Debogage sous Eclipse

Objectif du TP

L'objectif du TP est de se familiariser avec la perspective de debogage d'Eclipse.

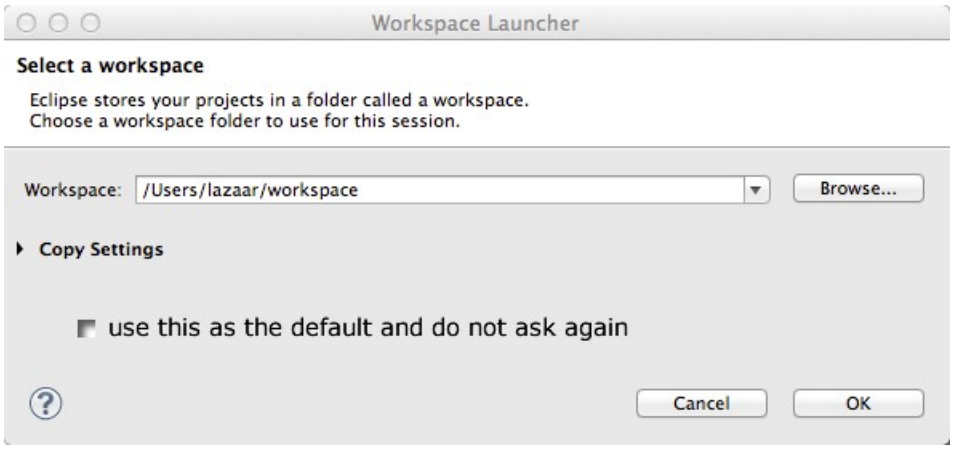
1 Eclipse : quelques manipulations de base

Eclipse est un environnement de développement gratuit, écrit en Java, et dédié à Java, ainsi qu'à d'autres langages grâce à la notion de plugin que nous n'aborderons pas ici. Nous indiquons ici des manipulations de base. Si vous connaissez déjà Eclipse, lisez ce qui suit et assurez-vous que vous connaissez toutes les manipulations indiquées.

1.1 Lancer Eclipse

Lancez Eclipse avec la commande `eclipse &`, ou en double-cliquant sur le raccourci que vous avez sur le bureau. Au bout de quelques instants, on vous demandera de situer l'espace de travail dans lequel seront vos fichiers. Si vous travaillez sur un ordinateur partagé, il est conseillé de mettre l'espace de travail dans votre dossier Documents (sur Windows XP il se trouve dans le dossier Documents and Settings). Si vous êtes le seul utilisateur de votre système, mettez l'espace de travail où bon vous semble.

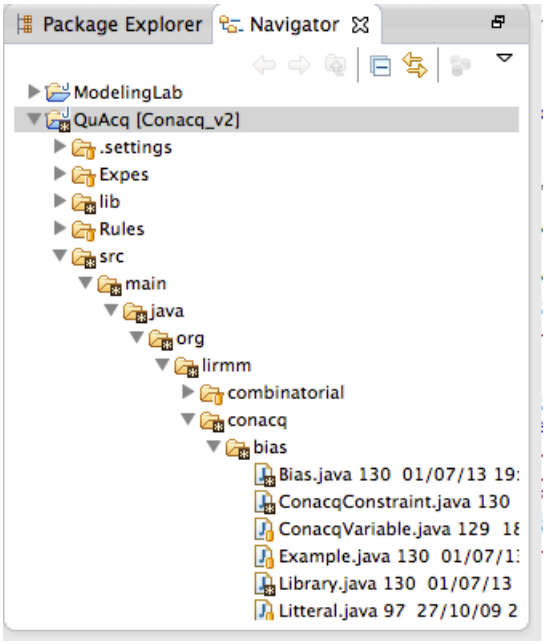
Sauf indication contraire, les fichiers sources de vos programmes se trouveront dans l'espace de travail. Il est donc important de se souvenir de l'emplacement de ce dernier pour accéder aux sources (par exemple, pour les transporter, les copier, etc.).



Si vous cochez la case *« Use this as the default and do not ask again »* eclipse ne vous posera plus cette question (mais il y a toujours un moyen pour changer ultérieurement l'espace de travail : *File>Switch Workspace>Other...*).

Le workspace est l'entité qui permet de conserver les projets et leur contenu.
Physiquement c'est un répertoire du système d'exploitation qui contient une hiérarchie de fichiers et de répertoires. Il y a d'ailleurs un répertoire pour chaque projet à la racine du workspace. Il est possible de parcourir cette arborescence et d'en modifier les fichiers avec des outils externes à Eclipse.

