Implementation de debuggers avec Java Debug Interface

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

Février 2025

Nous voulons implémenter un debugger simplifié controllable manuellement via une interface textuelle en utilisant Java Debug Interface. Ce TP se base sur l'utilisation de l'environnement de développement IntelliJ mais peut être réalisé avec Eclipse.

Organisation du TP. Le TP est divisé en deux parties. La première partie est fortement guidée vers la mise en place des éléments de base du debugger. La seconde partie constitue la partie notée que vous devrez rendre. Vous ne pouvez pas réaliser la seconde partie sans avoir réalisé la première, il faut donc vous appliquer, **la lire et la réaliser rigoureusement**.
Conseil : concevez, implémentez et testez pas-à-pas.

Rendu attendu. Par groupe de 2 personnes au plus, sous la forme d'un dépôt git auquel vous me donnerez accès au plus tard le vendredi 7 février 2025 :

> **Le code source** commenté de votre projet.
> Rendu sur un dépôt git de votre choix, privé, sur invitation. (PS : ne m'expliquez pas la partie 1 du TP...).

Documentation externe. Tous les détails de mise en oeuvre d'un debugger avec JDI ne sont pas décrits dans ce TP. La conception du TP repose fortement sur la documentation JDI, que vous trouverez ici:

- Javadoc JDI :
  `https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/`

Conseil : lisez la doc !
Conseil : ne copiez pas bêtement les tutoriels en ligne, je les connais...

1 Mise en place d'une session de debugging JDI

1.1 Configuration du projet

Créez un projet java **simple**, et créez un package de travail. Par exemple, vous pouvez créer le package `dbg` dans lequel vous mettrez vos

sources. Téléchargez l'archive présente sur `http://kloum.io/costiou/teaching/materials/projects/JDI-sources.zip` dans laquelle vous trouverez les 3 fichiers java pour commencer le projet. Placez ces trois fichiers dans votre package, et lancez le projet une première fois.

Pour que IntelliJ reconnaisse la configuration, il est parfois nécessaire d'effectuer une configuration supplémentaire:

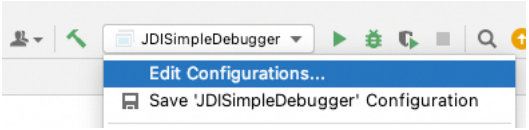

Figure 1: Éditez votre configuration.

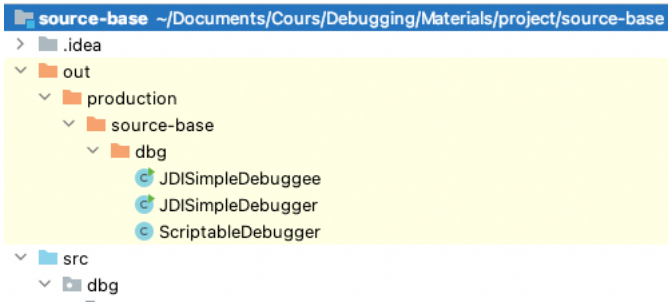

Figure 2: Observez la structure de votre projet. Localisez les binaires produits par la compilation.

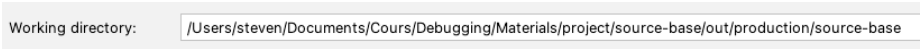

Figure 3: Dans votre configuration, le champ *Working directory* doit pointer sur le dossier immédiatement parent du dossier correspondant au package où sont les sources et dans lequel le résultat de la compilation est enregistré. Dans cet exemple, le dossier *source-base* est le dossier parent du dossier *dbg* dans lequel les sources contenues dans le package `dbg` ont été compilées.

Pour Eclipse, les Figure 4 et Figure 5 précisent la marche à suivre.

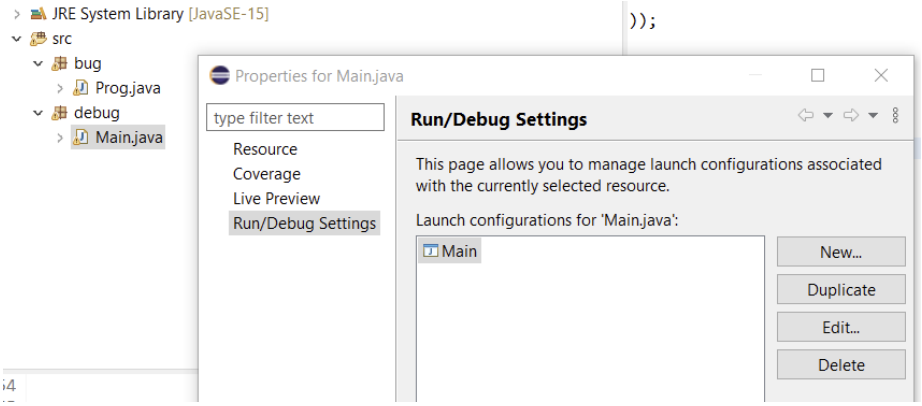

Figure 4: Dans les propriétés `Run/Debug Settings` de votre projet, sélectionnez la configuration `Main` (par exemple), et cliquez sur *Edit...*.

