Introduction au débogage avec Eclipse

C. Pain-Barre et H. Garreta

IUT INFO Année 2006-2007

1 Notion de debogage

Tout développeur est confronté un jour à un programme qui "plante" pour une raison apparemment inexpliquée. Dans ce cas, il y a plusieurs solutions :

- passer en revue le code en espérant trouver l'erreur
- garnir le code d'affichages divers et variés (`System.out.print...`) afin de tracer son exécution et de déterminer la ligne ou la méthode qui pose problème. Bien que largement utilisée, cette solution est fastidieuse et lorsque le problème est résolu, il faut ensuite supprimer ces affichages[1] ;
- utiliser un débogueur qui est un programme qui exécute le code et permet de suspendre son exécution, la reprendre, afficher le contenu de variables, etc.

C'est cette dernière solution que nous allons étudier avec le débogueur intégré à Eclipse.

2 Lancer le débogage

Le lancement du débogage se fait en utilisant le bouton  ou via le menu *Run* et en choisissant l'une des actions *Debug*. Parmi les choix proposés, il y a le choix *Debug...* qui permet notamment de spécifier des arguments au programme, qui seront alors récupérés dans le tableau `String[] args` de la méthode publique `main()`, de la même manière qu'avec le choix *Run...* du menu *Run*. Notons que si des arguments ont déjà été entrés par *Run...*, ils sont alors aussi utilisés pour le débogage.

Prenons le cas du programme très simple (et correct) suivant :

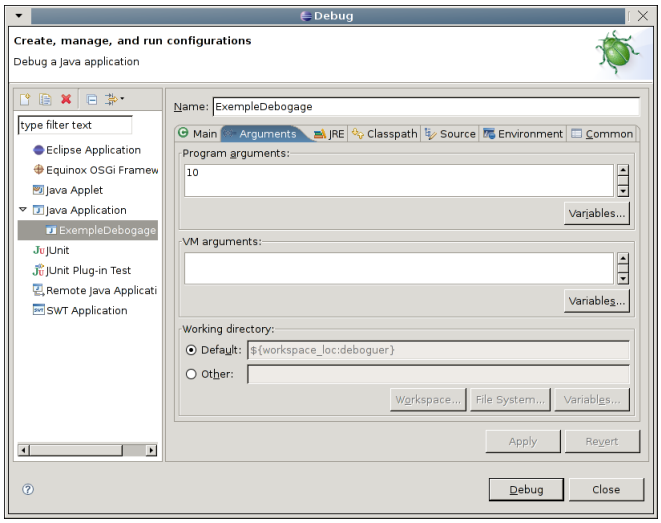
```
public class Simple {

    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        int puiss = 1;
        for (int i = 0; i <= n; ++i) {
            System.out.println(puiss);
            puiss <<= 1;
        }
    }
}
```

Ce programme affiche toutes les puissances de 2, jusqu'à la puissance indiquée en argument.

[1] notons que de nombreux développeurs utilisent cette méthode en conditionnant l'affichage par le test de variables booléennes (comme DEBOGUER)

Si l'on veut déboguer ce programme, il faut lui spécifier un argument. Via le choix *Debug...*, on choisit ensuite l'onglet *Arguments* :

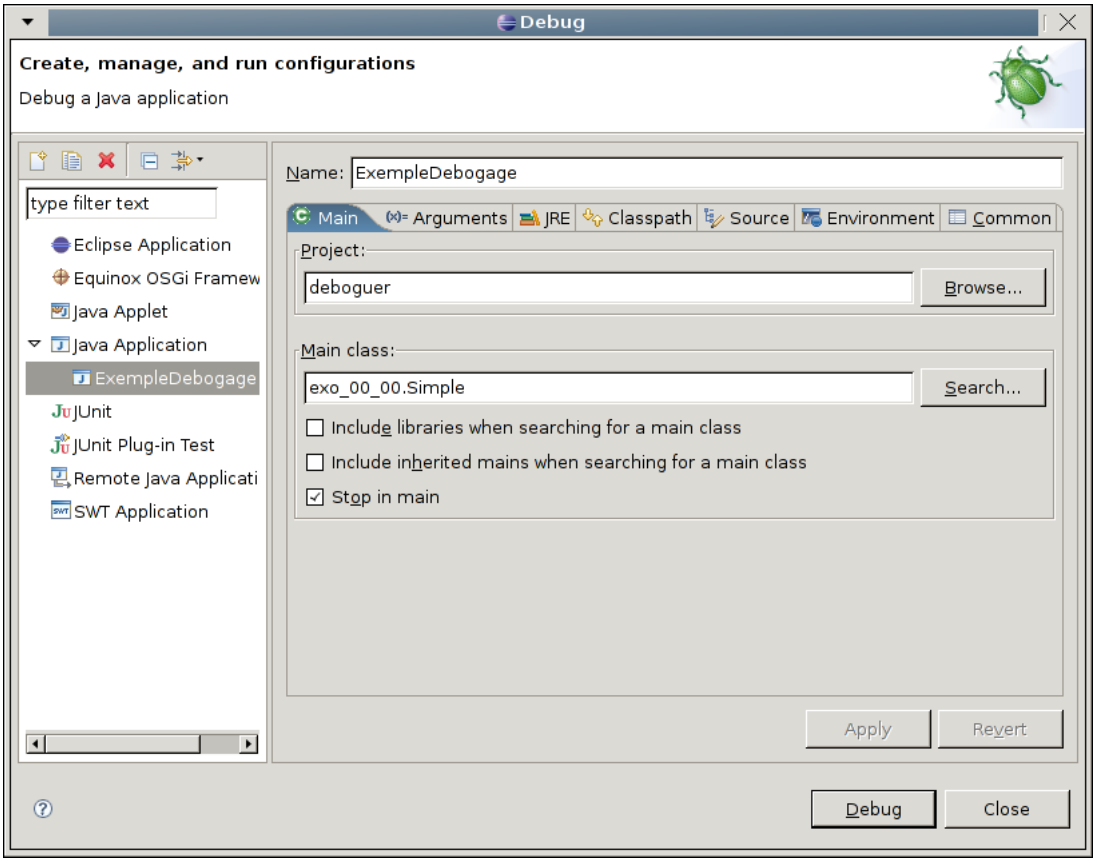


En cliquant ensuite sur *Debug*, le débogage du programme commence.

