Scriptable debuggers avec Java Debug Interface

Steven Costiou
EVREF, Centre Inria de l'Université de Lille
steven.costiou@inria.fr

Février 2025

Nous voulons désormais contrôler notre debugger avec un langage de script. Vous devez implémenter un mécanisme de contrôle par ligne de commande.

Implémentez les commandes suivantes, que vous devez pouvoir interpréter à partir de la ligne de commande. À chaque commande doit correspondre une méthode qui sera exécutée lorsque la commande est récupérée à partir d'une entrée utilisateur. Certaines commandes retournent un objet représentant le résultat de son exécution, avec une visualisation de la donnée demandée (lorsque c'est le cas). Par exemple, la commande **receiver** retourne une référence à l'objet exécutant la méthode et l'imprime également sur la console. L'idée est de pouvoir manipuler les objets retournés par certaines commandes de base pour implémenter d'autres commandes plus complexes. Il peut également être nécessaire de concevoir des modèles intermédiaires pour stocker les informations retournée par l'exécution des commandes.

Contrainte : Évitez de gérer ces commandes via des conditionnelles (switch ou if). Le patron de conception *Commande* est une manière efficace et facilement extensible d'implémenter simplement des commandes `https://en.wikipedia.org/wiki/Command_pattern`.

Contrainte : Vous devez au maximum garder le contrôle sur votre debugger. Cela signifie que vous ne laissez jamais le debugger avancer seul dans l'exécution (sauf si vous lui en donnez l'ordre). À chaque commande exécutée, vous devez pouvoir construire les éléments qui peuvent être demandées via l'API de votre langage de script.

1. **step** : execute la prochaine instruction. S'il s'agit d'un appel de méthode, l'exécution entre dans cette dernière.
2. **step-over** : execute la ligne courante.
3. **continue** : continue l'exécution jusqu'au prochain point d'arrêt. La granularité est l'instruction step.
4. **frame** : renvoie et imprime la frame courante.
5. **temporaries** : renvoie et imprime la liste des variables temporaires de la frame courante, sous la forme de couples nom $\rightarrow$ valeur.

6. **stack** : renvoie la pile d'appel de méthodes qui a amené l'exécution au point courant.
7. **receiver** : renvoie le receveur de la méthode courante (this).
8. **sender** : renvoie l'objet qui a appelé la méthode courante.
9. **receiver-variables** : renvoie et imprime la liste des variables d'instance du receveur courant, sous la forme d'un couple nom $\rightarrow$ valeur .
10. **method** : renvoie et imprime la méthode en cours d'exécution.
11. **arguments** : renvoie et imprime la liste des arguments de la méthode en cours d'exécution, sous la forme d'un couple nom $\rightarrow$ valeur.
12. **print-var(String varName)** : imprime la valeur de la variable passée en paramètre.
13. **break(String filename, int lineNumber)** : installe un point d'arrêt à la ligne lineNumber du fichier fileName.
14. **breakpoints** : liste les points d'arrêts actifs et leurs location dans le code.
15. **break-once(String filename, int lineNumber)** : installe un point d'arrêt à la ligne lineNumber du fichier fileName. Ce point d'arrêt se désinstalle après avoir été atteint.
16. **break-on-count(String filename, int lineNumber, int count)** : installe un point d'arrêt à la ligne lineNumber du fichier fileName. Ce point d'arrêt ne s'active qu'après avoir été atteint un certain nombre de fois count.
17. **break-before-method-call(String methodName)** : configure l'exécution pour s'arrêter au tout début de l'exécution de la méthode methodName.