Le workspace contient tous les éléments développés pour le projet : il est possible de créer, de dupliquer, de renommer ou de supprimer des éléments. Ces opérations de gestion sont réalisées dans la vue Navigateur à gauche.



1.2 Création d'un nouveau projet

Pour créer un projet, il suffit de faire *File>New>Java Project*. Vous allez obtenir le panneau ou le wisard *New Java Project* où, au minimum, vous devez donner un nom à votre projet puis cliquer sur *Finish*.

Sinon, pour une procédure complète, vous pouvez faire *File>New>Project* puis :

— Choisissez Java Project (Next)

— Nommez le projet . Eclipse crée dans le workspace un répertoire de ce nom.

— Choisir *Create Project in workspace*

— Dans la partie **JRE**, vérifiez que la **JRE** est bien 1.7 ou 1.8. Si ce n'est pas le cas, il y a peut-être un problème avec votre variable système **JAVA_HOME**. Quittez Eclipse et résolvez le problème avant de relancer Eclipse.

— Choisir *Create separate source and output folders* afin de séparer vos fichiers sources (d'extension .java) des fichiers de bytecode (d'extension .class), qu'en cas de perte on peut toujours refaire. Eclipse va créer dans votre projet un répertoire *src* et un répertoire *bin*.(Next)

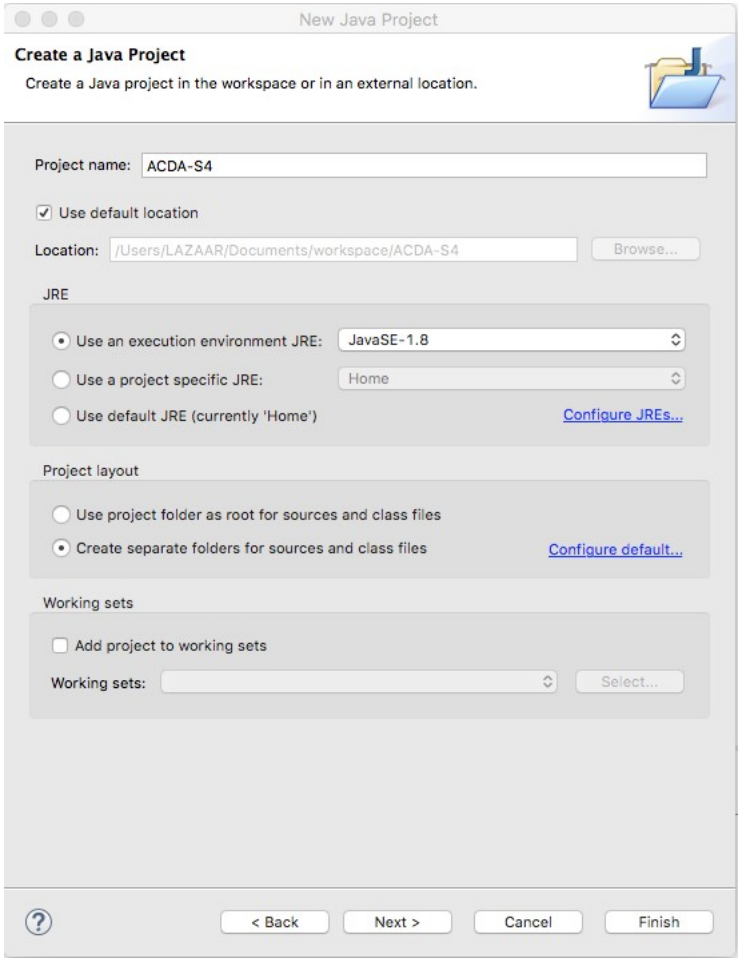
— Vous arrivez à des configurations fines qu'il n'est pas nécessaire de modifier pour l'instant. (Finish)

Avant de lancer Eclipse, il faut parfois positionner une variable d'environnement **JAVA_HOME** vers le répertoire d'installation de Java 1.7 ou 1.8 (echo $JAVA_HOME). Ce répertoire est souvent à l'intérieur de */usr/local/* et s'appelle *jdk1.(...)*). Si ce n'est pas le cas, positionnez-la (`export JAVA_HOME=chemin` où `chemin` est le chemin menant au répertoire d'installation de Java) et rajoutez cette ligne dans votre **.bashrc**, ce qui permettra de ne pas avoir à retaper cette ligne à chaque lancement d'Eclipse.