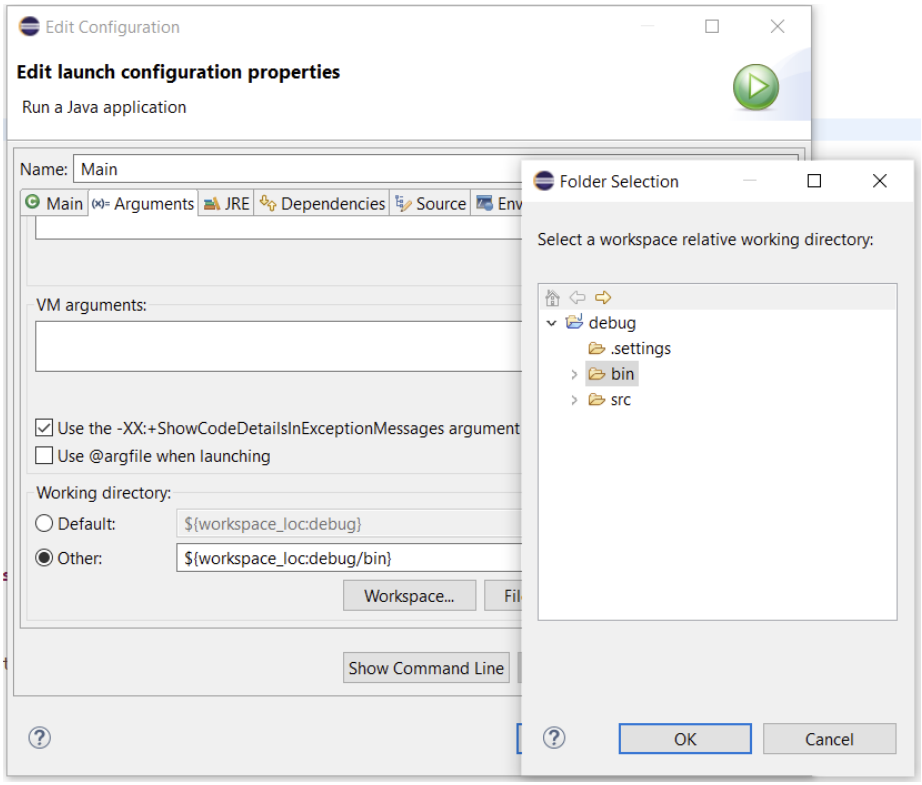

Figure 5: Dans l'onglet *Arguments*, allez en bas du panneau pour trouver la configuration du *Working directory*. Sélectionnez *Other*, puis choisissez via le bouton *Workspace...* le dossier *bin* où sont compilés les binaires du projet.

1.2 Projet de départ : le `SimpleDebugger`

La classe principale du projet est la classe `SimpleDebugger` (Figure 6). Cette classe contient une méthode `main` qui lance un debugger sur une classe `JDISimpleDebuggee` :

1. Instancie un `ScriptableDebugger`. Cet objet implémente le comportement de notre debugger.
2. Attache notre instance de debugger à la classe principale de notre programme à debugger.

```
public class JDISimpleDebugger {
    public static void main(String[] args) throws Exception {

        ScriptableDebugger debuggerInstance = new ScriptableDebugger();
        debuggerInstance.attachTo(JDISimpleDebuggee.class);

    }
}
```

Figure 6: La classe `SimpleDebugger`.

La classe `ScriptableDebugger` (Figure 7) possède deux variables d'instance `debugClass` qui représente la classe avec le `main` du programme à debugger et `vm` qui représente la machine virtuelle sur laquelle s'exécute la `debugClass`.

```
2 usages
private Class debugClass;
3 usages
private VirtualMachine vm;
```

Figure 7: Variables d'instance de la classe `ScriptableDebugger` .

