# 2022/2023(1)
# IF184301 Object Oriented Programming

Lecture #3a

# Eclipse IDE: Debugging

Misbakhul Munir IRFAN SUBAKTI
司馬伊凡
Мисбакхул Мунир Ирфан Субакти

# Debugging: What's that?

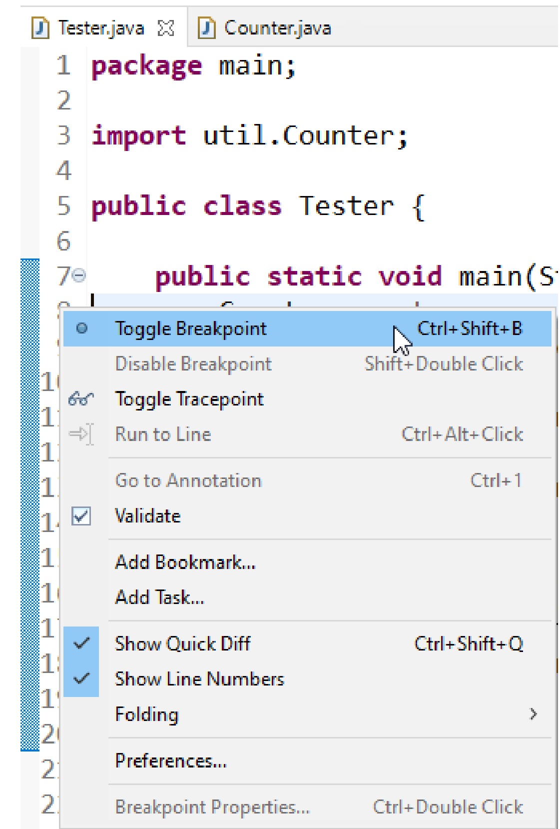
- *Debugging* allows you to run a program interactively while watching the source code and the variables during the execution.
- A *breakpoint* in the source code specifies where the execution of the program should stop during debugging. Once the program is stopped you can investigate variables, change their content, etc.
- To stop the execution, if a field is read or modified, you can specify *watchpoints*.
- *Breakpoints* & *watchpoints* → stop points

# Debugging support

- Eclipse allows you to start a Java program in *Debug mode*.
- Eclipse provides a *Debug perspective* which gives you a pre-configured set of *views*. Eclipse allows you to control the execution flow via debug commands.