Dans l'explorateur de gauche, on voit le nouveau projet avec un répertoire *src*, et la référence à la librairie système **JRE**.

1.3 Création d'une classe

Dans l'explorateur de gauche, sur l'icone *src*, faire un clic droit *>new>class*. S'ouvre un wizard *"New Java Class"*.



— ne pas changer le *source folder* qui doit être correct,

— donner le nom du package dans lequel devra être votre classe,

— nommer la classe (*HelloWorld* par exemple)

— ne pas toucher à la partie interfaces (c'est inutile pour le moment)

— cocher la case *main* si vous souhaitez mettre un main dans la classe.

Cela crée dans l'Explorer le package, avec la classe à l'intérieur. Dans l'*outline* (à droite), s'ouvre un explorateur sur la classe faisant apparaître ses propriétés. Au centre, se trouve le code source de la classe. Vous pouvez contrôler ce qui a été généré (notez qu'il y a un embryon d'annotation pour la génération de la documentation). Notez un commentaire TODO qui signale qu'il faut compléter le code de la méthode main. Les TODO sont générés automatiquement ou ajoutés par le programmeur.

L'ensemble des tâches à faire peut être visualisé(menu *Windows >Show>views >Tasks*).

1.4 Quelques manipulations du code Java

Vous pouvez maintenant écrire le code de la classe. Vous remarquerez qu'il est directement indenté. Si vous copiez du code non indenté ou si vous voulez rafraîchir l'indentation : sélectionner la partie concernée, clic droit *>source >correct indentation* (ou CRTL +I).

Eclipse corrige quelques erreurs de compilation. Créer une variable *clavier* de type *Scanner* :

— une petite croix apparaît en début de ligne pour signaler un problème (qui est explicité en passant la souris sur la croix).

— une petite lumière signale une solution.

— cliquer sur la lumière (clic gauche) et choisir la solution appropriée (ici : `import java.util.Scanner`).

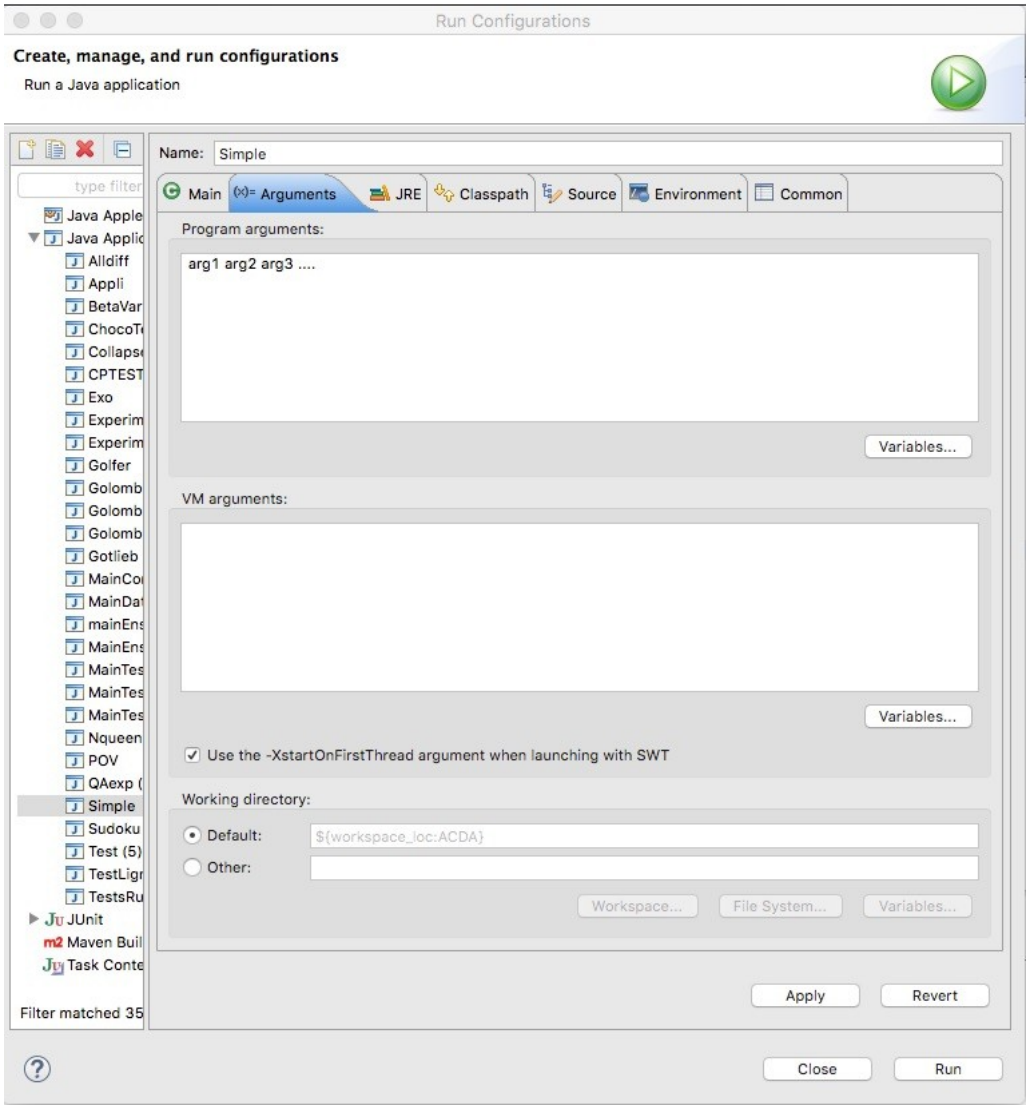
Complétion sémantique. Dans le corps de votre méthode main, tapez clavier. (c'est-à-dire *"clavier"* suivi de '.'). La liste des méthodes définies pour l'objet clavier est proposée ; il suffit de choisir celle qui nous intéresse (par exemple `nextInt`) en double-cliquant dessus.

