



IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

DevEnv – Java debugging (Eclipse version)

Software Development Engineering – IDL

Disclaimer: this lesson has been translated for you convenience using translation software powered by DeepL. Reasonable efforts have been made to provide an accurate translation, however, no automated translation is perfect nor is it intended to replace human translators. If you spot any error or weird translation, please tell us in order to fix it and to improve the document.

Objectives

- Learning to use the debugger in Eclipse
- Debugging a small program

Note: This tutorial focuses on Eclipse, but debuggers are available in most IDEs. A VSCode version of this lab session is also available. If you use a different IDE and feel comfortable with it, you can always try to adapt the tutorial to your IDE.

1 How does it work?

The debugger is a specific application that can be launched manually or automatically when an error occurs. When used manually, there are generally three steps:

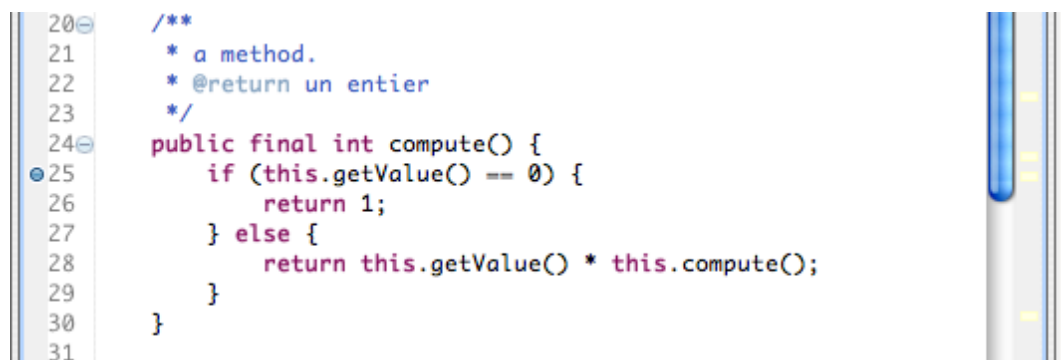


Figure 1: A breakpoint at line 25 in the `compute` method.

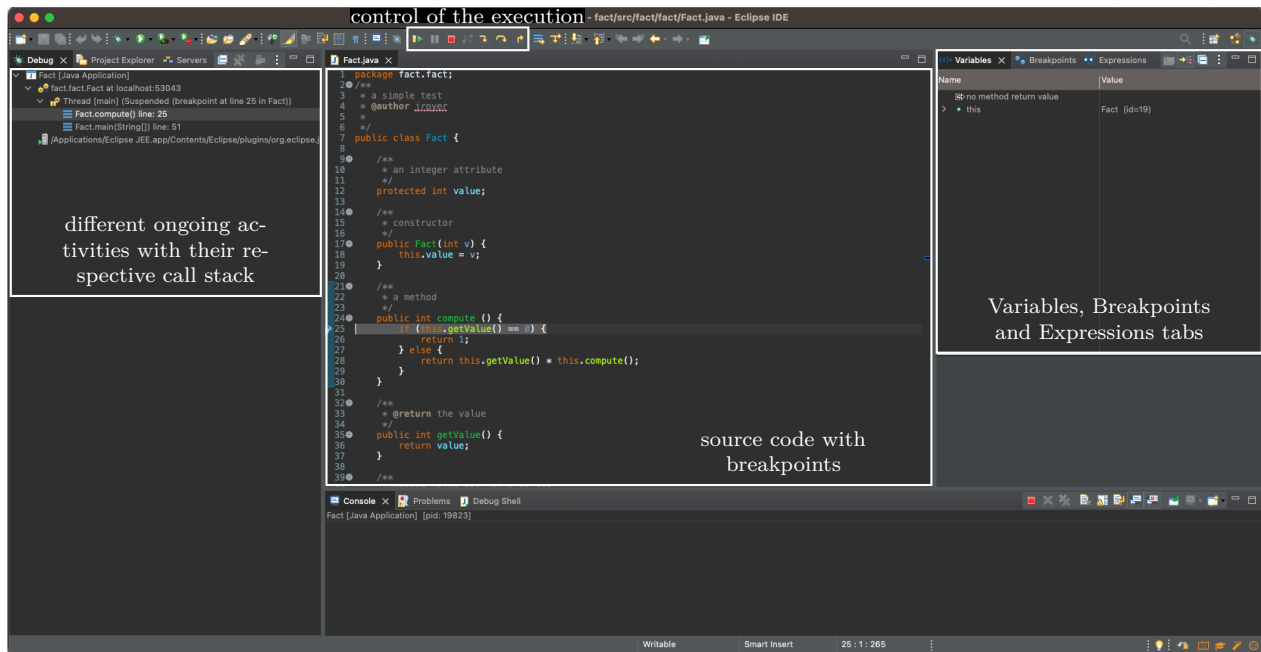


Figure 2: The debug view

1. Insert one or more *breakpoints* in the source code on the line you want to observe. See Figure 1 for an example. The breakpoint appears in the left column of the editor as a blue dot. Breakpoints are added by left-clicking in the column before the line of code where you want to set a breakpoint.