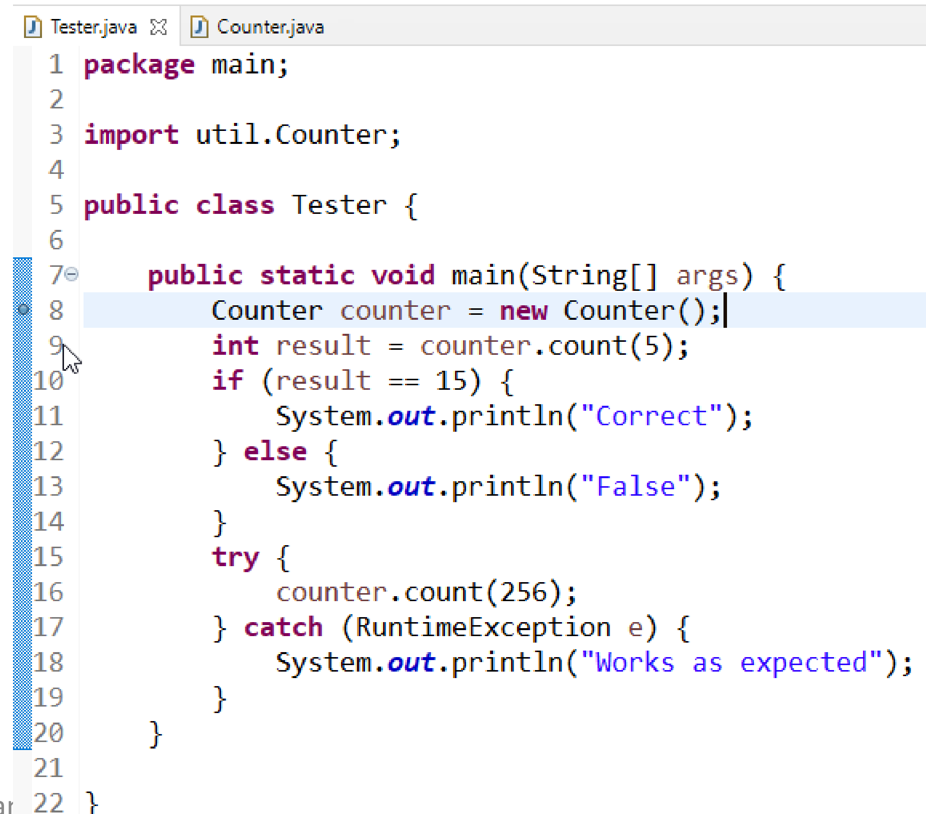
# Breakpoints: Setting up

- To define a breakpoint in your source code, right-click in the left margin in the Java editor and select *Toggle Breakpoint*. Alternatively you can double-click on this position.



