3 Annotations de génération

Sur la classe ***Index*** utiliser les annotations *@ToString* et *@EqualsHashCode*

- *@ToString* : affiche les champs *map,created* et *indexed*
- *@EqualsHashcode* : Se base sur les champs *source* et *map*

Tester

6.4 Design et scripting

Faire en sorte que la méthode d'indexation *d'Indexer* soit cachée

Faire en sorte que le script principal utilise l'indexer comme instance déléguée

Atelier 7 : GDK, librairies cœur

7.1 I/O

Modifier le code précédent afin de parcourir un répertoire à la recherche de fichiers *.txt* et de créer des index pour chaque fichier

7.2 Thread

Exécuter chaque travail d'indexation d'un fichier dans une thread séparé
Attendre la fin des threads pour afficher les indexs construits

7.2 XML

Construire un représentation XML de l'ensemble des indexs construits.
Nécessite ***groovy-xml.jar***

Atelier 8 : GDK, SQL

8.1 Script d'initialisation

- Choisissez une base de votre choix.
- Créer une base *formation*
- Récupérer le driver associé

Utiliser soit le mécanise de Grab, soit le classpath de votre IDE pour charger le Driver et créer une instance de groovy.sql.Sql

Écrire un script d'initialisation qui crée une table qui permettra de stocker les données d'une classe Index :

- id
- les dates de création et d'indexation
- Map des mots au format XML
- Map des mots au format JSON

8.2 Couche DAO

Créer une classe abstraite DAOAbstract qui

- définit 3 m éthodes abstraites :
  - `List getFields()`
  - `String getTablename()`
  - `String getIdField()`
- et qui implémente les fonctionnalités CRUD :
  - Insertion d'un enregistrement via un dataSet
  - Lecture, Mise à jour de champs, suppression via un id
  - Retourner toutes les lignes d'un data set

Créer ensuite la classe IndexDAO héritant de DAOAbstract

Tester dans le script principal :

- L'insertion des indexes
- L'affichage de tous les enregistrements de la base
- La suppression des enregistrements 1 par 1

Atelier 9 : Json et Groovlet

9.1 Sérialisation Json (Optionnel)

Sérialiser la map des index et stocker le résultat dans le champ jsonMap en base

9.2 Groovlet

Récupérer le projet Gradle fourni

Effectuer :
./gradlew jettyRun

Cela doit démarre un serveur Jetty sur le port 8080, accéder à la page d'accueil

Inspecter le code fourni, en particulier :

- Le fichier ***build.gradle*** (C'est du groovy!)
- Le code présent dans ***src/main/groovy*** : Il contient les classes des ateliers précédents
- Les exemples de groovlets dans ***src/main/webapp***

Compléter les groovlets ***getAll.groovy*** et ***post.groovy*** qui doivent respectivement :

- Renvoyer un tableau JSON qui contient tous les index stockés en base
- Qui permet d'indexer un texte et de créer une entrée en base

Atelier 10 : Tests unitaires

Récupérer les sources fournis

Travailler dans un nouveau package ***org.formation.test***

10.1 GroovyAssert

Écrire une classe de test unitaire simple qui utilise les méthodes statiques de *GroovyAssert* suivantes :

- *assertEquals*
- *notYetImplemeneted*
- *fail*

Écrire une classe de test *IndexTest* testant la classe Index, en particulier :

- Les méthodes *hasTermAndTerm*
- Les accès aux propriétés

10.2 StubFor et MockFor

Écrire une classe de Test pour *org.formation.service.IndexService* qui implémentera 2 méthodes permettant de simuler ***indexDao*** :

- 1 utilisant ***StubFor***
- l'autre ***MockFor***

Faire apparaître les différences de fonctionnement

Atelier 11 : Integration

Reprendre le projet Gradle fourni

11.1 Eval

Écrire une classe Java principale (dans *src/main/java*) qui évalue comme expression Groovy le premier de ses arguments et l'affiche sur la console

Modifier le fichier ***build.gradle*** et la tâche ***fatJar*** afin qu'il construise un exécutable de votre classe.

./gradlew fatJar

Tester

11.2 GroovyShell

Écrire un*e* classe Java principale qui exécute le script ***MainScript.groovy*** fourni en lui passant 2 paramètres :

- Le répertoire racine de scan
- Une expression régulière spécifiant les fichiers à indexer

Atelier 12 : DSL

En suivant l'exemple du cours, mettez au point les classes Groovy nécessaire afin que le fichier texte avec le DSL suivant :

```
move left
move right, by: 3.meters
move right, by: 3.m, at: 5.km/h
```

Affiche sur la console :

```
robot moved left by 1m at 1 km/h
robot moved right by 3m at 1 km/h
robot moved right by 3m at 5 km/h
```

Procéder par étapes