Eclipse nous permettra par la suite d'ajouter automatiquement beaucoup d'autres éléments dans le code et de le modifier facilement, par exemple pour renommer des éléments.

1.5 Exécution d'un programme

Finissez de développer votre premier programme. Pour exécuter le programme (qui contient un main), choisir *menu>run>run as>java application* (ou utiliser l'icone avec la flèche blanche dans un disque vert).

Si votre programme nécéssite de passer des arguments en entrée, il vous faut configurer la vue *menu>run>run as>run configurations...* :



2 Débogage du code Java avec Eclipse

Un programme « bogué » est un programme qui ne donne pas les résultats qu'il devrait. « Déboguer » un programme c'est chercher les erreurs de programmation à l'origine de tels dysfonctionnements. Pour aider le programmeur dans cette recherche, eclipse offre un mode debug permettant, entre autres choses :

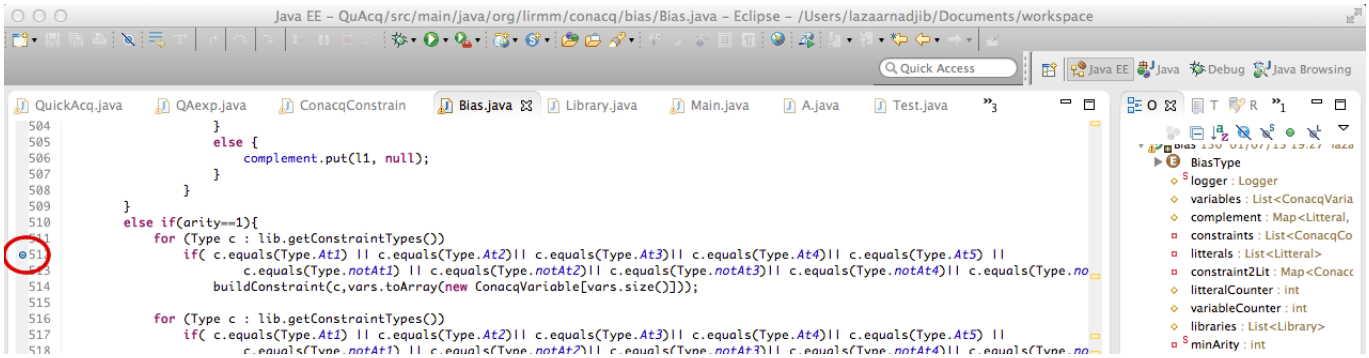
— la pose de marques, appelées **points d'arrêt**, sur des lignes du programme source, de telle manière que l'exécution s'arrêtera lorsque ces instructions seront atteintes ;

— lors de tels arrêts, l'examen des valeurs qu'ont alors les variables locales et les membres des objets ;

— à partir de là, l'exécution du programme pas à pas (c'est-à-dire ligne par ligne).