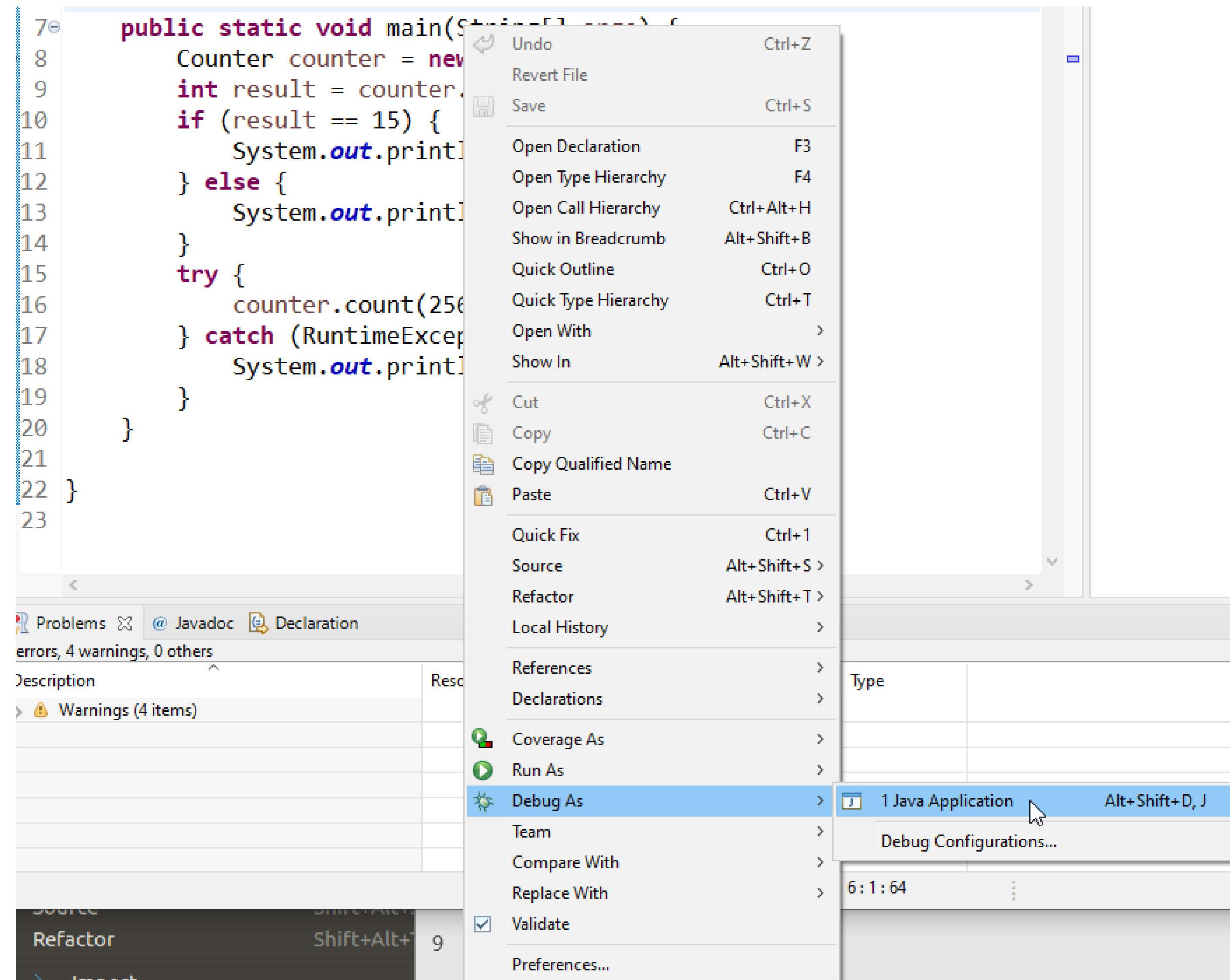
# Breakpoint: Example

- For example in the following screenshot we set a breakpoint on the line `Counter counter = new Counter();`



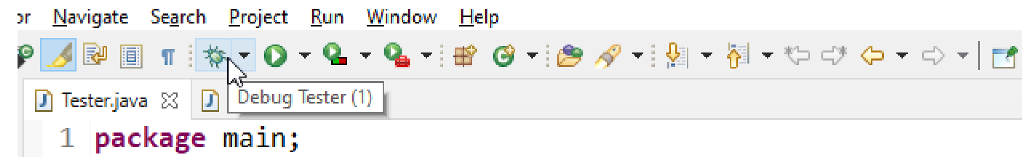
# Debugger: Starting

- To debug your application, select a Java file with a main method. Right-click on it and select **Debug As > Java Application**.



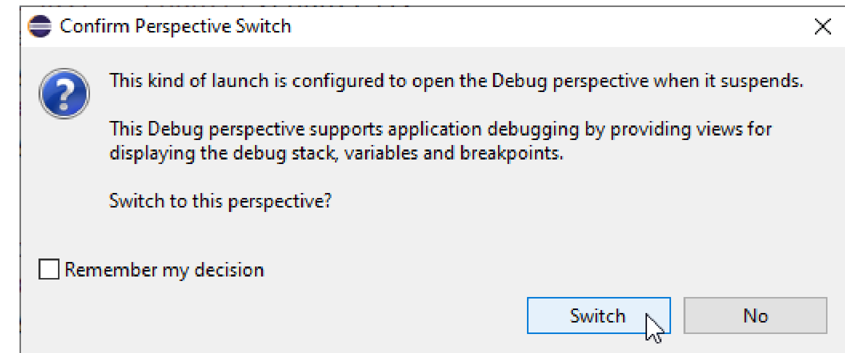
# Debug button

- If you started an application once via the context menu, you can use the created launch configuration again via the Debug button in the Eclipse toolbar.

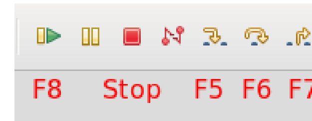


# Debug perspective

- If you have not defined any breakpoints, program will run normally. To debug the program you need to define breakpoints. Eclipse asks you if you want to switch to the *Debug perspective* once a stop point is reached. Answer *Switch* in the corresponding dialog. Afterwards Eclipse opens this *perspective*.



