UNIVERSITÉ BRETAGNE OCCIDENTALE

Rapport du debugger

Warda NEMMAR

Kevin Lumwanga

1. Les package du projet :

Le projet est composé principalement des packages suivants:

- **Core** : ce package contient 5 fichiers qui contiennent des méthodes spécifique à chaque commande, ces méthodes sont appelé dans l'autre package outils, et elles (ces méthodes) contiennent le code spécifique à chaque commande ;
- **Outils** : Ce package contient toutes les commandes implémenté dans le débuguer, ses classes contiennent principalement la méthode exécute qui fait appel aux méthodes dans le Core qui permet d'obtenir le résultat attendu pour chaque commande.

2. Les commandes :

Chaque commande est créée dans une classe spécifique qui détermine son fonctionnement dans la méthode implémentée dans la classe de chaque commande :

```
public void execute(Event event, ScriptableDebugger scriptableDebugger) {
    Appel à la méthode Core. Méthode(…)
…
 }
```

1. Step :

Execute la prochaine instruction. S'il s'agit d'un appel de méthode, l'exécution entre dans cette dernière.

```
public class CommandStep implements Command {
   @Override
   public void execute(Event event, ScriptableDebugger scriptableDebugger) {
      FabriqueCommand.enableStepRequest ((LocatableEvent) event,scriptableDebugger);
   }
}
```

La fonction execute de notre commande step fait appel à la méthode suivante :

```
public static void enableStepRequest(LocatableEvent event, ScriptableDebugger
scriptableDebugger) {
      if (scriptableDebugger.getStep() == null) {
         StepRequest step =
scriptableDebugger.getVm().eventRequestManager().createStepRequest(
                  event.thread(),
                  StepRequest.STEP_LINE,
                  StepRequest.STEP_INTO
         );
         scriptableDebugger.setStep(step);
      }
      scriptableDebugger.getStep().enable();
   }
```

- StepRequest demande de notification lorsqu'une une step est détecté dans la VM cible,ceci provoque un Stepevent qui sera stocké dans une pile: EventQueue
- La fonction scriptableDebugger.getStep() permets de récupérer la step actuel ;
    - Si scriptableDebugger.getStep() return un null doit créer une step requeste pour pouvoir exécuter STEP_LINE et STEP_INTO ;
    - StepRequest.STEP_INTO pour accéder au frame nouvellement créé.

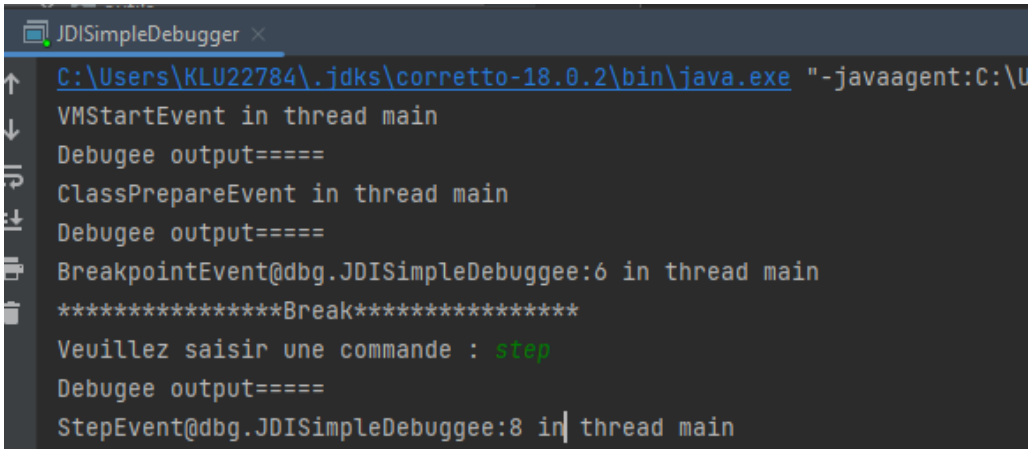

Figure 1: execution de la commande step

2. step-over :