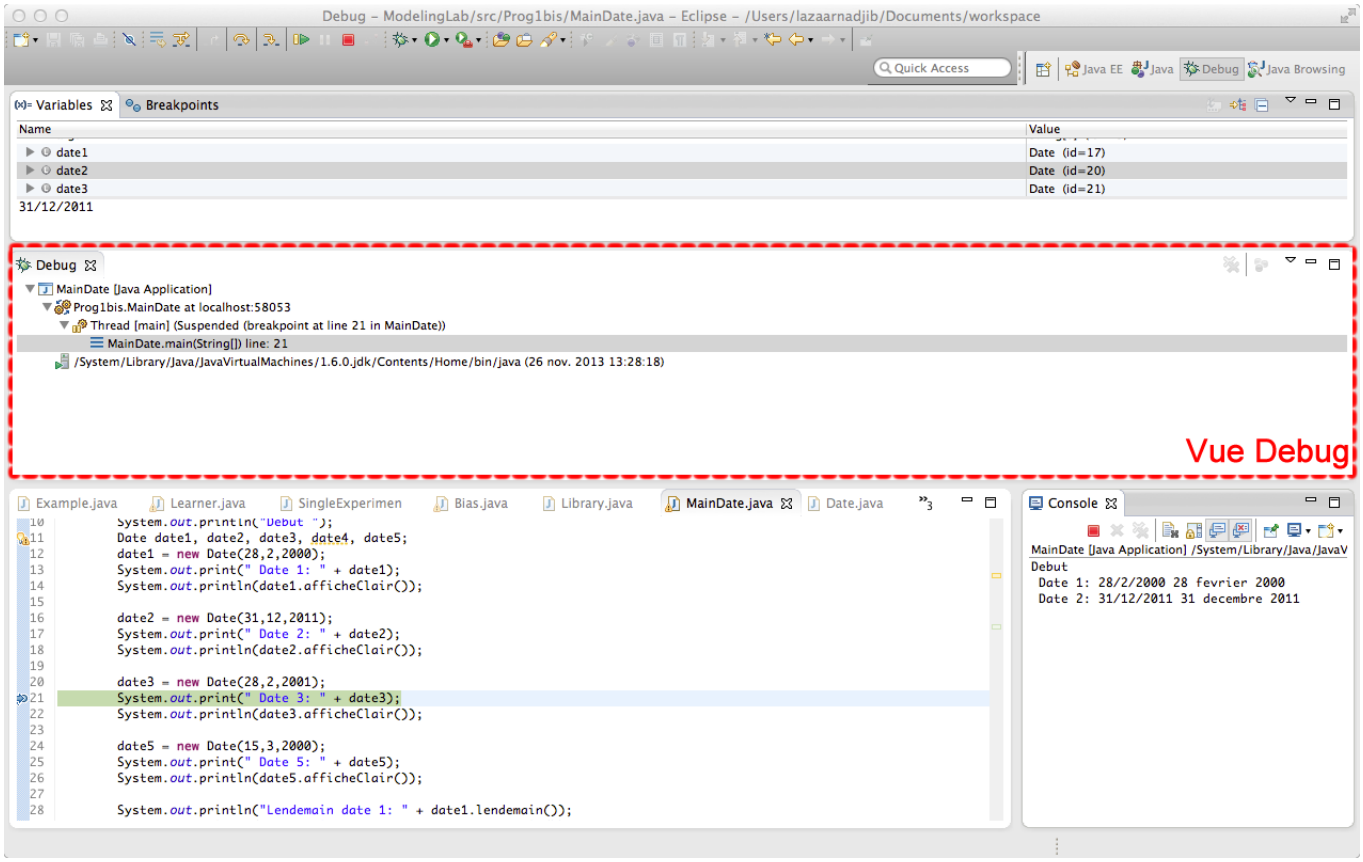
2.1 La perspective "Debogage"

Pour déboguer simplement un programme, il suffit de poser un point d'arrêt au début de l'endroit qu'on souhaite examiner en détail. Pour cela, il faut double-cliquer dans la marge, à gauche de la ligne en question, ce qui fait apparaître un disque bleu (cerclé de rouge dans la figure suivante) qui représente le point d'arrêt.