When using the debugger, the program will stop at this point and the user can then perform certain actions. The options available at a *breakpoint* are accessible via a context menu: *delete*, *disable*, or *set a condition*.

2. Next, launch the debugging application. To access **debug** mode, select the entry point of the application you want to observe and either use the **debug** context menu item or the toolbar button with a small insect icon:



3. After execution, if a breakpoint is reached, the debug perspective becomes active and various operations are possible.

Figure 2 shows this perspective:

- At the top, the toolbar contains several arrows that allow you to execute commands in step-by-step mode, among other things.
- At the top left, the various activities in progress and their stack of current calls: i.e., the various activities that led up to this stopping point.
- In the center, your source code with any breakpoints visible on this code in the left column.
- At the top right, three tabs:

- **Variables** allows you to observe objects and variables and set *watchpoints*.
 - **Breakpoints** shows breakpoints and watchpoints.
 - **Expressions** is a symbolic calculator that evaluates expressions in the current context.
- At the very bottom, the usual console.

2 A little test

1. Clone the IDL resources repository (<https://gitlab.imt-atlantique.fr/depinfo-public/idl-resources-practical-work>) and switch to the `devenv` branch using your new Git skills. Retrieve the `fact/src/fact/Fact.java` file, which defines a simple class for calculating the factorial of 3.
2. Execute it and observe a “stack overflow”.
3. Double-click in the left column of line 25: `if (this.getValue() == 0)`. If you see a blue dot appear, you’ve done it right because you’ve placed a breakpoint. If not, try again!
4. Start debug mode and use the **step over**  key several times.
5. What can we observe about the execution stack and the value of the attribute in the `this` object?
6. In the display view, you can view primitive values, objects and their attributes, or collections and their elements.
7. Deduce what is missing, correct the program, and check again with the debugger that the behavior is acceptable.

3 Explanations

In the debug view, there are three important buttons:

- *step into* : continues execution of the line and stops at the first line of code of the first method contained in the line. You can see a stack in the call stack. Allows you to see the code in depth.
- *step over* : executes the current line and stops at the next one. Without stacking.
- *step return* : executes the line and what follows until the next `return` statement in the method. Unstack.

4 Implementation

In the IDL resources repository, still in the `devenv` branch, find the directory named `bad4debug` containing a Petri net model for the MAPD common thread ("Fil Rouge"). However, this is not a good solution to your problem, just a way to experiment with the debugger. The result does not really match the predictions, and I cannot find my errors!

1. Retrieve the code and open it with Eclipse.
2. Try to find what is wrong using the debugger.
3. Correct and test.
4. Add your name and comments about any errors you find in a file you could commit into your own Git repository...

4.1 Your turn!

To begin with, the first mistake is easy to find.

1. Run `Test2Debug`, and you will get an index error in an array.
2. Place a breakpoint on line 106 of the `PetriNet` class.
3. Launch the debugger.
4. This is an opportunity to play around with the “steps” buttons, but clearly “step into” will take you through all the twists and turns of the code, including the standard library classes. *Step over* moves forward line by line, which is more interesting for you, and after a few clicks you should have an idea of the problem. Finally, *step return* allows you to move a little faster to reach the error.
5. After a few *step over* commands, check the value of “`t`”, understand it, and correct it.

But this time you notice that the message in the last line of the test is suspicious.

1. Why does the answer `true` seem wrong to you?
2. Place a breakpoint somewhere in the `testEffect` method.
3. Find what is wrong and correct the problem.
4. A careful reading of the code should have put you on the right track...

And there are surely many other errors, so feel free to look for them and correct them.

5 Going further

To go further, try experimenting with the following *optional* items.

- The debugger stops at a breakpoint or an uncaught exception.
- It is possible to set a breakpoint on an intercepted exception, an unintercepted exception, or both. To do this, use the *Add Java Exception Breakpoint* button  in the breakpoints tab.
- For a breakpoint, you can configure the number of occurrences before it is taken into account or an activation condition. To do this, edit the breakpoint properties or look at its description in the breakpoint view.
- The “Expressions” view allows you to calculate the value of an expression based on the variables in the processes being debugged.