- La méthode exécute de CommandStepOver appel à FabriqueCommand. enableStepOverRequest qui spécifie le fonctionnement de step-over comme suit :

```
public static void enableStepOverRequest(LocatableEvent event, ScriptableDebugger
scriptableDebugger) {
    if (scriptableDebugger.getStepOver() == null) {
        StepRequest stepOver =
scriptableDebugger.getVm().eventRequestManager().createStepRequest(
                event.thread(),
                StepRequest.STEP_LINE,
                StepRequest.STEP_OVER
        );
        scriptableDebugger.setStepOver(stepOver);
    }
    scriptableDebugger.getStepOver().enable();
}
```

- Si scriptableDebugger.getStepOver() return un null on doit créer une step request pour pouvoir exécuter STEP_LINE et STEP_Over ;
- StepRequest.STEP_LINE permet de Passer à l'emplacement suivant sur une ligne différente
- StepRequest.STEP_OVER permet de Passer à la fin des blocs des step.

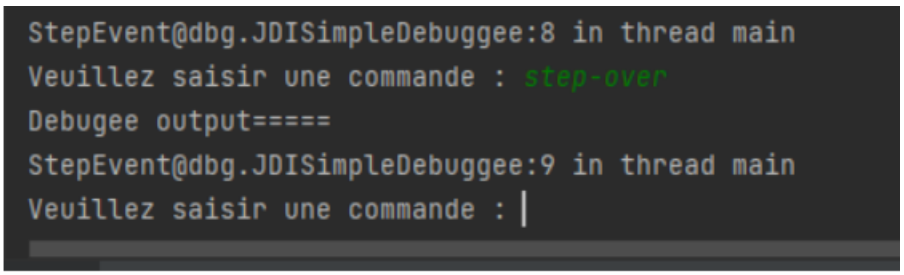

Figure 2: ecécution de la commande step-over

3. Continue :

Continue l'exécution jusqu'au prochain point d'arrêt. La granularité est l'instruction step.

Cette commande permet de passer d'un breakpoints à un autre. S'il n'y a qu'un seul breakpoints l'exécution se termine.

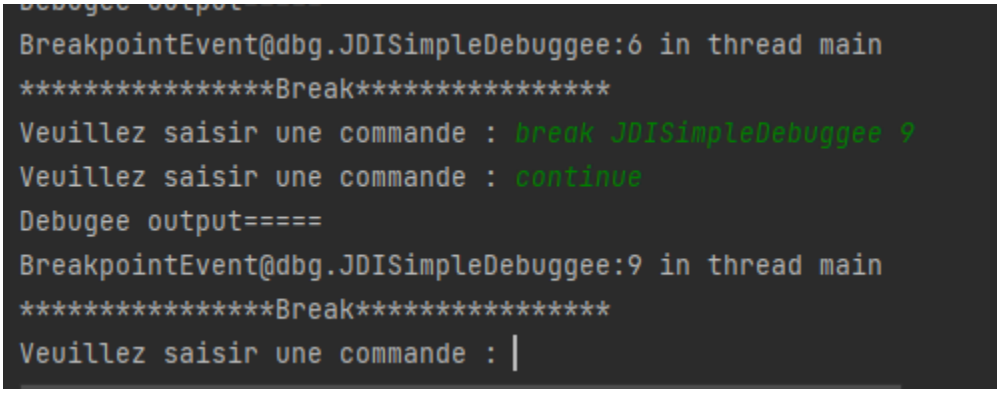

Figure 3commande break et continue

4. Frame :

- La méthode exécute de CommandFrame appel à FabriqueFrameCommand. EnableCommandFrame qui récupère la Frame actuel comme suit:

```
public final class FabriqueFrameCommand {
   public static void enableCommandFrame(Event event, ScriptableDebugger
scriptableDebugger) throws ClassNotLoadedException,
IncompatibleThreadStateException, AbsentInformationException,
InterruptedException {
        StackTraceElement[] stackframe =  Thread.currentThread().getStackTrace();
        System.out.println(" frame || "+stackframe[0]);
        scriptableDebugger.choseMethode(event);
   }
…
}
```

- La fonction StackTraceElement[] dans la classe StackFrame récupère la pile d'exécution des frames, et la frame actuel correspond à l'élément [0] de cette pile.