# Program execution

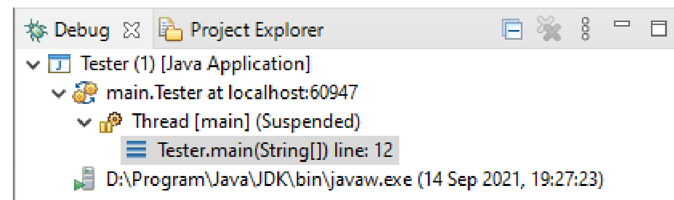



- Eclipse provides buttons in the toolbar for controlling the execution of the program you are debugging. Typically, it is easier to use the corresponding keys to control this execution.
- You use shortcut key to step through your coding. The meaning of these keys is explained in the following table.

| Key | Description |
|---|---|
| F5 | F5 executes the currently selected line and goes to the next line in your program. If the selected line is a method call the debugger steps into the associated code. |
| F6 | F6 steps over the call, i.e., it executes a method without stepping into it in the debugger. |
| F7 | F7 steps out to the caller of the currently executed method. This finishes the execution of the current method and returns to the caller of this method. |
| F8 | F8 tells the Eclipse debugger to resume the execution of the program code until is reaches the next breakpoint or watchpoint. |

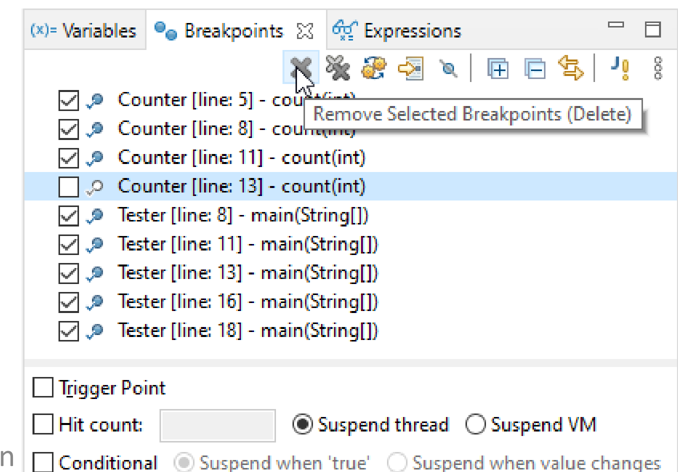
# Program execution (continued)

- The call stack shows the parts of the program which are currently executed and how they relate to each other. The current stack is displayed in the *Debug* view.



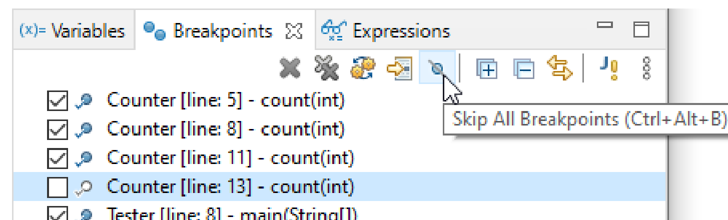
# Breakpoints view

- The *Breakpoints* view allows you to delete and deactivate *breakpoints* and *watchpoints*. You can also modify their properties.
- To deactivate a breakpoint, remove the corresponding checkbox in the *Breakpoints* view. To delete it you can use the corresponding buttons in the view toolbar. These options are depicted in the following screenshot.



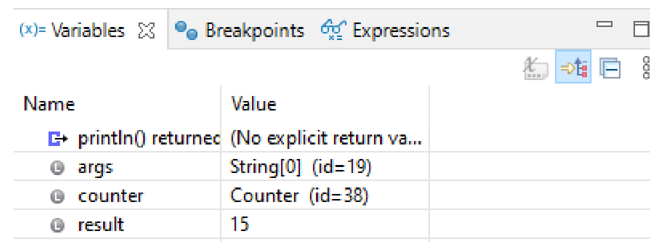
# Breakpoints view (continued)

- If you want to disable all breakpoints at the same time, you can press the Skip all breakpoints button. If you press it again, your breakpoints are reactivated. This button can be seen in the following screenshot.

