SCIENCES SORBONNE UNIVERSITÉ

SRCS : Systèmes Répartis Client/Serveur

TME 2 : Interpréteur interactif de commandes

Objectifs pédagogiques

- I/O Java
- construction d'objets avec l'API de réflexivité
- sérialisation d'objet

Objectifs

L'objectif de ce TME est de programmer un interpréteur interactif de ligne de commande (à l'instar d'un shell) en utilisant l'API standard des Entrées/Sorties et de la réflexivité. Notre outil va donc recevoir des commandes sous la forme `cmd arg1 arg2 ... argN` et l'exécuter ou éventuellement envoyer une erreur (ex : commande inexistante, arguments erronés, etc.). Le TME est composé de trois exercices incrémentaux, chaque exercice améliore la version de l'interpréteur développé à l'exercice précédent. Dans le premier exercice, nous allons programmer une première version de l'interpréteur qui déploiera et offrira deux commandes internes (`cat` et `echo`). Dans le deuxième exercice, l'interpréteur offrira deux nouvelles commandes internes, `deploy` et `undeploy`, qui permettra respectivement à l'utilisateur de déployer ses propres commandes ou de les supprimer. Enfin, le troisième exercice offrira une commande interne `save` qui permettra de sauvegarder dans un fichier la configuration actuelle de l'interpréteur (i.e., l'ensemble des commandes qui y sont déployées) et de pouvoir construire un interpréteur à partir d'une sauvegarde. Dans le cadre de ce TME, le package le travail sera `srcs.interpretor`.

Informations complémentaires

En java, pour manipuler le système de fichiers (créer/supprimer un fichier, tester l'existence d'un fichier, etc.) de la machine hôte, le jdk offre la classe `java.io.File` et la classe utilitaire `java.nio.file.Files`. Vous aurez certainement besoin de ces outils pour faire ce TME, n'hésitez-donc pas à consulter la javadoc.

Exercice 1 – Interpréteur et commandes de base

Pour programmer les différentes commandes de l'interpréteur, nous allons appliquer le Design Pattern *Command* qui permettra d'être beaucoup plus extensible et modulaire. Nous allons donc définir une interface java `Command` qui offre une méthode `void execute(PrintStream out);`. Chaque commande que l'on programmera dans ce TME sera donc matérialisée par une classe qui implante cette interface. Les arguments de la commande seront passés au constructeur de la classe d'implantation sous la forme d'une liste de chaînes de caractères. Il faudra supposer que toute classe d'implantation offre donc un tel constructeur (impossible de le spécifier dans une interface). La liste contiendra tous les mots de la ligne de commande y compris le nom de la commande (qui sera identifié par l'indice 0). En cas d'erreur sur les arguments (nombre, type), le constructeur jettera une exception `IllegalArgumentException`. L'argument `out` passé en paramètre est un flux de sortie permettant à la commande d'émettre des données de retour à l'utilisateur (exemple : le ou les résultat(s) de la commande).

Question 1

Dans le package de travail, définir une interface `Command` ;

Question 2

Programmer les deux classes suivantes qui implantent l'interface `Command` :

- `Echo` : qui émet la liste de ses arguments sur son flux de sortie (sauf l'argument d'indice 0 qui indique le nom de la commande)
- `Cat` : qui prend en argument un chemin de fichier et qui émet sur son flux de sortie le contenu de ce fichier. Une `IllegalArgumentException` est jetée par le constructeur si aucun paramètre n'est donné ou si le paramètre n'existe pas ou bien si le paramètre ne désigne pas un fichier régulier.

Nous allons maintenant implanter la classe `CommandInterpretor` qui sera la classe métier de l'interpréteur. A minima, cette classe :

- possède un attribut de type `Map<String,Class<? extends Command>>` qui permet d'associer un nom de commande à sa classe d'implantation. Ceci permet de connaître les commandes reconnues par l'interpréteur.
- un constructeur par défaut (sans paramètre) qui instancie l'attribut et qui y configure les commandes internes *cat* (associée à la classe `Cat`) et *echo* (associée à la classe `Echo`)
- une méthode publique d'instance `getClassOf` qui pour un nom de commande donné, renvoie la classe correspondante. Si la commande n'existe pas, la référence `null` est renvoyée.
- une méthode publique `perform` sans résultat, qui prend deux arguments : une ligne de commande (`String`) et un flux de sortie, et qui peut jeter n'importe quelle exception (`throws Exception`). Cette méthode :

  1. sépare chaque mot de la ligne de commande pour produire une liste de mots. Il peut y avoir plusieurs espaces entre chaque mot, il est préférable d'utiliser `StringTokenizer` (voir javadoc) plutôt que la méthode `split` de la classe `String`.
  2. teste si le premier mot correspond bien à un nom de commande existante. Dans le cas où la commande n'existerait pas, une `CommandNotFoundException` (classe fournie dans les ressources du TME) est jetée.
  3. instancie une commande avec la classe correspondante et appelle la commande `execute` en passant en paramètre le flux de sortie renseigné en argument.