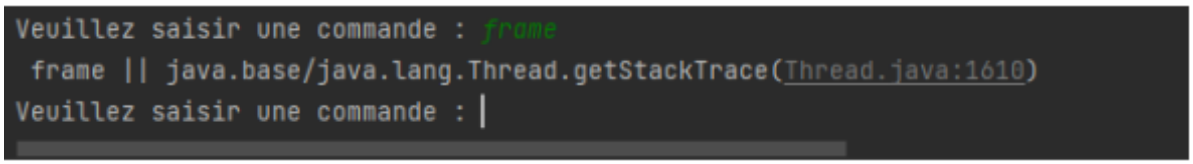

Figure 4: Exécution commande Frame

5. temporaries :

Renvoie et imprime la liste des variables temporaires de la frame courante, sous la forme de couples nom → valeur.

- stackFrame.visibleVariables () permets de récupérer la liste des variables locales de la frame courante, ces données récupérées on les stocks dans une Map<nom Variable, valeur Variable>, puis on affiche ces valeur on les parcourant avec une boucle foreach ;
- stackFrame.getArgumentValues () permet de récupérer la liste des arguments s'ils existent et les stocker (leurs valeurs) dans une liste res.

```
Veuillez saisir une commande : temporaries
StackFrame : dbg.JDISimpleDebuggee:6 in thread instance of java.lang.Thread(name='main', id=1)
Name  ==> Value
args ==> instance of java.lang.String[0] (id=465) ;
Veuillez saisir une commande :
```

Figure 5: Exécution de la comande temporaries

1. stack :

Renvoie la pile d'appel de méthodes qui a amené l'exécution au point courant:

```
public class CommandStack implements Command{
   @Override
   public void execute(Event event, ScriptableDebugger scriptableDebugger) throws
IncompatibleThreadStateException, AbsentInformationException, InterruptedException,
ClassNotLoadedException {
      StackTraceElement[] stackframe =  Thread.currentThread().getStackTrace();
      for (StackTraceElement sf:stackframe
      ) {
         System.out.println(" Stack || "+sf);
      }
      scriptableDebugger.choseMethode(event);
   }
}
```

- La méthode Thread.currentThread().getStackTrace() de StackTraceElement[] permet de récupérer la trace des méthodes qui ont ramené au thread courant, cette trace est stocké dans un tableau de StackTraceElement.

```
Veuillez saisir une commande : stack
 Stack || java.base/java.lang.Thread.getStackTrace(Thread.java:1610)
 Stack || dbg.outils.CommandStack.execute(CommandStack.java:25)
 Stack || dbg.ScriptableDebugger.choseMethode(ScriptableDebugger.java:256)
 Stack || dbg.outils.CommandReceiverVariable.execute(CommandReceiverVariable.java:38)
 Stack || dbg.ScriptableDebugger.choseMethode(ScriptableDebugger.java:256)
 Stack || dbg.ScriptableDebugger.startDebugger(ScriptableDebugger.java:292)
 Stack || dbg.ScriptableDebugger.attachTo(ScriptableDebugger.java:136)
 Stack || dbg.JDISimpleDebugger.main(JDISimpleDebugger.java:7)
Veuillez saisir une commande :
```

Figure 6: exécution de la commande stack

7 . receiver :

La méthode execute de cette commande fait appel à la méthode suivante: FabriqueFrameCommand.enableCommandReceiver (event, scriptableDebugger);

```
public static void enableCommandReceiver(Event event, ScriptableDebugger
scriptableDebugger) throws ClassNotLoadedException, IncompatibleThreadStateException,
AbsentInformationException, InterruptedException {
      StackFrame stackFrame = ((LocatableEvent) event).thread().frame(0);
      ObjectReference ob = stackFrame.thisObject();
```

```
        if (ob == null) {
            System.out.println("Receiver Class: " + stackFrame.location().method());
        } else {
            System.out.println("Receiver : " + ob);
        }
        scriptableDebugger.choseMethode(event);
    }
```

- Event.thread().frame(0) récupère la frame actuelle qui stocké dans la pile des thread avec le numéro d'indice [0] ;
- Cette méthode fait appel à la classe ObjectReference du frame (0) (la frame actuelle);
- la méthode thisObject () permet de récupérer la valeur de 'this' du frame courante frame(0).

8. sender :

- Event.thread ().frame(1) de la classe StackFrame permets récupérer la frame qui nous remmène vers la frame courante(0) ;
- Si (LocatableEvent)event).thread().frames().size()==1 donc le sender n'existe pas, c'est à dire que on a pas de sender ;
- Si on est dans une classe static , le sender n'existe pas , d'où vient la condition: if stackFrame.thisObject() == null.