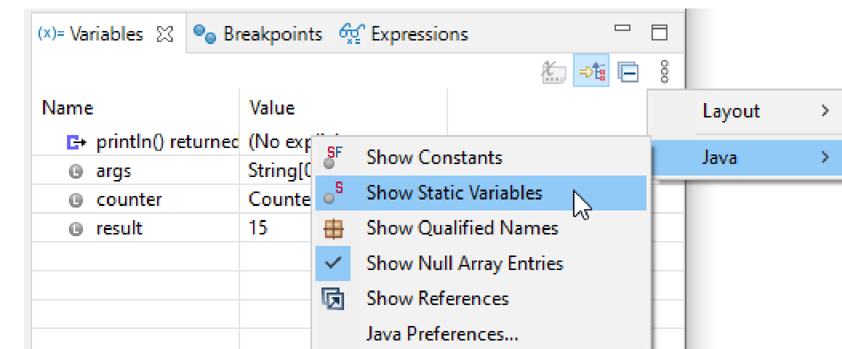


# Evaluating variables

- The *Variables* view displays fields and local variables from the current executing stack. Please note you need to run the debugger to see the variables in this view.

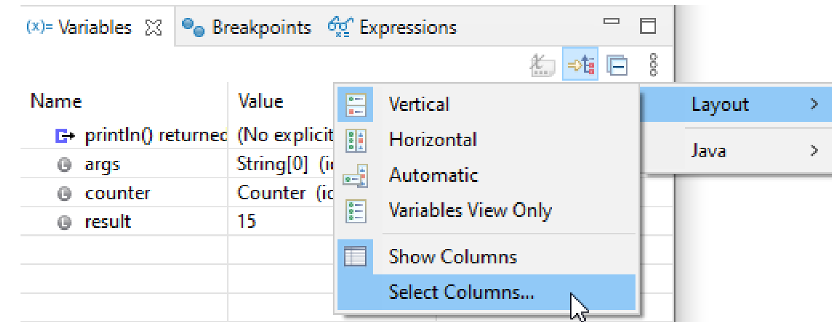


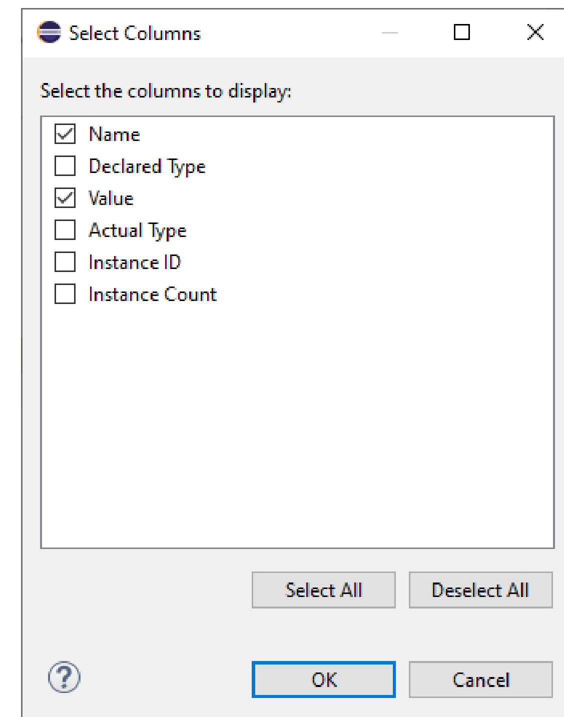

- Use the drop-down menu to display static variables.