3 Deboguer dès l'exécution

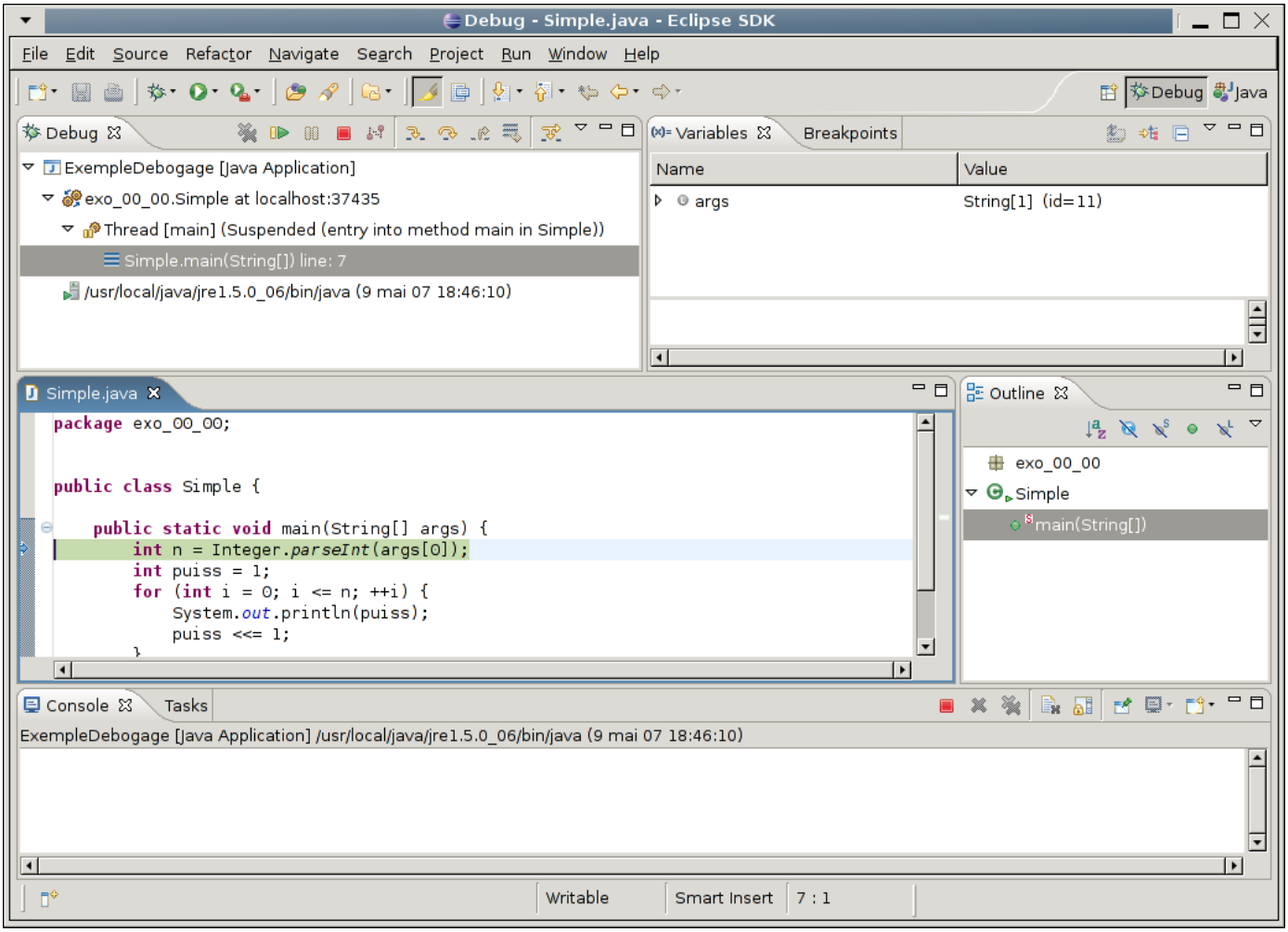
Pour le moment, le débogage du programme se limite à son exécution. Il s'arrêtera s'il provoque une exception non capturée, ou s'il se termine normalement. Pour observer son comportement, il faudrait demander au débogueur de suspendre son exécution à un ou plusieurs endroits choisis.

Une première possibilité est de demander, à partir de la configuration du débogage (choix *Debug...*), de suspendre l'exécution au début de l'exécution de la méthode `main()`. Pour cela, il faut cocher la case correspondante dans l'onglet *Main* :



puis cliquer sur *Apply*.

Dans ce cas, lorsque l'on lance le débogage, Eclipse ouvre la perspective *Debug* (après avoir demandé s'il faut l'ouvrir), dédiée au débogage :