Lorsque notre `SimpleDebugger` attache son instance de `ScriptableDebugger` à la classe à debugger (Figure 6), la méthode `attachTo(Class debuggeeClass)` de `SimpleDebugger` est appelée (Figure 8) avec en paramètre la classe à debugger. Cette dernière renseigne sa `debugClass`, instancie une machine virtuelle (`connectAndLaunchVM()`) puis démarre la session de debugging (`startDebugger()`).

```
public void attachTo(Class debuggeeClass) {

    this.debugClass = debuggeeClass;
    try {
        vm = connectAndLaunchVM();
        startDebugger();
```

Figure 8: Configuration, instantiation de la vm et démarrage du debugger.

La méthode `connectAndLaunchVM()` (Figure 9) utilise l'API **JDI** pour instancier une interface vers une machine virtuelle que nous pourrons manipuler. Cette machine virtuelle est exécutée localement, et votre programme s'y connecte via un connecteur (par défaut, il s'agit d'un accès réseau au *loaclhost*). La VM est paramétrée avec un argument *main* qui indique où trouver la méthode `main` à exécuter, ici dans notre `debugClass`.

```
public VirtualMachine connectAndLaunchVM() throws IOException,
        IllegalConnectorArgumentsException, VMStartException {
    LaunchingConnector launchingConnector =
            Bootstrap.virtualMachineManager().defaultConnector();
    Map<String, Connector.Argument> arguments = launchingConnector.defaultArguments();
    arguments.get("main").setValue(debugClass.getName());
    VirtualMachine vm = launchingConnector.launch(arguments);
    return vm;
}
```

Figure 9: Interface vers la VM qui exécute le programme à debugger.

La méthode `startDebugger()` (Figure 10) démarre la boucle d'interception d'évènements de la VM. Pour cela, nous appelons la méthode `remove()` sur la file d'évènements de la VM. Cette méthode se met en attente du prochain ensemble d'évènements qui seront soulevés par la VM. Lorsqu'un ensemble d'évènements est intercepté, chaque évènement présent dans cet ensemble est traité (ici simplement imprimés dans la console) puis le contrôle est rendu à la VM de manière explicite via l'appel de `resume()`. Il est important de rendre explicitement le contrôle de l'exécution à la VM, car certains évènements (comme les breakpoints) sont bloquants et interrompent l'exécution. Il faut donc dire à la VM qu'elle peut reprendre l'exécution du programme en cours de debug.

```java
public void startDebugger() throws VMDisconnectedException, InterruptedException {
    EventSet eventSet = null;
    while ((eventSet = vm.eventQueue().remove()) != null) {
        for (Event event : eventSet) {
            System.out.println(event.toString());
            vm.resume();
        }
    }
}
```

Figure 10: Boucle d'interception d'évènements de la VM.

L'exécution de ce code doit vous donner 3 évènements et une exception :

```
VMStartEvent in thread main
VMDeathEvent
VMDisconnectEvent
Virtual Machine is disconnected:
        com.sun.jdi.VMDisconnectedException
```

VMStartEvent est l'évènement de démarrage de l'exécution de votre programme par la VM. Il s'agit du premier évènement reçu, mais un certain nombre de lignes de code système ont été exécutés préalablement à l'envoi de cet évènement.

VMDeathEvent est l'évènement de terminaison de la VM. Elle a fini d'exécuter votre programme.

VMDisconnectEvent est l'évènement de déconnexion de la VM : vous n'avez plus accès à son interface.

Enfin, *VMDisconnectedException* est soulevé car votre boucle d'interception d'évènements essaie d'obtenir des évènements d'une machine virtuelle déconnectée et ne prends pas en compte ce cas.

1.3 Interception de l'évènement *VMDisconnectEvent* et logs du programme debuggé

Nous allons tout d'abord traiter le problème d'exception non gérée. Pour cela, nous allons intercepter un premier évènement : le *VMDisconnectEvent*. Lorsque cet évènement est soulevé, l'exécution de notre debuggee est terminée, et donc nous devons sortir de notre boucle de gestion d'évènements.

Rajoutez le traitement correspondant dans la boucle d'interception d'évènements :

```java
if(event instanceof VMDisconnectEvent) {
 System.out.println("===End of program.");
 return;
}
```

Notre debugger se termine maintenant normalement, sans soulever d'exception. Cependant, si vous observez le code de la classe `JDISimpleDebuggee` vous noterez que le programme debuggé devrait imprimer des informations sur la console. Or, aucune information ne s'affiche. C'est parce que vous ne voyez que les informations correspondant au flux de sortie de votre debugger, et non pas celui de la VM qui exécute le debuggee.