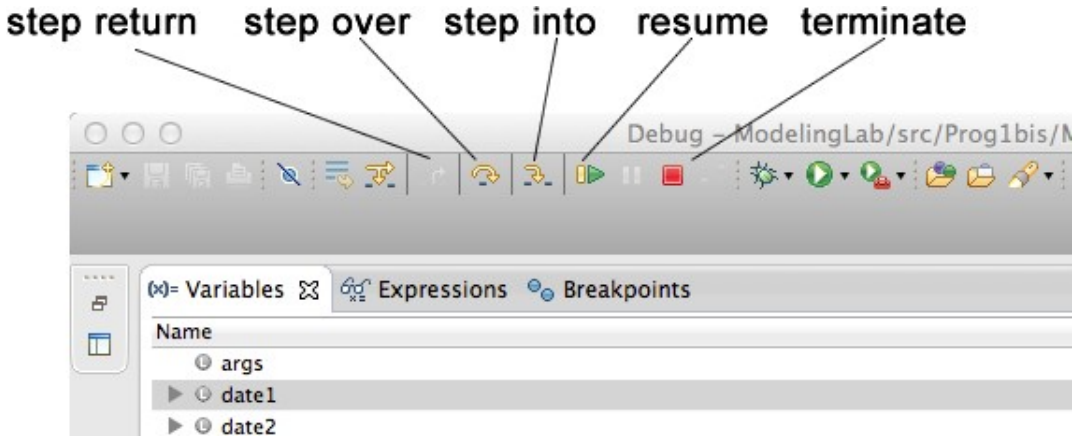
Il faut ensuite lancer le débogage, à l'aide du bouton à gauche de celui qui lance l'exécution, représentant une punaise (bug)

L'exécution est alors lancée et se déroule normalement jusqu'à atteindre le point d'arrêt. Eclipse demande alors la permission de changer de perspective (ensemble et disposition des vues montrées) et adopte l'apparence de la figure suivante :



2.2 La vue "Debug"

La vue Debug, au milieu de la fenêtre, montre la pile d'exécution, c'est-à-dire, pour chaque thread, l'empilement des méthodes qui se sont mutuellement appelées (méthodes commencées et non terminées). Dans la figure précédente, par exemple, on attire notre attention sur la méthode MainDate.main, plus précisément la ligne 21 du fichier source, dans laquelle l'exécution est arretée.



En haut de cette vue se trouvent des boutons très utiles. Parmi les principaux :

— *step return* : relancer l'exécution normale, jusqu'à la fin de la méthode dans laquelle on est arrêté et le retour à la méthode qui a appelé celle-ci ;

— *step over* : faire avancer l'exécution d'une ligne. Si cette dernière contient un appel de méthode, ne pas détailler l'activation de celle-ci, c'est-à-dire considérer l'appel comme une instruction indivisible (raccourci F6) ;

— *step into* : avancer l'exécution d'une ligne. Si un appel de méthode est concerné, détailler son activation, c'est-à-dire aller dans la méthode et s'arrêter sur sa première ligne (raccourci F5) ;

— *resume* : relancer l'exécution normale, jusqu'à la fin du programme ou le prochain point d'arrêt ;

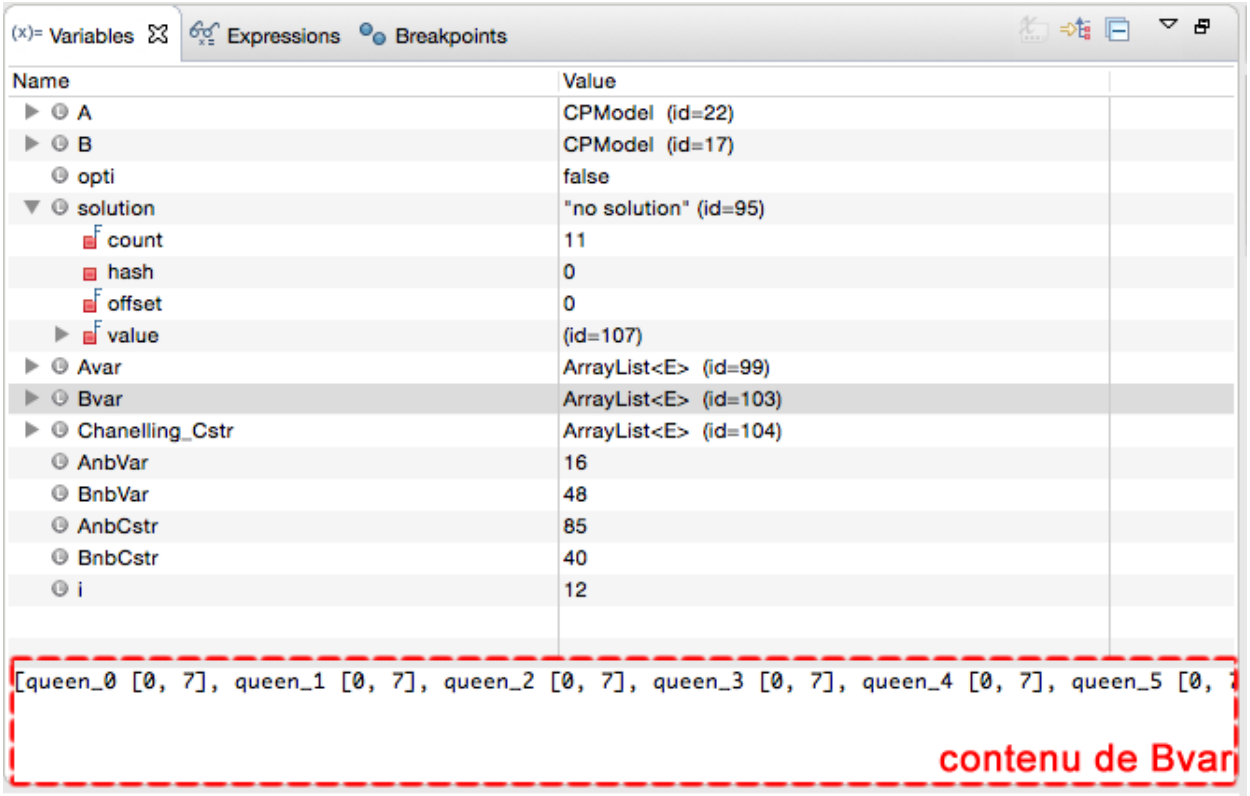
— *terminate* : terminer l'exécution.