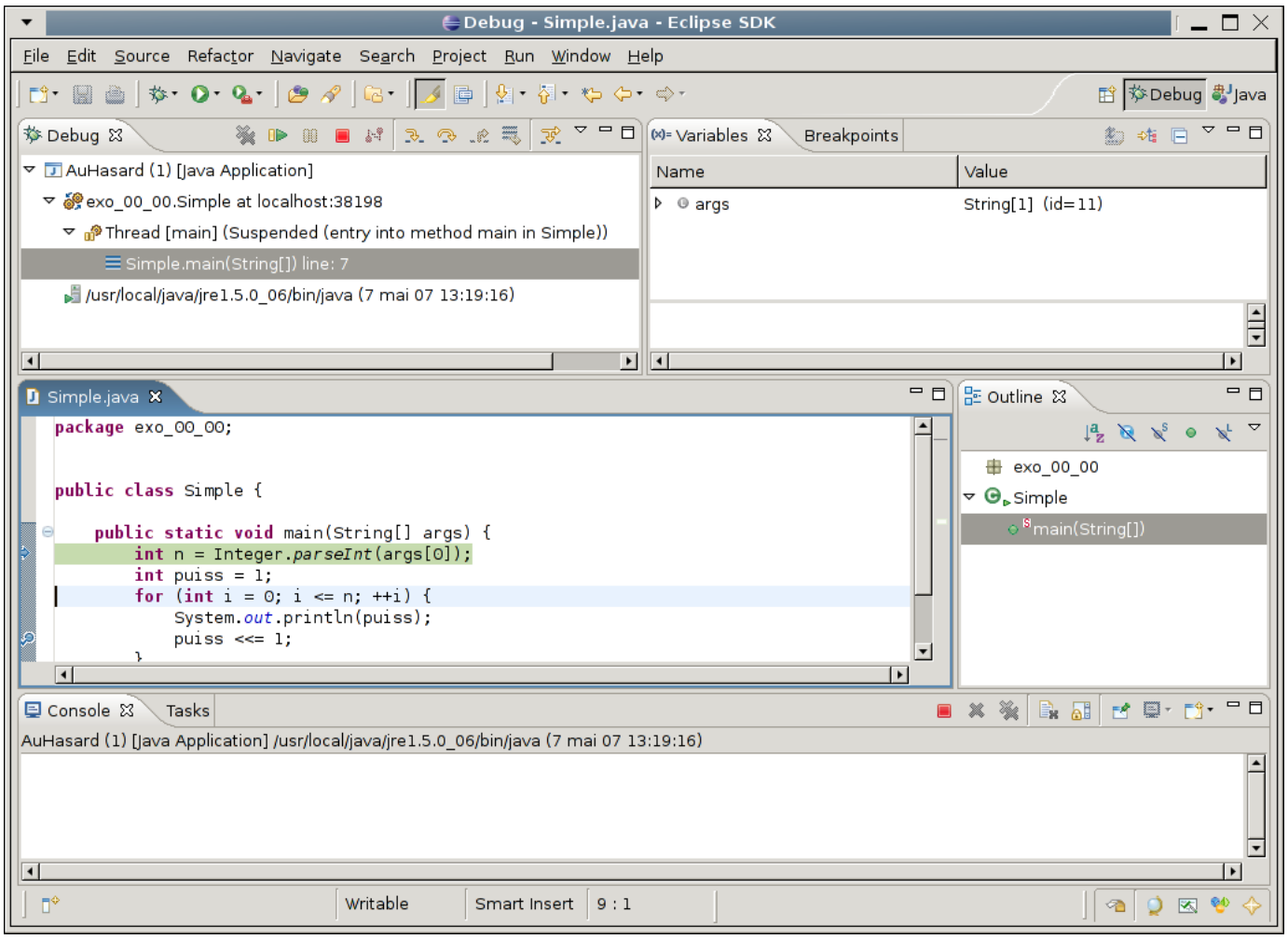
Avant de passer en revue quelques possibilités offertes par cette perspective, abordons une autre façon de suspendre l'exécution du programme en utilisant des **points d'arrêt** (*breakpoints*).

4 Les points d'arrêt

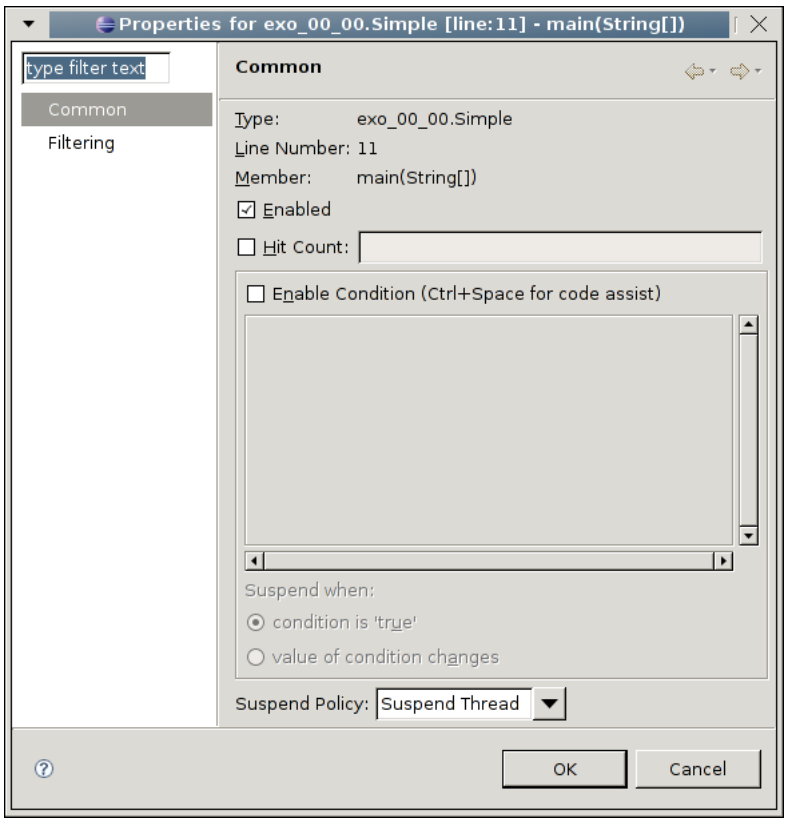
Comme leur nom l'indique (mal), les points d'arrêt permettent de suspendre l'exécution du programme. Ils peuvent être placés avant de lancer le débogage ou même pendant le débogage !

Pour placer un point d'arrêt, il suffit d'effectuer un clic droit sur la marge de gauche, sur la ligne sur laquelle on veut placer le point d'arrêt, et de choisir *Toggle Breakpoint* (effectuer un double clic gauche sur la marge produit le même effet). Un petit rond bleu ● est alors placé dans la marge.

Dans notre exemple, si l'on place un point d'arrêt sur la ligne `puiss <<= 1;`, alors la fenêtre ressemblera à :



Bien entendu, on peut placer des points d'arrêt à différents endroits du programme, mais on peut aussi préciser des conditions d'activation du point d'arrêt. Pour cela, il faut effectuer un clic droit sur le point d'arrêt et choisir *Breakpoint Properties. . .*. La fenêtre de ses propriétés est alors affichée :