À la fin de l'exécution du debuggee, nous devons donc récupérer les informations imprimées sur le flux de cette VM. Pour cela, à la fin de l'exécution, nous récupérons l'*input stream* du processus courant de la VM du debuggee, et nous préparons une écriture sur le flux de sortie du debugger (*i.e.*, `System.out`) :

```
System.out.println("End of program");
InputStreamReader reader =
  new InputStreamReader(vm.process().getInputStream());
OutputStreamWriter writer = new OutputStreamWriter(System.out);
try {
    reader.transferTo(writer);
    writer.flush();
} catch (IOException e) {
    System.out.println("Target VM inputstream reading error.");
}
```

Relancez votre exécution : vous devriez maintenant observer les traces d'exécution de votre debuggee !

```
VMStartEvent in thread main
VMDeathEvent
VMDisconnectEvent
End of program
Simple power printer -- starting
1600.0
Virtual Machine is disconnected: com.sun.jdi.VMDisconnectedException
```

Si les traces ne s'affiche pas correctement, cela signifie que la configuration de votre environnement en 1.1 n'a pas été correctement effectuée.

1.4 Interrompre l'exécution: premier *breakpoint*

Nous allons maintenant ajouter un premier point d'arrêt au début de la méthode `main` de notre debuggee. Pour placer ce première point d'arrêt, il nous faut d'abord intercepter la préparation de notre `debugClass` en mémoire afin d'être certain qu'elle soit connue de la VM.

Dans notre méthode `attachTo(Class debuggeeClass)` (Figure 8) nous ajoutons avant le démarrage du debugger un appel à une nouvelle méthode `enableClassPrepareRequest(Virtual Machine vm)` :

```
vm = connectAndLaunchVM();
enableClassPrepareRequest(vm);
startDebugger();
```

Cette méthode configure une **requête** envoyée à la VM. Pour chaque requête active, la VM génère l'évènement correspondant lors de l'exécution du debuggee. Chaque évènement expose via une API une partie du contexte d'exécution du programme que nous pouvons utiliser pour obtenir de l'information et debugger.

Toutes les requêtes fonctionnent de la même manière : il faut les configurer puis les activer (`request.enable()`). La VM du debuggee continuera à générer les évènements correspondant à une requête tant que celle-ci est active. Il faut donc, pour arrêter la génération d'évènement correspondants à une requête, désactiver explicitement cette dernière (`request.disable()`).

Dans la méthode ci-dessous, nous configurons notre requête pour intercepter la préparation de notre `debugClass` puis nous activons la requête.

```
public void enableClassPrepareRequest(VirtualMachine vm) {
 ClassPrepareRequest classPrepareRequest =
```

```
    vm.eventRequestManager().createClassPrepareRequest();
  classPrepareRequest.addClassFilter(debugClass.getName());
  classPrepareRequest.enable();
  }
```

Une fois la requête active, nous pouvons ajouter la gestion de l'évènement correspondant dans notre boucle d'interception d'évènements. Ici, lorsque notre classe est préparée par la machine virtuelle, nous configurons immédiatement un point d'arrêt sur la première ligne du programme `main` :

```
if (event instanceof ClassPrepareEvent) {
  setBreakPoint(debugClass.getName(), 6);
  }
```

La méthode `setBreakPoint(String className, int lineNumber)` configure une requête pour placer et activer un point d'arrêt dans la classe nommée `className` à la ligne `lineNumber`. Nous devons d'abord trouver la classe du debuggee, puis obtenir la *location* de la ligne dans le fichier de cette classe, c'est-à-dire la représentation dans le modèle interne de la VM de l'instruction située à la ligne demandée dans le code source. Cela doit respecter deux contraintes :

- Le numéro de ligne sélectionnée doit contenir une instruction, c'est-à-dire ne pas être vide. Les ligne de code source vide ne sont pas représentée par du code compilé.
- Plusieurs locations sont possible pour une même ligne de code source. En effet, vous pouvez répartir une même instruction sur plusieurs lignes de code source, mais vous ne pouvez pas placer votre point d'arrêt au milieu d'une instruction. Vous devez donc sélectionner la première location retournée, qui correspond à la ligne dans votre code source où commence une instruction.

