# Environnements de développement (intégrés)

## JDT (débogage), outils d'analyse statique

Patrick Labatut
labatut@di.ens.fr
http://www.di.ens.fr/~labatut/

Département d'informatique
École normale supérieure

Centre d'enseignement et de recherche en technologies de l'information et systèmes
École des ponts

Transparents inspirés de ceux du cours de Mihaela Sighireanu en 2006-2007.

# Plan

# Introduction

## Définition

Un débogueur (symbolique) est un outil permettant d'inspecter l'exécution d'un programme (tout en visualisant le code source original) de plusieurs manières :

- exécution pas-à-pas,
- suspension de l'exécution lorsque certaines conditions sont remplies,
- inspection de la valeur des variables et de la mémoire,
- (modification dynamique de l'état du programme).

## Pré-requis

Pour fonctionner, le débogueur a besoin d'avoir un code binaire :

- « annoté », pour faire le lien avec le code source (numéros des lignes dans le code source, noms des variables, ...),
- dépourvu d'optimisations qui rendraient impossible le débogage.

# Lancer un programme en mode débogage

Pour produire du code binaire utilisable avec un débogueur :

- en C, avec `gcc` : compiler avec l'option `-g`,
- en Java, avec `javac` : compiler avec l'option `-g`,
- en Java, sous Eclipse : le JDT le fait automatiquement...

Pour lancer le programme en mode débogage :

- en C, avec `gdb` : `gdb binaire`,
- en Java, avec `jdb` : `jdb package/Classe` (`Classe` doit contenir une méthode `main()`),
- en Java, sous Eclipse : sélectionner un élément (projet ou fichier de source) dans la vue *Package Explorer* et utiliser *[ Run / Debug As / Java Application ]* (et basculer en perspective *Debug*).

# Conseils pour faciliter le débogage

*Attention :* certaines fonction des débogueurs repose sur un découpage du code source en lignes.

- Ne pas regrouper plusieurs instructions sur une même ligne, écrire chaque instruction sur une ligne différente,
- Récupérer les valeurs renvoyées par des appels de méthode dans des variables temporaires,
- Donner des noms facilement identifiables aux variables (en particulier éviter les homonymies entre variables/champs/méthodes de méthodes/classes différentes).

# Vocabulaire 1/2

Point d'arrêt[1] : indique au débogueur de suspendre l'exécution du programme au point où il est défini. Plusieurs types de points d'arrêts :

- sur une ligne (arrêt avant l'exécution de la ligne),
- appel à une fonction ou une méthode (en entrée ou en sortie),
- accès en lecture et/ou écriture à une variable ou un champ,
- évènement (exception, chargement de bibliothèque/classe).

Pile d'appels[2] : mémorise les appels successifs aux fonctions/méthodes toujours en cours d'exécution jusqu'à l'instruction courante (cf. exemple suivant).

Cadre de pile[3] : structure de données au sein de la pile des appels qui pour un appel particulier à une fonction/méthode stocke les variables locales correspondantes (cf. exemple suivant).

---

[1] *Breakpoint*, en anglais.
[2] *Call stack*, en anglais.
[3] *Stack frame*, en anglais.

# Vocabulaire 2/2

```
1  class CallStack {
2      public static void main(String[] args) {
3          method1(1, 2);
4      }
5      public static void method1(int i1, int j1) {
6          method2(i1, i1 == j1);
7      }
8      public static void method2(int i2, boolean b2) {
9          if (!b2)
10             method1(i2, i2);
11         System.out.println(i2);
12     }
13 }
```

État de la pile d'appel une fois dans le deuxième appel à la méthode `method2()` (la partie droite correspondant aux cadres de pile) :

| | | |
|---|---|---|
| `method2()` | `boolean b2` | `true` |
| | `int i2` | `1` |
| `method1()` | `int j1` | `1` |
| | `int i1` | `1` |
| `method2()` | `boolean b2` | `false` |
| | `int i2` | `1` |
| `method1()` | `int j1` | `2` |
| | `int i1` | `1` |
| `main()` | `String[] args` | `...` |

# La perspective *Debug* du JDT

Le JDT fournit une perspective *Debug* dédiée au débogage de programmes Java.

Cette perspective est accompagnée de plusieurs vues dont :

- *Debug* : historique d'exécution, pile d'appel,
- *Variables* : visualiser les variables locales,
- *Breakpoints* : liste, type et état des points d'arrêts,
- *Expressions* : évaluer des expressions à l'exécution.

L'instruction courante est marquée par une flèche dans la marge de gauche de la vue *Java Editor*.

# Points d'arrêt 1/2

Créer/supprimer un point d'arrêt : *[ Mouse-R / Toggle Breakpoint ]*, dans la marge de gauche de la vue *Java Editor*.

Activer/désactiver un point d'arrêt : *[ Mouse-R / Enable Breakpoint ]* / *[ Mouse-R / Disable Breakpoint ]* (s'il est désactivé, le point d'arrêt est marqué par un rond blanc (au lieu de bleu) et est simplement ignoré par le débogueur).