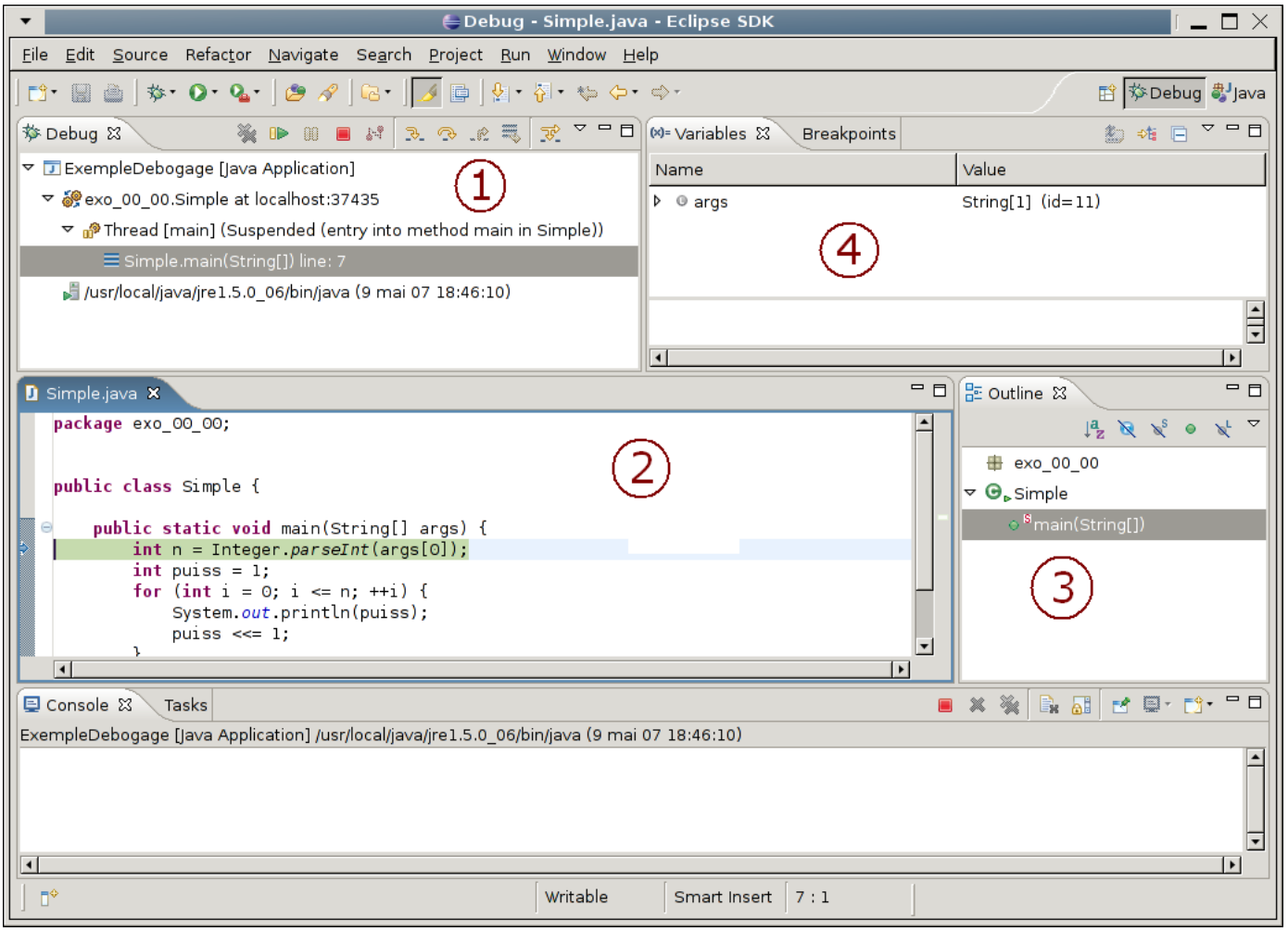
À gauche, on peut voir voir qu'il y a deux sous-menus : *Common* et *Filter*. Le menu *Filter* sert à préciser les threads qui seront affectés par le point d'arrêt. Dans la partie *Common*, on peut préciser :

- *Enabled* : doit être coché pour activer le point d'arrêt. Si on le désactive, le point d'arrêt sera ignoré (mais pas supprimé) et l'icône devient grisé . On obtient le même effet en effectuant un clic droit sur le point d'arrêt et en choisissant *Disable Breakpoint*.
- *Hit Count* : permet d'indiquer le nombre de fois que l'on peut passer sur le point d'arrêt sans que le programme (thread) soit suspendu
- *Enable Condition* : permet d'indiquer une condition sous la forme d'une expression booléenne, ou pour un changement de valeur d'une variable :
  - expression booléenne : par exemple `i > 4 && vect.size() != 3`. Il faut alors choisir *Suspend when condition is true*.
  - changement de valeur : par exemple `i` ou `vect.size()`. Il faut choisir *Suspend when value of condition changes*.

  Lorsqu'un point d'arrêt est conditionné, un point d'interrogation est accolé à son icône.

5 Gérer le débogage

Le débogage lancé, le thread exécutant le code est alors suspendu sur le premier point d'arrêt (dont la condition est satisfaite), ou en début de la méthode `main()`. En supposant que nous ayons demandé une suspension au début de la méthode `main()`, la perspective de débogage est :



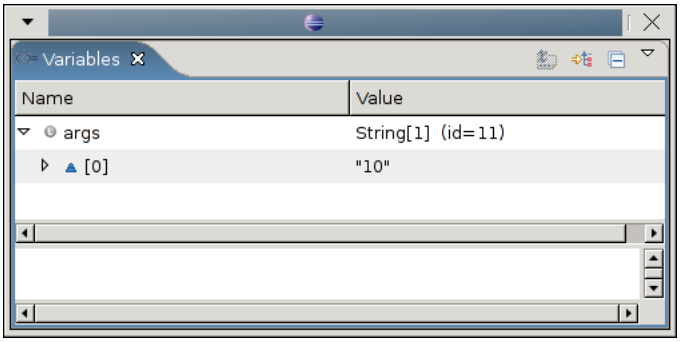
où le thread est suspendu sur la ligne indiquée par un fond vert pâle et une flèche dans la marge . Cette perspective est une composition de plusieurs vues :

- la vue *Debug* (①) qui indique le(s) thread(s) en cours et les lignes sur lesquelles ils sont suspendus
- la vue appelée *Display* (②) qui présente le code en cours de débogage (ici, `Simple.java`)

- la vue *Outline* (③) qui permet d'explorer les packages/classes
- la vue (④) composée de 2 onglets présentant chacun une vue :
  - ◇ la vue *Variables* qui permet de visualiser les variables, leur valeur et éventuellement leurs données membres
  - ◇ la vue *Breakpoints* qui affiche les différents points d'arrêts qui ont été placés

Toutes les vues sont détachables de la perspective afin de les présenter dans des fenêtres séparées et indépendantes. Cela se fait en effectuant un clic droit sur la barre de titre de la vue concernée et en choisissant *Detached*. D'autres vues peuvent être affichées par l'intermédiaire du menu *Window* → *Show View*. Notamment la vue *Expressions* qui permet d'afficher le résultat d'expressions.