# Evaluating variables (continued)

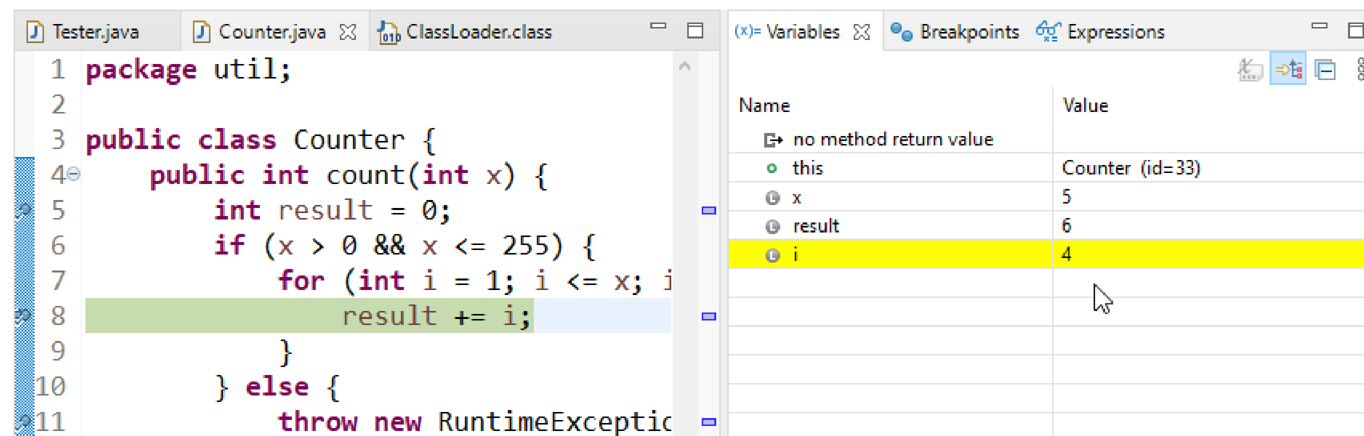
- Via the drop-down menu of the *Variables* view you can customize the displayed columns.
- For example, you can show the actual type of each variable declaration. For this select **Layout** > **Select Columns...** > **Actual Type**.



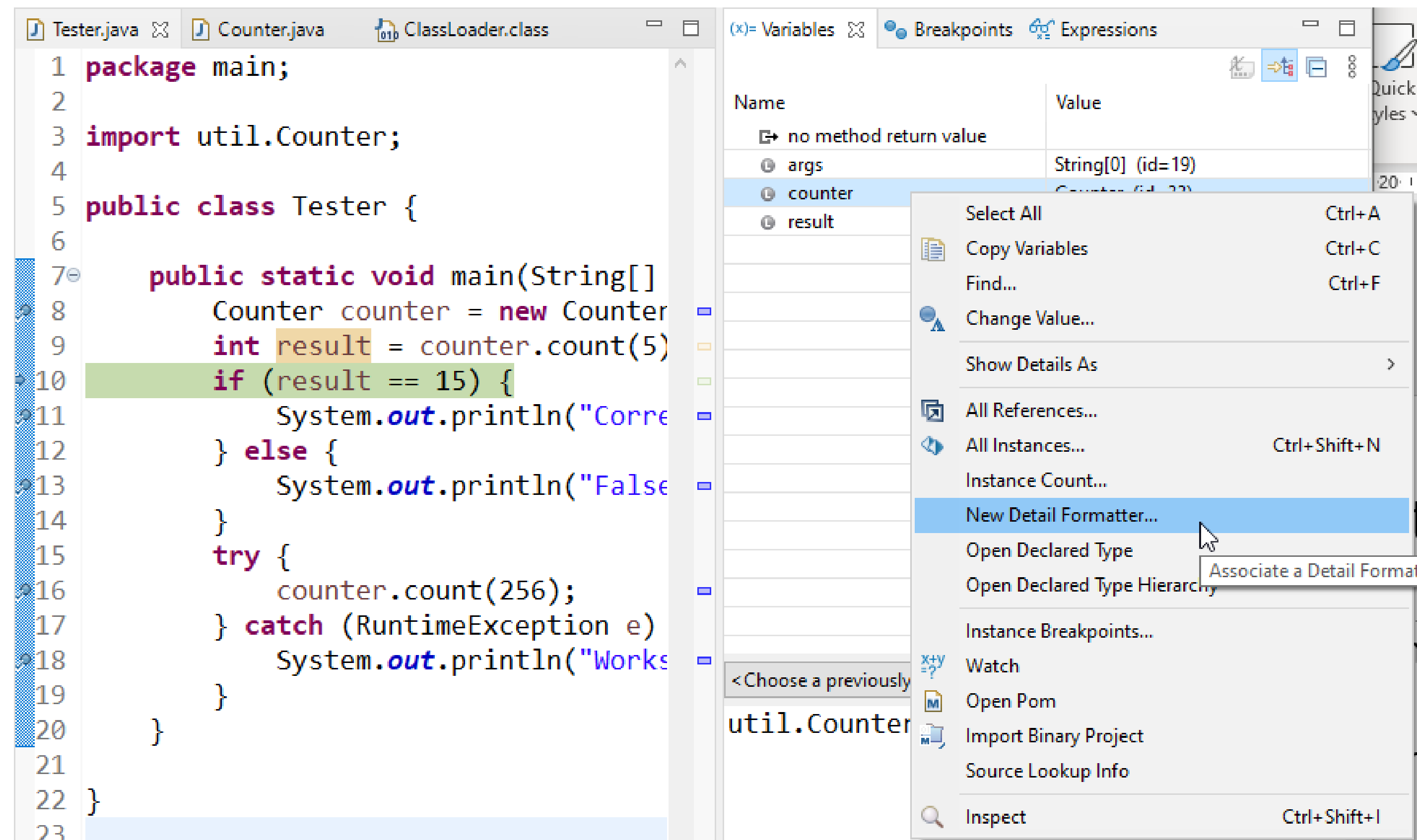
# Variable assignments: changing in debugging