Question 3

Implanter la classe `CommandInterpretor`

Question 4

Tester le code produit dans cet exercice avec la classe de test JUnit

`srcs.interpretor.test.CommandInterpretorTest1`.

Question 5

Dans les méthodes de test `testCommandCatError1` et `testCommandCatError2` de la classe de test `srcs.interpretor.test.CommandInterpretorTest1`, nous testons si l'interpréteur renvoie bien une exception à une commande qui serait mal paramétrée. On peut remarquer que l'exception attendue est une `InvocationTargetException` et non une `IllegalArgumentException` qui pourtant est bien ce qui est censé être jeté par le constructeur. Pourquoi ?

Exercice 2 – Déployer/supprimer des commandes dynamiquement

Nous souhaitons à présent avoir la possibilité d'ajouter ou de retirer dynamiquement des commandes de l'interpréteur. Pour cela nous allons programmer les commandes suivantes :

- "*deploy*" qui permet d'ajouter une nouvelle commande dans l'interpréteur. Elle prend trois paramètres :
  - ◇ le nom de la nouvelle commande
  - ◇ le chemin du répertoire racine (ou du jar) dont l'arborescence contient le .class de la commande
  - ◇ le nom absolu de la classe de la commande.

  Une exception `IllegalArgumentException` sera jetée :
  - ◇ si le chemin n'existe pas ou n'est pas un dossier
  - ◇ s'il existe déjà une commande du même nom dans l'interpréteur
  - ◇ si le chargement de la classe se passe mal
- "*undeploy*" qui permet de supprimer une commande de l'interpréteur. Elle prend un seul argument qui est le nom de la commande à supprimer.

Vous implanterez ces commandes dans des classes membres interne à la classe `CommandInterpretor`. Ceci permet de pouvoir accéder à la map qui est attribut interne de l'interpréteur tout en conservant l'encapsulation.

Question 1

Pourquoi il est impossible d'utiliser la méthode `Class.forName` sur le troisième argument de la commande "*deploy*" ?

Question 2

Programmer la commande `deploy`. Vous utiliserez la classe `URLClassLoader` qui permet de charger des classes à partir d'un ensemble d'URL (passé en paramètre du constructeur). Pour traduire un `File` en `URL` java, il faut faire :

```
1 URL url = new File(name_fichier).toURI().toURL();
```

Question 3

Programmer la commande `undeploy`.

Question 4

Ajouter dans le constructeur par défaut de l'interpréteur les commandes internes `deploy` et `undeploy` dans la map qui associe nom de commande et classe des commandes. Modifier en conséquence la méthode `perform` pour pouvoir instancier des objets de commandes associées à une classe interne. Pour différencier de manière générique ces classes des autres, vous testerez si la commande représente une classe interne de la classe `CommandInterpretor`.

Question 5

Tester le code produit dans cet exercice avec la classe de test JUnit

`srcs.interpretor.test.CommandInterpretorTest2`.

Ce test lance une compilation via la commande javac. Il est necessaire de s'assurer que la version de java que vous utilisez dans votre IDE est compatible avec le jdk pointé par la commande javac.

Exercice 3 – Interpréteur persistant

Dans cet exercice nous souhaitons pouvoir rendre l'état de l'interpréteur persistant, c'est-à-dire pouvoir sauvegarder sa map de commandes dans un fichier d'une part et pouvoir instancier un interpréteur à partir d'un fichier. Dans la classe `CommandInterpretor`, nous allons programmer :

- une commande "*save*" (classe interne `Save` membre de `CommandInterpretor`) qui sauvegarde la map de commandes dans un fichier dont le nom est passé en paramètre. On utilisera la classe `ObjectOutputStream`.

- un constructeur qui prend un nom de fichier en paramètre, qui appelle le constructeur par défaut et qui complète la map à partir des informations sauvegardées dans le fichier. On utilisera la classe `ObjectInputStream`

Question 1

Quel problème allons nous rencontrer dans le chargement pour les commandes externes qui ont été déployées dynamiquement avant la sauvegarde ?

Question 2

Pour corriger le problème de la question précédente, vous aurez besoin en partie de spécifier un `ClassLoader` autre que celui par défaut à l'`ObjectInputStream` qui lit la map dans le fichier. Pour ce faire il faut définir une classe fille de `ObjectInputStream` et l'utiliser pour lire le fichier. Cette classe fille doit redéfinir la méthode

```
1 protected Class<?> resolveClass(final ObjectStreamClass objectStreamClass)
```

qui est appelée suite à un appel à `readObject` lorsque l'on souhaite obtenir l'objet `Class` de l'objet en cours de lecture dont la description est en paramètre.

Question 3

Tester le code produit dans cet exercice avec la classe de test JUnit

`srcs.interpretor.test.CommandInterpretorTest3`.