À ce stade, on peut déjà observer dans la vue *Variables* qu'il existe une variable `args` dont la valeur est un tableau de 1 objet de type `String` (i.e. `String[1]`). En cliquant sur le triangle à gauche de la variable, on développe sa vue et on peut observer que le premier élément de ce tableau contient la chaîne "`10`", qui est le premier argument du programme :



5.1 Contrôle de l'exécution du programme

Plusieurs boutons de la vue *Debug* permettent de continuer, suspendre ou terminer l'exécution du programme :

- (*Resume*) pour continuer l'exécution du programme depuis l'instruction où il a été suspendu, jusqu'à rencontrer un point d'arrêt, être suspendu manuellement, ou terminer (normalement ou par une exception non capturée) ;
- (*Suspend*) pour suspendre l'exécution du programme là où il en est ;
- (*Terminate*) pour terminer l'exécution du programme (comme si l'on tapait CTRL-C).

D'autres boutons permettent de contrôler finement l'exécution du programme lorsqu'il a été suspendu :

- (*Step Into*) pour exécuter la ligne courante. Si celle-ci contient un appel à une méthode, alors le débogueur se placera sur la première ligne de cette méthode. Notons que le débogueur ne pourra explorer une méthode que si le code source lui est accessible ;
- (*Step Over*) pour exécuter la ligne courante et se placer sur la ligne qui suit (éventuellement en sortant d'une méthode). Ainsi, contrairement au *Step Into*, s'il y avait un appel de méthode, celle-ci aura été exécutée mais pas explorée ;
- (*Step Return*) pour retourner de l'appel d'une méthode. Ainsi le débogueur va exécuter toutes les instructions restantes afin de terminer l'appel courant de la méthode en cours d'exploration.

Enfin, il est possible de demander au débogueur d'exécuter toutes les instructions jusqu'à une ligne donnée (si tant est que cela soit possible) en se plaçant sur la ligne souhaitée et en tapant CTRL-R (qui est un raccourci pour *Run to Line* du menu *Run*).

> ⓘ Selon la configuration et la disponibilité des sources, le débogueur n'explore pas les constructeurs ni les méthodes des classes de l'API de Java.

5.2 Exemple

Afin d'illustrer un peu les possibilités du débogueur, compliquons un peu notre programme en y ajoutant des méthodes, et en le rendant faux par la même occasion :

```
public class Simple {

    static int traiterArguments(String[] args) {
        int res = 0;
        if (args.length != 1) {
            System.err.println("Nombre d'arguments incorrect");
            System.exit(2);
        }
        try {
            res = Integer.parseInt(args[0]);
            if (res < 0) {
                System.err.println("l'argument doit être un entier naturel");
                System.exit(2);
            }
        } catch (NumberFormatException e) {
            System.err.println("l'argument n'est pas un entier");
            System.exit(2);
        }
        return res;
    }

    static void ecrirePuissances(int n) {
        int puiss = 1;
        for (int i = 0; i <= n; ++i) {
            System.out.println(puiss);
            puiss <<= 2;
        }
    }

    public static void main(String[] args) {
        int n = traiterArguments(args);
        ecrirePuissances(n);
    }
}
```

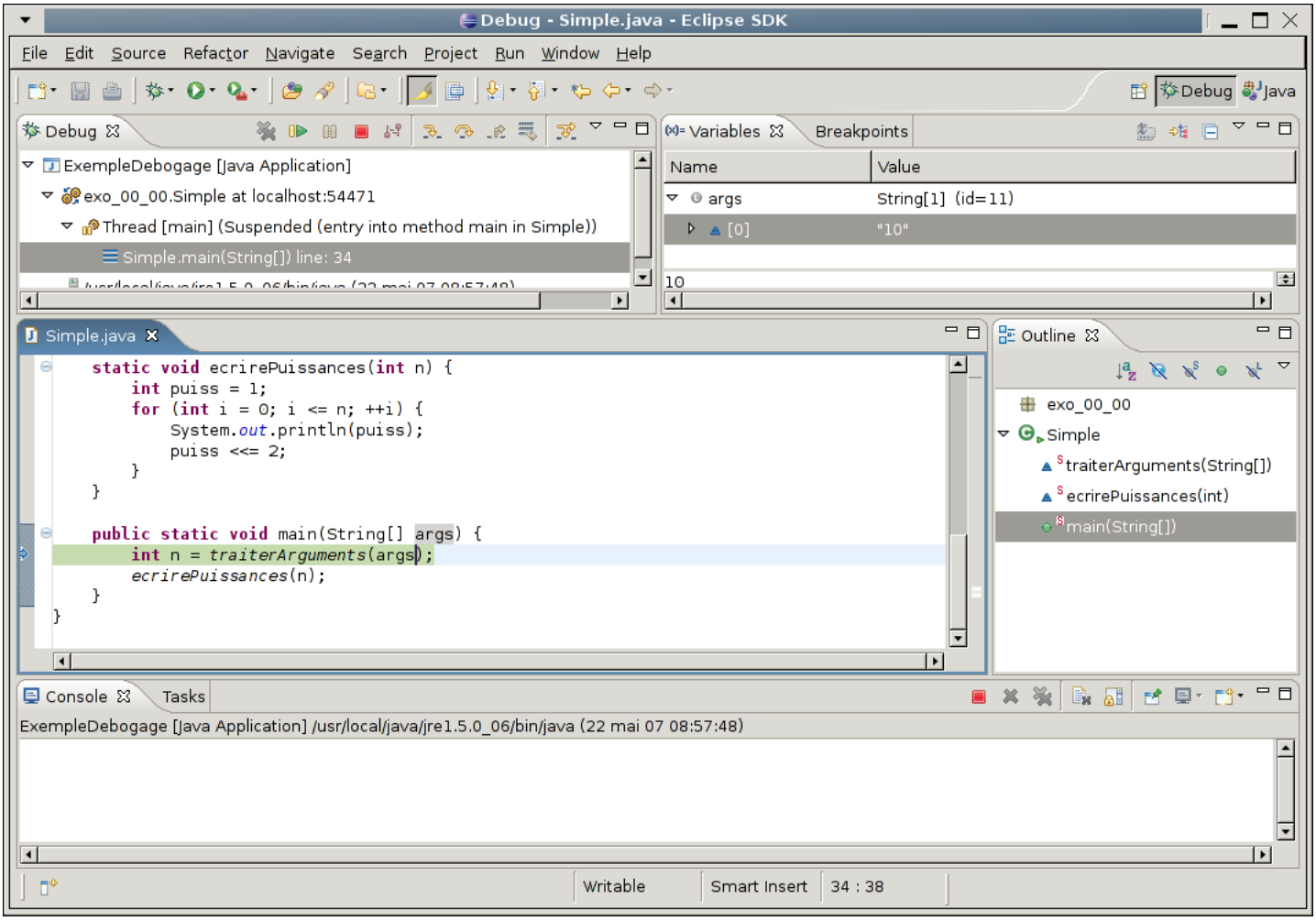
Si on l'exécute avec 10 comme argument, on obtient le résultat suivant :

```
1
4
16
64
256
1024
4096
16384
65536
262144
1048576
```