En haut et à droite de la fenêtre principale se trouvent les vues Variables et Expressions. La première affiche les valeurs courantes des variables locales de la méthode en cours, la deuxième affiche les valeurs courantes des expressions sélectionnées avec la commande Watch (cliquer avec le bouton droit sur l'expression à surveiller).

Le debogueur d'eclipse possède bien d'autres commandes très puissantes, comme les points d'arrêt conditionnels et la possibilité de modifier les valeurs des variables du programme.
Prenez un peu de temps pour les explorer.

2.3 La vue "Variables"

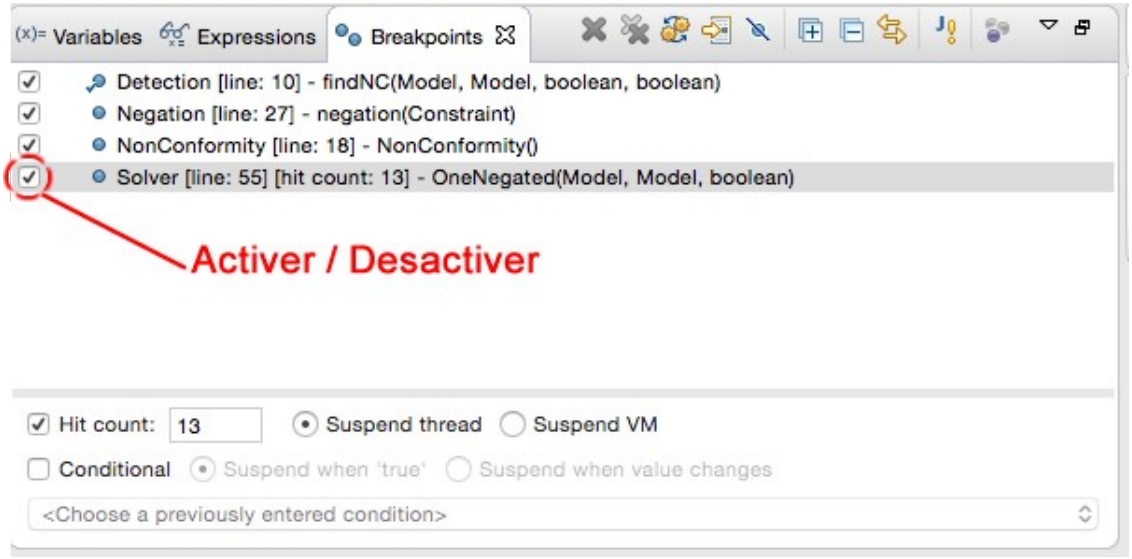
Cette vue permet de visualiser les valeurs des variables utilisées dans les traitements en cours de débogage. Ces valeurs peuvent être unique si la variable est de type primitive, par exemple la variable booléenne "opti" qui a la valeur "false", ou former d'une arborescense contenant chacun des champs si la variable est un objet, par exemple la variable "Bvar" de type ArrayList et dont le contenu est affiché :



2.4 La vue "Points d'arrêts"

La vue "Points d'arrêts", ou "breakpoints", permet de recenser tous les points d'arrêts définis dans l'espace de travail. Dans l'exemple suivant, nous avons quatre breakpoints :

1. Dans la fonction "findNC" à la ligne 10 de la classe "Detection".
2. Dans la fonction "negation" à la ligne 27 de la classe "Negation".
3. Dans le constructeur "NonConformity()" à la ligne 18 de la classe "NonConformity".
4. Dans la fonction "OneNegated" à la ligne 55 de la classe "Solver".