Types de points d'arrêt disponibles dans le JDT :

- Ligne (*[ Mouse-R / Toggle Breakpoint ]*),
- Méthode (*[ Run / Toggle Method Breakpoint ]*),
- Champ (*[ Run / Toggle Watchpoint ]*),
- Exception (*[ Run / Add Exception Breakpoint. . . ]*),
- Chargement de classe (*[ Run / Add Class Load Point. . . ]*).

# Points d'arrêt 2/2

Modifier les propriétés d'un point d'arrêt : dans la vue *Breakpoints*, sélectionner le point d'arrêt, utiliser le menu contextuel *[ Mouse-R ]*.

- Tous les types de points d'arrêt : compteur (*Hit Count*), permet d'atteindre directement la *n*-ème occurence du point d'arrêt,
- Ligne : condition (*[ Mouse-R / Breakpoint Properties. . . ]*),
- Méthode : en entrée ou en sortie,
- Champ : accès en lecture et/ou en écriture,
- Exception : rattrapée ou non.

# Pile d'appel, variables et expressions

Dans la vue *Debug* : visualisation des exécutions, de la pile d'appel courante et contrôle de l'exécution (aussi disponible dans *[ Run ]*).

- *Terminate* : met fin au programme,
- *Resume* : reprend l'exécution normale (jusqu'au prochain point d'arrêt),
- sélection d'un cadre de pile puis *Drop To Frame* : remet la pile d'appel et les variables locales dans l'état correspondant,
- *Step Over*, *Step Into*, *Step Return* : exécute l'instruction en sautant/sans sauter les appels de méthodes, saute la méthode qui vient juste d'être appelée.

Dans la vue *Variables* : variables locales.

On peut aussi survoler la vue *Java Editor* pour avoir les valeurs courantes des variables et des champs.

Dans la vue *Expressions* : rajouter des expressions à évaluer (avec mise à jour dynamique de l'évaluation lors de l'exécution).

# Démonstration

# Analyse statique : introduction 1/2

## Définition

Technique permettant de déduire des résultats sur l'exécution d'un programme sans avoir à l'exécuter[4].

## Problème

L'analyse de programmes est indécidable : il n'existe pas d'algorithme fiable permettant de détecter des erreurs dans un programme à partir de son code source/code objet[5].

En pratique, on peut diminuer la complexité du programme à analyser et/ou la précision de l'analyse souhaitée pour rendre cette analyse possible.

---

[4] par opposition à l'analyse dynamique qui a lieu à l'exécution...

[5] cf. Théorème de Rice : toute propriété non-triviale d'une machine de Turing est indécidable

# Analyse statique : introduction 2/2

## (Un des) objectif(s)

Prévenir les bogues potentiels avant qu'ils ne se déclarent à l'exécution (et qu'il ne devienne nécessaire d'utiliser le débogueur).

Exemple d'application : logiciels critiques (Ariane V. . . ).

Exemples d'outil en C : `lint`/`splint` qui détecte les erreurs potentielles dans le code (moins utilisé aujourd'hui, car les compilateurs C rapportent également le même type d'erreurs).

Attention, l'analyse statique reste relativement lourde et ne dispense pas de l'écriture de tests unitaires.

# Un exemple d'outil d'analyse statique : *FindBugs*

*FindBugs*, logiciel libre (encore en cours de développement), disponible à la fois sous forme de programme indépendant ou de greffon pour Eclipse.

Utilisé pour trouver des centaines (voire milliers) de bogues dans des projets importants : JDK, Netbeans, Eclipse...

## Pré-supposé

Malgré les tests unitaires et toute l'expérience des programmeurs, les bugs qui subsistent restent en général relativement simples. *FindBugs* essaie de trouver des bogues « classiques » (*Bugs Patterns*)[6].

Pour installer le greffon pour Eclipse, cf. : `http://findbugs.sf.net/` ([ *Help / Software Update / Find and Install...* ]).

[6] cf. `http://findbugs.sourceforge.net/bugDescriptions.html` pour avoir la liste de ces *Bugs Patterns*.

# Utilisation de *FindBugs* depuis Eclipse

Le greffon rajoute une perspective dédiée et plusieurs vues.

On lance l'analyse d'un élément du *Package Explorer* via *[ Mouse-R / FindBugs / FindBugs ]*.

Les rapports de bogues potentiels viennent enrichir les avertissements affichés dans la vue *Problems*.

Différentes catégories de « bogues » :

- *P* : Performance (problème de performance, code non optimal)
- *B* : Bad Practice (mauvaise pratique)
- *D* : Dodgy (problème délicat)
- ...

Priorité de « bogues » : *H* : élevée, *M* : moyenne, *L* : faible.

Évidemment, le rapport de bogues est imparfait, +/- de faux positifs suivant les catégories et parfois *FindBugs* n'arrive pas à trouver certains problèmes...

# Démonstration