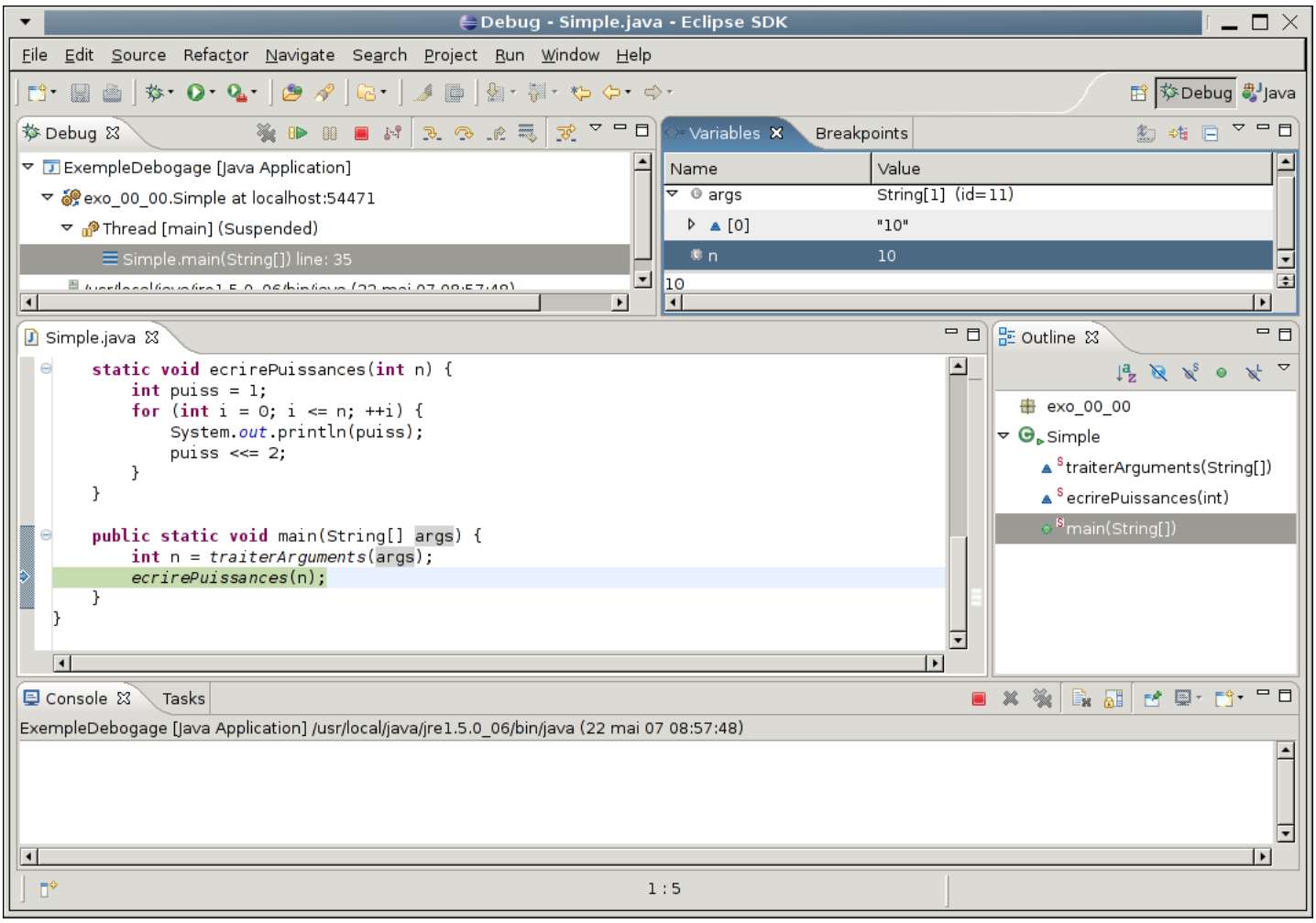
ce qui ne correspond pas aux 11 puissances de 2 de 2^0 jusqu'à 2^{10}...

Afin de trouver l'erreur, on va déboguer le programme.