- The *Variables* view allows you to change the values assigned to your variable at runtime. This is depicted in the following screenshot.



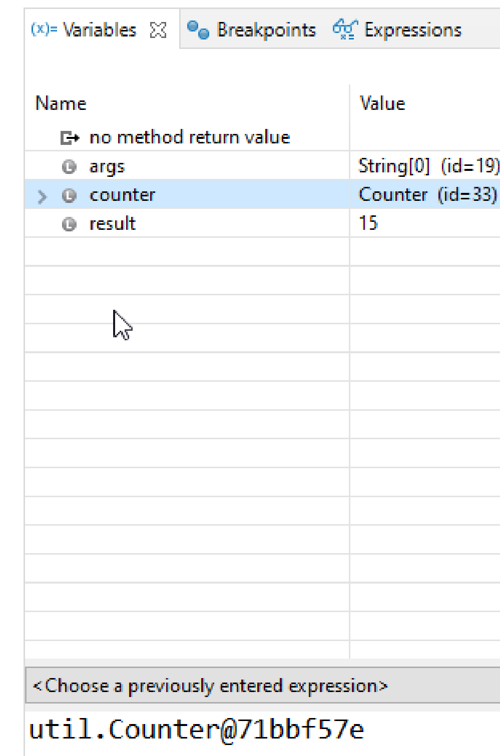
# Variable displaying: Detail formatter

- By default the *Variables* view uses the `toString()` method to determine how to display the variable.
- You can define a *Detail Formatter* in which you can use Java code to define how a variable is displayed.



# Variable displaying: Detail formatter (cont'd)

- For example, the `toString()` method in the `Counter` class may show meaningless information, e.g., `util.Counter@71bbf57e`. To make this output more readable you can right-click on the corresponding variable and select the **New Detail Formatter**... entry from the context menu.



# Variable displaying: Detail formatter (cont'd)

- Afterwards you can use a method of this class to determine the output. In this example the `getResult()` method of this class is used. This setup is depicted in the following screenshot.

Tester.java | Counter.java | ClassLoader.class

```
 1 package util;
 2
 3 public class Counter {
 4
 5     private int result = 0;
 6
 7     public int getResult() {
 8         return result;
 9     }
10
11     public int count(int x) {
12         if (x > 0 && x <= 255) {
13             for (int i = 1; i <= x; i++) {
14                 result += i;
15             }
16         } else {
17             throw new RuntimeException("x sho
18         }
19         return result;
20     }
21
22 }
```