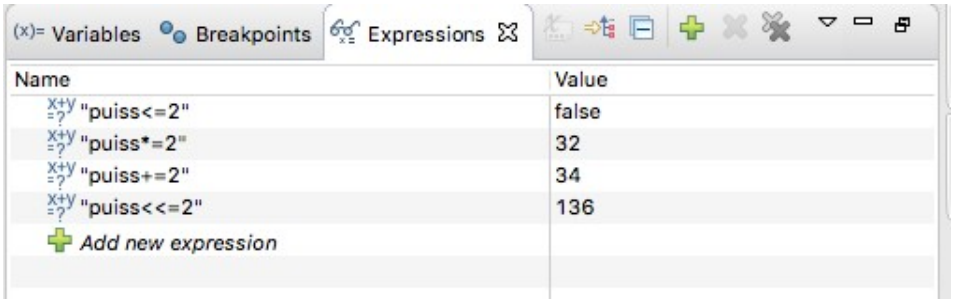
L'option "Nombres d'occurences" ("hit count" dans la figure) permet de préciser le nombre de fois ou le point d'arrêt exécuté sera ignoré avant qu'il n'intérrompe l'exécution. Ceci est particulièrement pratique lorsque l'on débogue du code contenu dans une boucle et que l'on connait l'itération qui pose problème.

Il est également possible d'activer ou de désactiver un point d'arrêt (voir figure).

Un double clic sur un des points d'arrêts permet d'ouvrir l'éditeur de code directement sur la ligne ou le point d'arrêt est défini.

2.5 La vue "Expressions"

Comme l'illustre la capture d'écran, cette vue vous permettra de renseigner des expressions qui seront ensuite réévaluées à chaque fois que vous avancerez d'un pas dans le débogage.



2.6 Revenir à l'affichage classique

Pour revenir à l'affichage classique, il suffit de cliquer sur la perspective "Java" en haut à droite comme le montre la figure suivante :


Java EE
<Java>
Debug
Java Browsing
Team Synchronizing
C/C++
Java

Exercice 1

Soit le programme suivant qui prend en entrée un entier naturel n et affiche les puissances suivantes : $2^0 \ldots 2^n$.

```
1  public class Puissance {
       static int traiterArguments(String[] args) {
3          int res = 0;
           if (args.length != 1) {
5              System.err.println("Nombre d'arguments incorrect");
               System.exit(2);
7          }
           try {
9              res = Integer.parseInt(args[0]);
               if (res < 0) {
11                 System.err.println("l'argument doit etre un entier naturel");
                   System.exit(2);
13             }
           } catch (NumberFormatException e) {
15         System.err.println("l'argument n'est pas un entier");
           System.exit(2);
17     }
       return res;
19 }
   static int[] puissances(int n) {
21     int puiss = 1;
       int[] results= new int[n];
23     for (int i = 0; i <= n; ++i) {
           results[i]=puiss;
25         puiss <<= 2;
       }
27     return results;
   }
29 public static void main(String[] args) {
       int n = traiterArguments(args);
31     int[] tab= puissances(n);

33     for(int i=0; i<tab.length; i++)
               System.out.println(tab[i]);
35
   }
37 }
```

Question 1 • Lancez le programme "Puissance" sous Eclipse avec les valeurs suivantes : -1, 0, 5, "test".

Question 2 • Définissez cinq cas de tests de la fonction "`puissances`".

Question 3 • Est ce que le programme en question répond à la spécification de départ ?

Question 4 • Deboguez le code avec l'argument 10 en entrée :

1. Demandez au debogueur de stopper à la ligne **28**.
2. Consultez le contenu du vecteur `args` et s'assurer que la valeur 10 est bien passée en entrée.
3. Faites avancer l'exécution pas à pas.
4. Ajoutez et analysez les expressions `puiss<=2` ; `puiss«=2` ; `puiss*=2` ; `puiss+=2` .
5. Localisez la faute et proposer une correction.
6. Relancez les tests sur "ecrirePuissances".