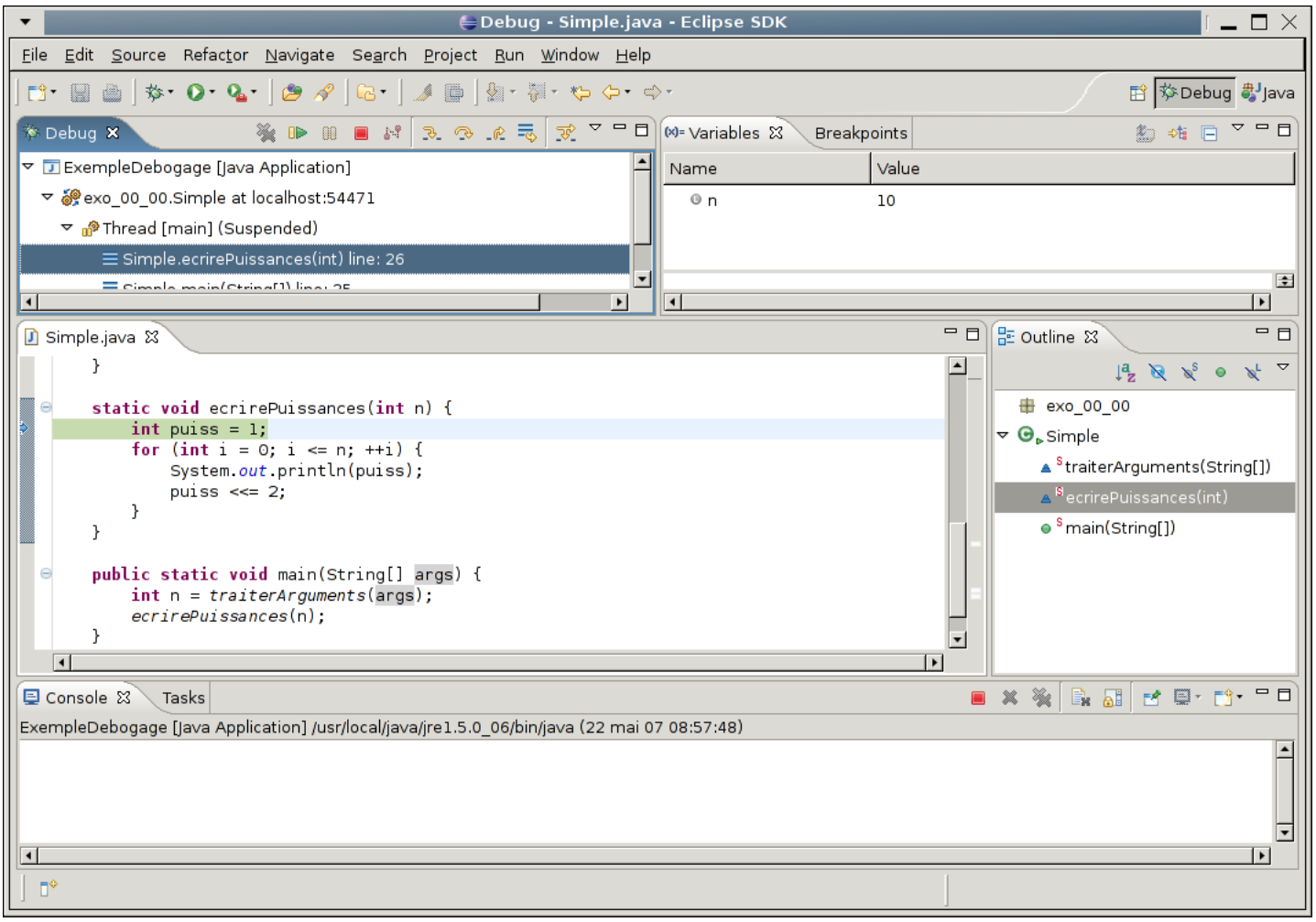
Dans un premier temps, on va demander au débogueur de stopper au début de la méthode `main()` par l'intermédiaire du menu *Debug...*, puis on lance le débogage. Le thread est alors suspendu. En regardant le contenu de la viariable `args` et de son premier élément, on vérifie que l'argument est bien la chaîne `"10"` (c'est déjà ça) :



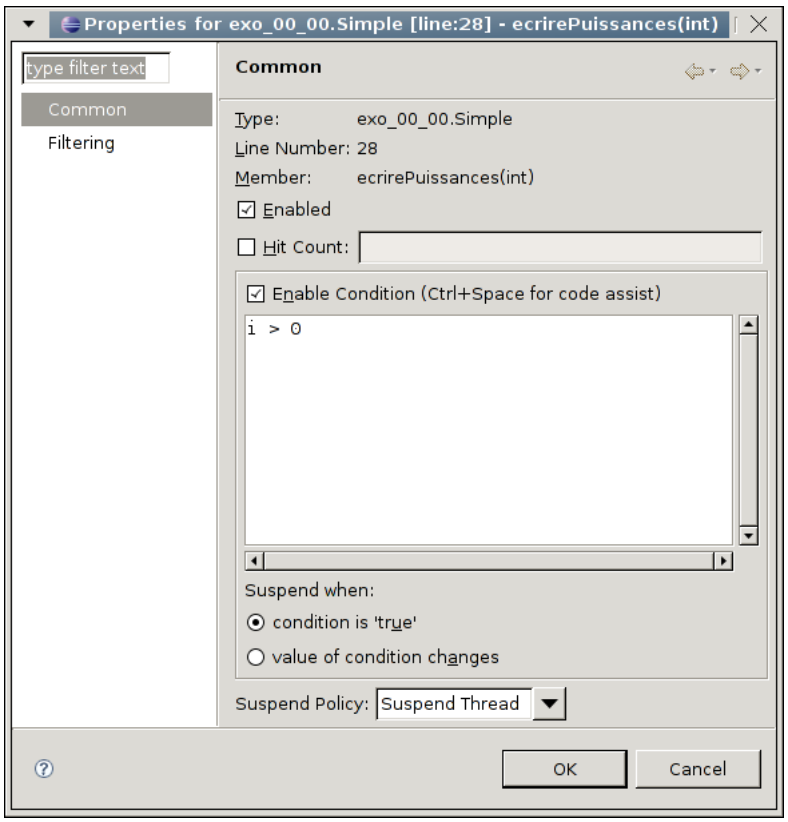
Puisque le programme affiche 11 nombres (dont 10 sont incorrects) on va supposer que la méthode `traiterArguments()` est correcte. On la saute donc en cliquant sur *Step Over* (). On arrive alors sur la ligne suivante et on observe que la variable `n` est apparue dans la vue *Variables* et qu'elle contient bien `10` :