9. receiver-variables :

Renvoie et imprime la liste des variables d'instance du receveur courant, sous la forme d'un couple nom → valeur.

- On utilise le même principe que receiver pour récupérer l'objet actuel (this) avec stackFrame.thisObject() objectReference;
- On crée une Map<field.name,field.value> pour stocker les variables a afficher : tels que le nom ce sont les value de la Map , et les clés sont les noms des variables;
- objectReference.getValues(objectReference.referenceType().allFields()) permet de récupérer toutes les variables du receiver, ces variables on ajoute leurs <nom,valeur> dans la Map objectReference puis on les affiche.

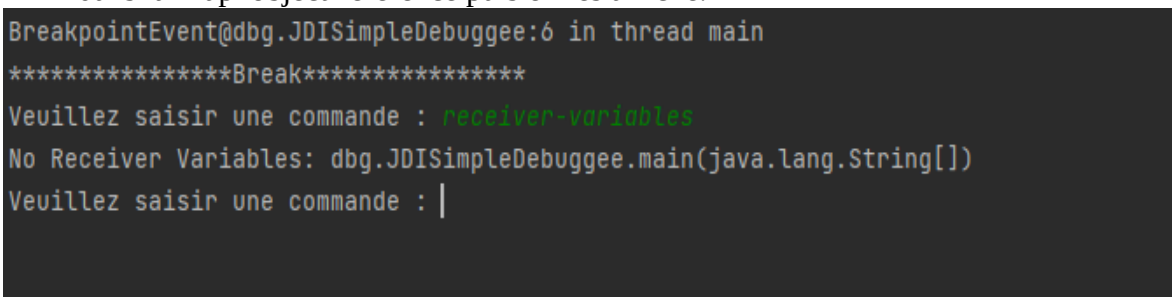

Figure 7: commande receiver-variables

10 . method :

- Tout d'abord il faut récupérer la frame actuel (frame(0));
- Récupérer l'objet référence de this ObjectReference;
- Pour cette méthode on a pensé à récupérer les informations suivantes:
- le type de retours de la méthode en utilisant la fonction: stackFrame.location().method().returnType();

- Le nom de la méthode: stackFrame.location().method().name();
- Les arguments : stackFrame.location().method().argumentTypeNames());
- La type de retour de la méthode: stackFrame.location().method().returnType();

```
***************Break***************
Veuillez saisir une commande : method
La methode  en cours d execution :
     void main ([java.lang.String[]])

le nom de la methode courante : main
Les parametres sont  : [java.lang.String[]]
Le type de retour est : void
```

Figure 8: exécution de la commande méthode

11 . arguments :

Renvoie et imprime la liste des arguments de la méthode en cours d'exécution, sous la forme d'un couple nom → valeur.

- Tout d'abord il faut récupérer la frame actuel (frame(0));
- Puis,on test si la méthode courante n'a pas d'argument avec la condition suivante: stackFrame.location().method().arguments().size() == 0 ;
- Si la condition est vérifié on affiche un message disant qu'on a pas de paramétres ;
- Sinon, on continue:
- Les arguments(paramètres) de la méthode actuelle sont accessibles via la méthode suivante: stackFrame.location().method().arguments().

12. print-var(String varName) :

Imprime la valeur de la variable passée en paramètre.

- On commence par tester si on a le nombre correspondant d'arguments, un message d'erreur sera affiché sinon ;
- ensuite il faut récupérer la frame actuel (frame(0));
- ensuite on cherche dans les variables locales de la frame actuel la variable avec le nom varName passé en paramètre:stackFrame.visibleVariableByName(varName);
- Si la variable existe on affiche sa valeur avec la fonction stackFrame.getValue(local Variable);
- Si la variable n'existe pas on affiche un message qui l'indique.

13. break(String filename, int lineNumber) :

Execute fait appel à FabriqueBreakCommand.enableSetBreakPoint(event, scriptableDebugger);
La FabriqueBreakCommand est la fabrique pour toutes les méthode break.

- Cette commande permet de placer un breakpoint à la ligne **lineNumber** de la classe **filename**.
- On parcourt toutes les classes du vm, on vérifie si la classe **filename** existe bien dans la vm.

