SCIENCES SORBONNE UNIVERSITÉ

SRCS : Systèmes Répartis Client/Serveur

TME 2 : Interpréteur interactif de commandes

Objectifs pédagogiques :

- **I/O Java en mode caractères**
- **construction avec l'API reflexive**
- **sérialisation d'objet**
- **annotations**

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 de java comme les Entrées/Sorties, la réflexivité et les annotations. 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.). Dans ce TME, il est supposé que vous utilisez l'IDE Eclipse.

Exercice 1 – Interpréteur et commandes de base

Question 1

Écrire une classe `Interpreteur` qui possède une méthode `void run()` (que nous allons compléter ensuite). Définir une méthode `static void main` (dans une classe dédiée ou dans la classe `Interpreteur`) qui instancie un interpréteur et qui appelle `run`

Question 2

Nous allons commencer par programmer l'interaction entre l'utilisateur et l'interpréteur via le clavier et la console. Dans un premier temps, compléter la méthode `run()` pour que chaque ligne saisie sur l'entrée standard soit affichée sur la sortie standard (conseil : utiliser un `BufferedReader`). Le programme s'arrêtera lorsque l'on fera un Ctrl+D.

Question 3

Modifier la méthode `run` :

- pour ignorer les lignes vides
- pour extraire et afficher uniquement le premier mot de la ligne (le nom de la commande).

Nous allons à présent coder les différentes commandes de notre interpréteur en suivant le design pattern *Command*. 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`.

Question 4

Définir une interface `Command` qui offre la méthode `void execute()`.

Question 5

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

- `Echo` : qui affiche sur la sortie standard ses arguments (sauf l'argument d'indice 0)
- `Exit` : qui prend un argument optionnel de type entier, affiche le message `"Fin"` et arrête la JVM avec `System.exit(int value)`

Question 6

Dans l'interpréteur ajouter en attribut une map qui associe une `String` à une `Class` de `Command`. Ceci permet de connaître les commandes reconnues par l'interpréteur et de faire le lien entre le nom de la commande saisie au clavier et la classe java de commande correspondante.

Question 7

Dans le constructeur de l'interpréteur, déclarer dans la map les commandes `exit` et `echo` qui seront associées respectivement à la classe`Exit` et `Echo`.

Question 8

Modifier la méthode `run` pour qu'elle puisse instancier la bonne classe de commande en fonction de la ligne saisie et fait appel à exécute si aucune exception n'a été jetée. Si la commande saisie est inconnue, une exception `IllegalArgumentException` est jetée. Dans tous les cas une exception n'est pas fatale, on se contentera d'afficher un message d'erreur et de redonner la main à l'utilisateur (les seules façon d'arrêter l'interpréteur est soit de faire un Ctrl+D, soit de taper la commande `exit` définie précédemment)

Question 9

Tester les commandes `echo` et `exit`.

Exercice 2 – Déployer/supprimer des commandes dynamiquement

Nous souhaitons à présent à voir 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
  - ◇ si 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.

Dans tous les cas, on ne changera pas la signature de `execute()` avec une clause `throws`

Question 1

Pourquoi il est impossible d'utiliser la méthode `Class.forName` sur le troisième argument ?

Question 2

Pourquoi est-il très utile de déclarer ces classes en interne dans la classe `Interpreteur` ?

Question 3

Programmer la commande `deploy`. Vous utiliserez la classe `URLClassLoader` dont le constructeur prend en paramètre un tableau d'objets typés `URL`. Pour traduire un chemin en URL java, il faut faire :

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

Question 4

Programmer la commande `undeploy`.

Question 5

Ajouter dans le constructeur de l'interpréteur les commandes `deploy` et `undeploy` dans la map qui associe nom de commande et classe des commandes. Modifier en conséquence la méthode `run` 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 `Interpreteur`.

Question 6

Créer un deuxième projet Eclipse, configurer son build path pour qu'il dépende des sources du projet de l'interpréteur (on a besoin de l'interface `Command`) et y copier la classe suivante :

```
public class Add implements Command {

 private final int a;
 private final int b;

 public Add(List<String> args) {
   if(args.size() < 3) {
     throw new IllegalArgumentException("usage add : <operande1> <operande2>");
   }
   this.a=Integer.parseInt(args.get(1));
   this.b=Integer.parseInt(args.get(2));
 }


 @Override
 public void execute() {
   System.out.println(a+b);
 }

}
```

Question 7

Tester les commandes `deploy` et `undeploy` en déployant, en essayant puis en supprimant la commande `add` ci-dessus.

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 commande dans un fichier d'une part et pouvoir instancier un interpréteur à partir d'un fichier.

Question 1

Programmer :

- une commande `save` (classe interne `Save`) 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`

Dans un premier temps on supposera que toutes les classes de commandes de la map se situe dans le projet Eclipse de l'interpréteur (cas des commandes `deploy`, `undeploy`, `exit`,`echo` et `save`).

Question 2

Pour tester la commande `save` :

- supprimer la ligne de code du constructeur par défaut qui déclare la commande `echo`.
- lancer l'interpréteur
- déployer la commande `echo`
- faire appel à la commande `save`
- taper la commande `exit` (fin de la JVM)
- relancer l'interpréteur (nouvelle JVM)
- vérifier que la commande `echo` est disponible sans refaire un `deploy`

Question 3

Refaire la même chose mais utiliser la commande `add` au lieu de la commande `echo`. Quel est le problème ?

Question 4

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

```
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 5

Définir une annotation de classe `DontSave` qui permet de marquer une classe commande pour empêcher la sauvegarde lors de l'appel à la commande `save`. Cette annotation possèdera un atribut `String reason` (avec une valeur par défaut égale à `unknown`) qui permettra au programmeur de la commande de préciser pourquoi il ne faut pas la sauvegarder. Modifier le code de la commande `save` en conséquence pour qu'elle ne sauvegarde pas une commande tagué `DontSave` et qu'elle affiche sur la sortie standard la raison

Question 6

Tester votre annotation sur la commande `add`