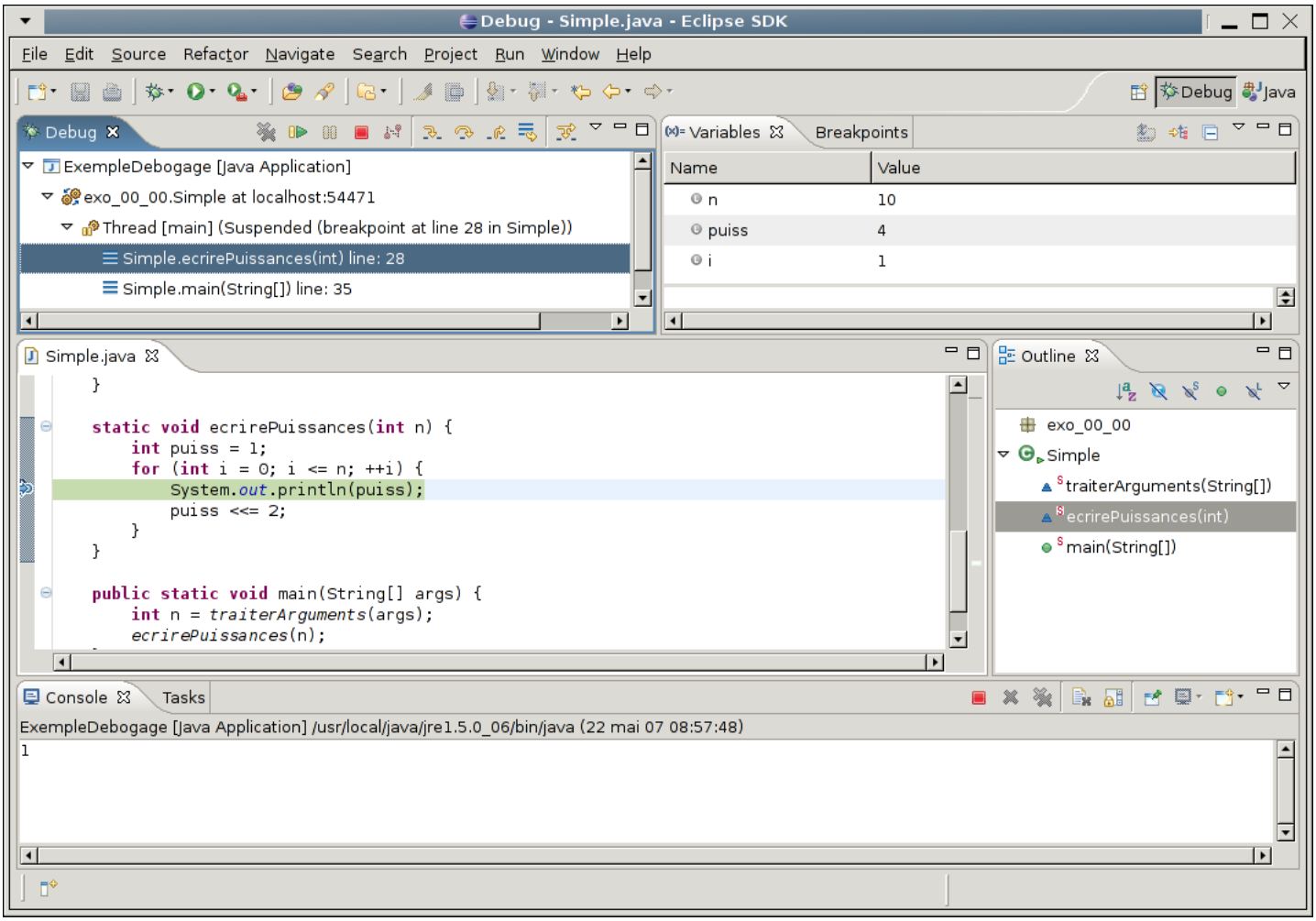
Le problème se situe donc (comme on pouvait s'en douter) dans la méthode `ecrirePuissances()`. On l'explore en cliquant sur *Step Into* () et l'on arrive sur la première ligne de cette méthode :



Puisque le premier chiffre est le bon (et pour utiliser un peu les fonctionnalités du débogueur), on va faire en sorte de passer l'exécution du premier tour de boucle. Pour cela, on va placer un point d'arrêt conditionnel sur la ligne `System.out.println(puiss);` en mettant comme condition que `i` doit être supérieure à 0 :



Puis, l'on clique sur *Resume* (▶) pour continuer l'exécution :



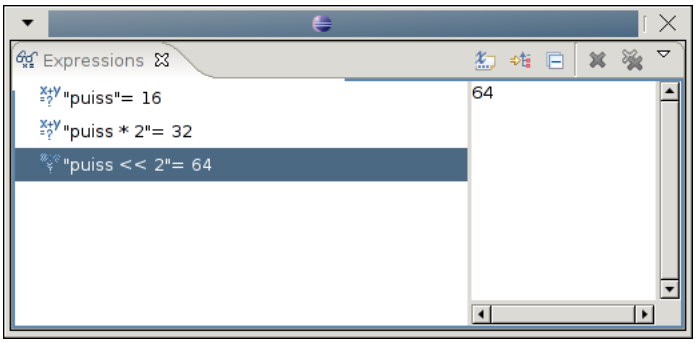
On remarque alors que :

- `i` vaut bien 1
- `1` a été écrit et correspond bien à 2^0
- `puiss` vaut (déjà) 4 (qui correspond à 2^2), alors qu'à ce stade il devrait valoir 2 (soit 2^1).

En cliquant une première fois sur *Step Over* (⤼), 4 est écrit sur la console et l'on se place sur la ligne `puis <<= 2;`. En cliquant à nouveau sur *Step Over* (⤼), cette ligne est exécutée et `puiss` passe de 4 (2^2) à 16 (2^4). L'instruction ne semble donc pas faire passer d'une puissance de 2 à la suivante mais à la suivante encore. Vérifions en faisant affciher le résultat des expressions suivantes (en demandant l'affichage de la vue *Expressions* par le menu *Window* → *Show Views* et en ajoutant une à une ces expressions, par un clic droit sur la vue et *Add Watch Expression*) :

- `puiss`
- `puiss * 2`
- `puiss << 2`

on obtient alors l'affichage suivant (ici réduit à une fenêtre) :



On voit bien que `puiss << 2` ne multiplie pas `puiss` par 2 mais par 4. C'est un mauvais mélange de `puiss * 2` et de `puiss << 1` qui elles le font. On a enfin trouvé cette mystérieuse erreur.

6 Conclusion

On aura pu, au long de ce petit exemple, se faire une idée de la facilité de débogage offerte par un tel outil. D'autres possibilités existent, notamment celle de pouvoir placer des points d'arrêt sur la levée d'exceptions. À vous d'en tirer le meilleur parti pour vos besoins.