- Si la classe existe bien dans la vm, on place le breakpoints à la ligne **lineNumber**.
- Si la classe n'existe pas dans la vm, alors on ne fait rien.
- On redonne la main à l'exécutable à fin de saisir une nouvelle commande.

14. breakpoints :

- CommandBreakPoints.execute appel FabriqueBreakCommand.enableBreakPoints ;
- On a besoin de récupérer tous les breakpoints de notre VM avec la méthode : scriptableDebugger.getVm ().eventRequestManager().breakpointRequests())
- Une fois la liste des points d'arrêt est récupérée, pour chaque point , on a la classe BreakpointRequest contient une fonction isEnabled() qui permets tester si les event requests sont enabled ;
- Si isEnabled() est à vrai donc on a un breakpoint à afficher comme suit :
    - Breakpoint.location().Name pour le nom de la classe où le break point est situé ;
    - Le numéro de la ligne est récupéré avec breakpoint.location().lineNumber() ;

Figure 9: ecécution de la commande breakpoints

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.

- Pour cette commande on a deux arguments à prendre en considération ;
- On commence donc par vérifier le bon nombre d'arguments ;
- CommandBreakPointOnce.execute appel FabriqueBreakCommand. enableSetBreakPointOnce ;
- Pour récupérer le nom de la classe on a besoin du premier argument, ceci est à récupérer avec scriptableDebugger.getCmd ().split (" ") qui découpera notre ligne de commande en 3 mots selon le séparateur « espace » : le premier argument qui est le nom de la classe est le [1], le numéro de la ligne correspond u deuxième argument [2] ;
- La fonction setBreakPointOnce(className, lineNumber, scriptableDebugger) installe le point d'arrêt dans l'endroit souhaité;
    - scriptableDebugger.getVm().allClasses()) récupère toutes les classe de notre VM ;
    - On parcourt ces classe ensuite : si le nom de la classe correspond à ClassName donnée en paramètre :
        - On crée un breakpoint à ligne « lineNumber » avec scriptableDebugger.getVm().eventRequestManager().createBreakpointRequest(targetClass.locationsOfLine(lineNumber).get(0));
        - Puis on l'active avec la fonction BreakpointRequest.enable() ;

- Enfin on ajoute le nouveau breakpoint créé à la liste des BreakpointRequest du Debugger avec la fonction scriptableDebugger.getBreakpointRequestList().add(breakpointRequest);

- Thread.sleep(200) permet d'attendre 200ms avant redonner la main à l'utilisateur pour saisir une autre commande;

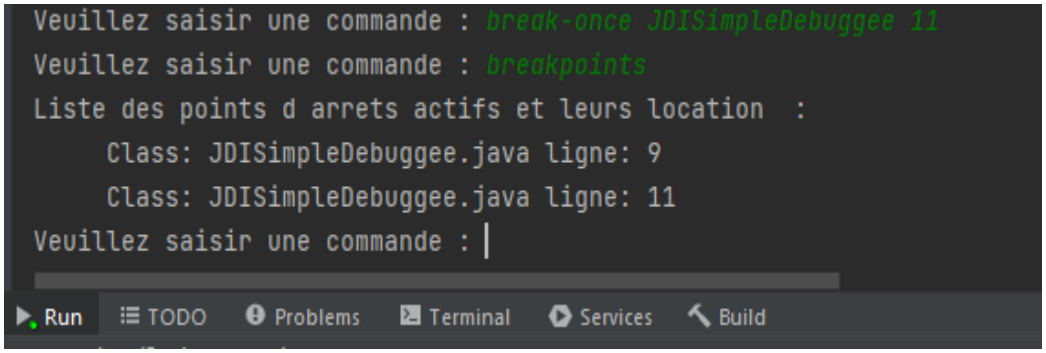

Figure 10: commande break-once

16. break-on-count(String filename, int lineNumber, int count) :

Installe un point d'arrêt à la ligne lineNumber du fichier file Name. Ce point d'arrêt ne s'active qu'après avoir été atteint un certain nombre de fois count.