Une fois la location obtenue, nous configurons une requête de point d'arrêt sur cette instruction et activons cette requête.

```
public void setBreakPoint(String className, int lineNumber)  {
  for (ReferenceType targetClass : vm.allClasses())
    if (targetClass.name().equals(className)) {
      Location location =
        targetClass.locationsOfLine(lineNumber).get(0);
      BreakpointRequest bpReq =
        vm.eventRequestManager().createBreakpointRequest(location);
      bpReq.enable();
    }
  }
```

Lorsque vous relancez votre debugger, vous observez maintenant un `BreakpointEvent` dans la console !

1.5 Contrôle de l'exécution : premier *step*

Nous allons maintenant configurer un stepping de l'exécution lors de l'interception de notre premier point d'arrêt. Nous créons et appelons une nouvelle méthode `enableStepRequest(LocatableEvent event)` à partir de notre boucle d'interception. Il est important de *caster* l'évènement générique que la VM nous a envoyé en un évènement de type `LocatableEvent`, qui contient

des informations contextuelles à la location à laquelle le point d'arrêt a stoppé l'exécution :

```
if (event instanceof BreakpointEvent) {
  enableStepRequest((BreakpointEvent) event);
  }
```

Dans cette méthode, nous configurons et activons une requête de *stepping* :

```
public void enableStepRequest(LocatableEvent event) {
  StepRequest stepRequest =
    vm.eventRequestManager().
      createStepRequest(event.thread(),
                        StepRequest.STEP_MIN,
                        StepRequest.STEP_OVER);
    stepRequest.enable();
  }
```

Nous devons préciser :

1. Le thread qui sera exécuté pas à pas. Ici, celui à partir duquel a été généré l'évènement intercepté.

2. La granularité du stepping : `STEP_MIN` step jusqu'à la prochaine location disponible sur la même ligne (*e.g.*, un appel de méthode) et `STEP_LINE` step directement jusqu'à la prochaine ligne.

3. Le type de stepping : `STEP_INTO` (step dans chaque nouvelle frame), `STEP_OVER` (step par dessus chaque nouvelle frame) ou `STEP_OUT` (step jusqu'à sortir de la frame courante).

Une fois cette requête active, la VM générera un évènement *step* pour chaque instruction satisfaisant ce paramétrage et ce jusqu'à la fin du programme ou jusqu'à ce que la requête soit désactivée. Pour un thread donné, il n'est pas possible d'avoir plus d'une seule requête de step active à la fois.

Redémarrez votre debugger et observer les évènements de stepping loggés sur la console.

1.6 Observer l'état du programme : les `event`

Un `event` représente un évènement notable de l'exécution, et soulevé par la VM. Lors de la construction d'un debugger, nous exploitons ces évènements pour observer l'état du programme en cours de debug, extraire de l'information sur ce programme et implémenter des fonctionnalités.

1. Placez un point d'arrêt sur chaque cas de traité dans votre boucle d'évènements (préparation des classes, breakpoints et step),

2. lancez votre debugger en mode *debug*,

3. à chaque fois que votre debugger s'arrête, observez l'objet *event* : parcourez son état et son graphe d'objets.

1.7 Contrôle manuel : une première commande textuelle

Nous voulons maintenant donner le contrôle manuel de l'exécution aux développeuses et développeurs :

1. Créez une nouvelle méthode pouvant lire du texte sur le flux d'entrée standard. Cette méthode détecte la commande *"step"*. Si la développeuse écrit *"step"*, alors un stepping est configuré. Si la développeuse écrit autre chose, le programme continue jusqu'à la fin de son exécution.

2. Dans la gestion des évènements correspondant au points d'arrêt, remplacer le paramétrage d'un stepping par un appel à cette méthode. En d'autres mots : vous donnez le contrôle aux développeurs lors de l'arrêt sur le premier breakpoint.

3. Ajoutez la prise en charge des évènements de stepping : à chaque step, reprenez le contrôle et attendez à nouveau une entrée utilisateur.

2 Contrôle du debugger en ligne de commande

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 → 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 → 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 → 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.

3 BONUS : Contrôle par une interface graphique

Nous voulons maintenant construire une interface graphique permettant, en utilisant votre langage de script, de contrôler et de visualiser l'exécution de votre programme en cours de debug.

Implémentez une interface graphique simple permettant d'exécuter toutes les commandes définies dans la Section 2. Vous devez notamment pouvoir :

1. Utiliser des boutons pour exécuter chaque commande,
2. visualiser, à chaque step, le code source au point d'exécution courant,
3. visualiser, à chaque step, l'ensemble du contexte : frame, call-stack, variables locales et d'instances,
4. visualiser l'ensemble des points d'arrêt actifs et leurs propriétés.