- CommandBreakOnCount.execute fait appel à la fonction suivante : FabriqueBreakCommand.enableBreakOnCount;
- Pour récupérer les arguments donné en paramètres on utilise la fonction scriptableDebugger.getCmd ().split (" ") de la même façon que la commande précédente ;
- On appel la fonction setBreakOnCount(className, lineNumber, count, scriptableDebugger) qui installe le point d'arrêt dans l'endroit souhaité :
  - On commence par désactiver les breaks qui existe déjà dans notre debugger disactiveAllStep(scriptableDebugger.getVm());
  - Cette fonction a un comportement similaire à celle de la commande précédente, mais ici on prend un 4^{ième} paramètre qui est count ;
  - scriptableDebugger.getVm().allClasses()) récupère toutes les classe de notre VM ;
  - On parcourt ces classes ensuite : et on cherche celle avec un nom égale à ClassName donnée en paramètre :
    - On crée un breakpoint à ligne « lineNumber » avec scriptableDebugger.getVm().eventRequestManager().createBreakpointRequest(targetClass.locationsOfLine(lineNumber).get(0));
    - On disable le point d'arrêt avec la fonction BreakpointRequest.disable() ,ceci pour le réactiver que quand on atteint le count :
      - On recupère toutes les StepRequest avec la méthode vm.eventRequestManager().stepRequests() puis on les désactive ;
    - Enfin on ajoute le nouveau breakpoint créé à la liste des BreakpointRequest du Debugger avec la fonction scriptableDebugger.getBreakpointRequestList().add(breakpointRequest);

- scriptableDebugger.getBreakpointRequestCount().setCount(count); pour mettre à jour la valeur du count pour notre nouveau point d'arrêt qu'on vient de créer ;
- scriptableDebugger.getBreakpointRequestCount().setBreakpointRequest(breakpointRequest) pour ajouter notre point arrêt count à notre débugger.

- Thread.sleep(200) permet d'attendre 200ms avant redonner la main à l'utilisateur pour saisir une autre commande.

```
****************Break****************
Veuillez saisir une commande : break-on-count JDISimpleDebuggee 15 2
Veuillez saisir une commande : step
Debugee output=====
StepEvent@dbg.JDISimpleDebuggee:8 in thread main
Veuillez saisir une commande : step
Debugee output=====
StepEvent@java.lang.invoke.DirectMethodHandle:328 in thread main
```

Figure 11: exemple d'exécution break-on-count

17. break-before-method-call(String methodName) :

Configure l'exécution pour s'arrêter au tout début de l'exécution de la méthode methodName.

- On commence par vérifier qu'on a le bon nombre d'arguments ;
- La méthodefait la configuration pour arrêter au tout début de methodName scriptableDebugger.enableBreakBeforeMethodCall(methodName);
- scriptableDebugger.choseMethode(event) permet de donner la main au utilisateurs pour choisir la prochain méthode a exécuter;

```
****************Break****************
Veuillez saisir une commande : break-before-method-call printPower
Veuillez saisir une commande :
```

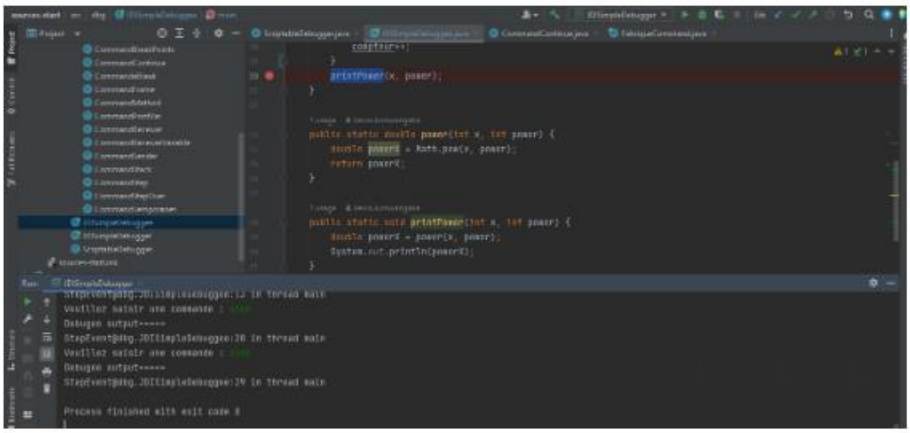

Figure 12: Exemple d'exxécution de la commande break-before-method-call

3. Configuration IntelliJ: