

Debugging with Eclipse

Ing. Sebastian Daberdaku

25/03/2014

Overview

- **What is debugging?**
- **Debugging support in Eclipse**

What is debugging? (1)

- *Debugging* allows you to run a program interactively while watching the source code and the variables during the execution.
- By *breakpoints* in the source code you specify where the execution of the program should stop. To stop the execution only if a field is read or modified, you can specify *watchpoints* .

What is debugging? (2)

- *Breakpoints* and *watchpoints* can be summarized as *stop points*.
- Once the program is stopped you can investigate variables, change their content, etc.

Debugging support in Eclipse

- Eclipse allows you to start a Java program in *Debug mode*.
- Eclipse has a special *Debug perspective* which gives you a preconfigured set of *views*. In this *perspective* you control the execution process of your program and can investigate the state of the variables.

Eclipse Debug perspective

The screenshot displays the Eclipse IDE in the Debug perspective. The main window shows the Java application running, with a breakpoint set at line 9 of `Main.java`. The Variables window shows the `args` parameter as a `String[]` (id=16). The Outline window shows the class structure, with the `main(String[]) : void` method selected. The Console window is empty.

```
package de.vogella.debug.first;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.count();
        System.out.println("We have counted " + counter.getResult());
    }
}
```

Name	Value
args	String[] (id=16)

de.vogella.debug.first

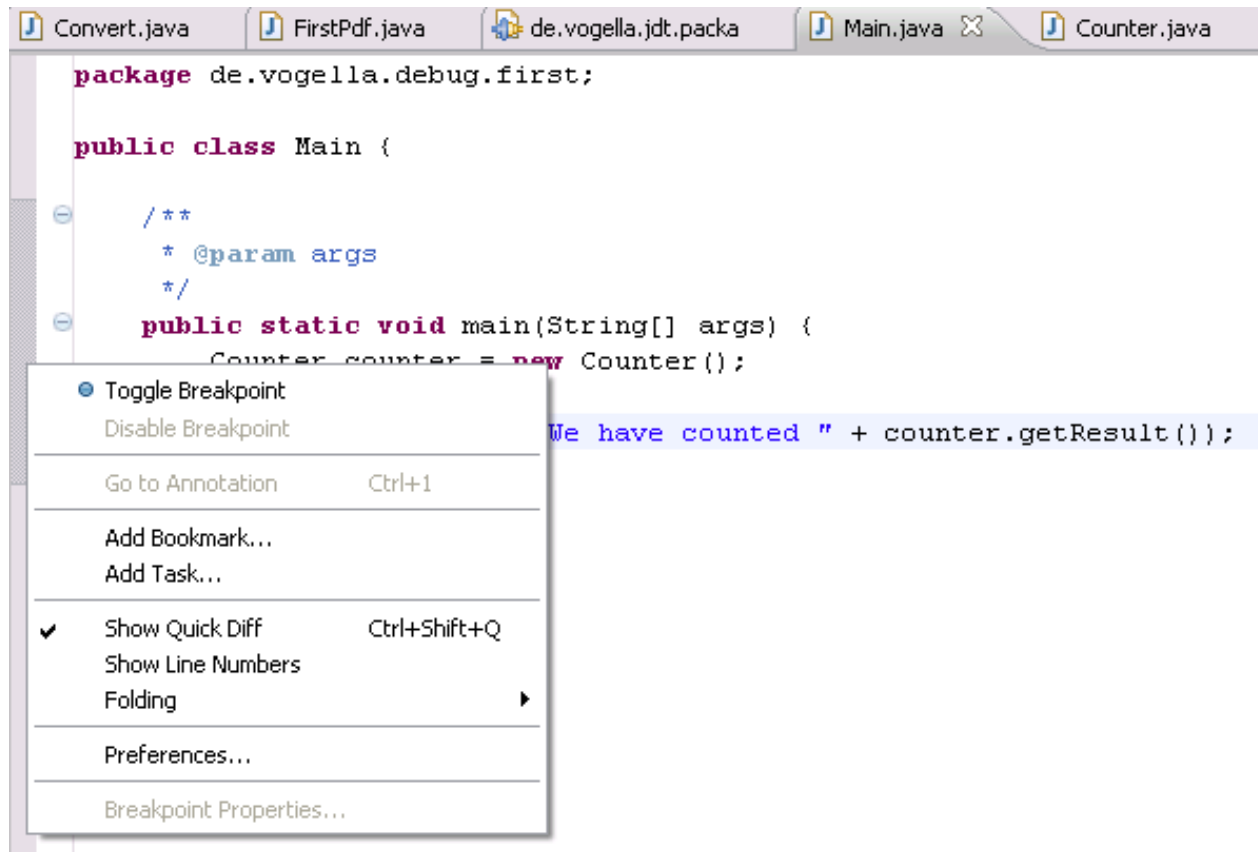
- Main
 - main(String[]) : void

Main (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (06.07.2009 11:30:57)

Writable Smart Insert 9 : 1

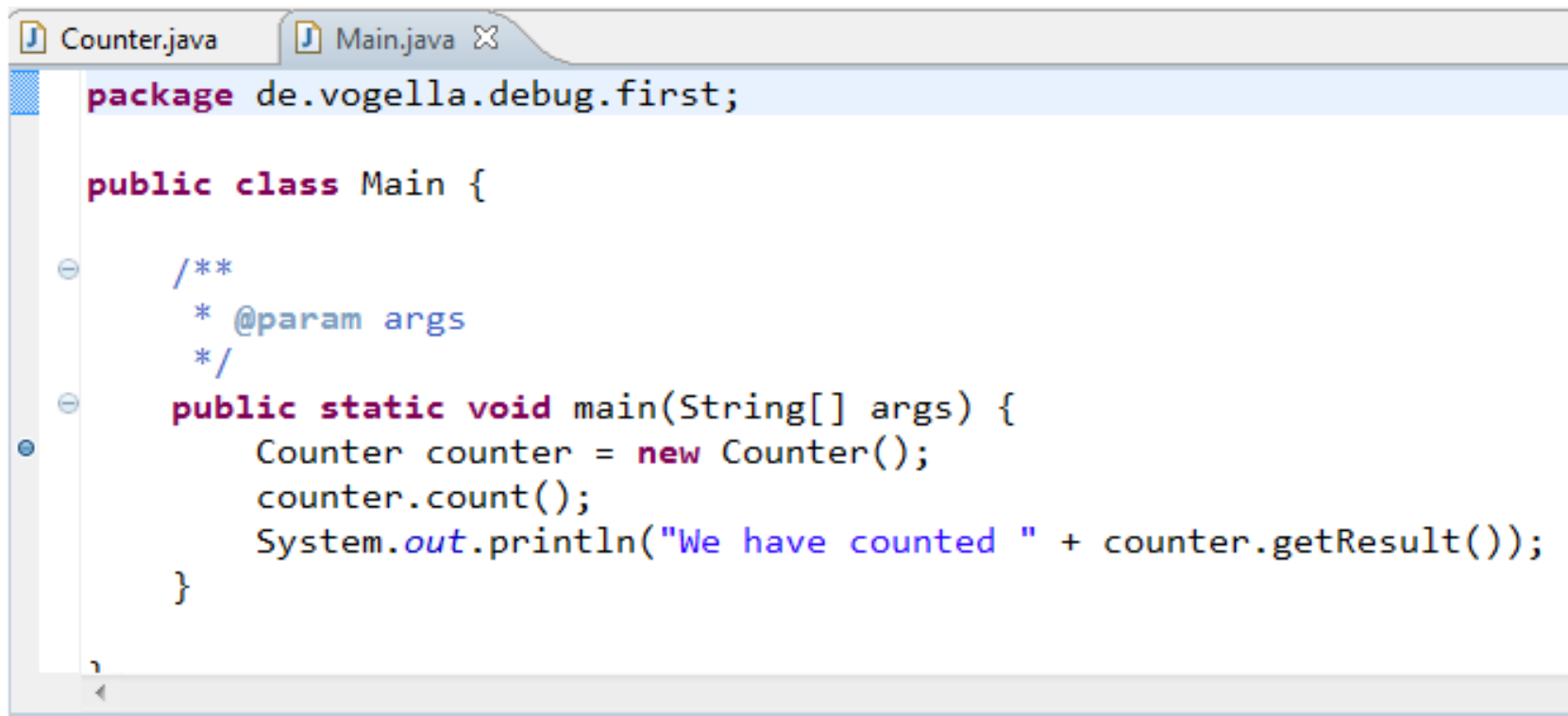
Setting Breakpoints (1)

- To set breakpoints in your source code right-click in the small left margin in your source code editor and select Toggle Breakpoint. Alternatively you can double-click on this position.



Setting Breakpoints (2)

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



The screenshot shows an IDE window with two tabs: 'Counter.java' and 'Main.java'. The 'Main.java' tab is active and displays the following code:

```
package de.vogella.debug.first;

public class Main {

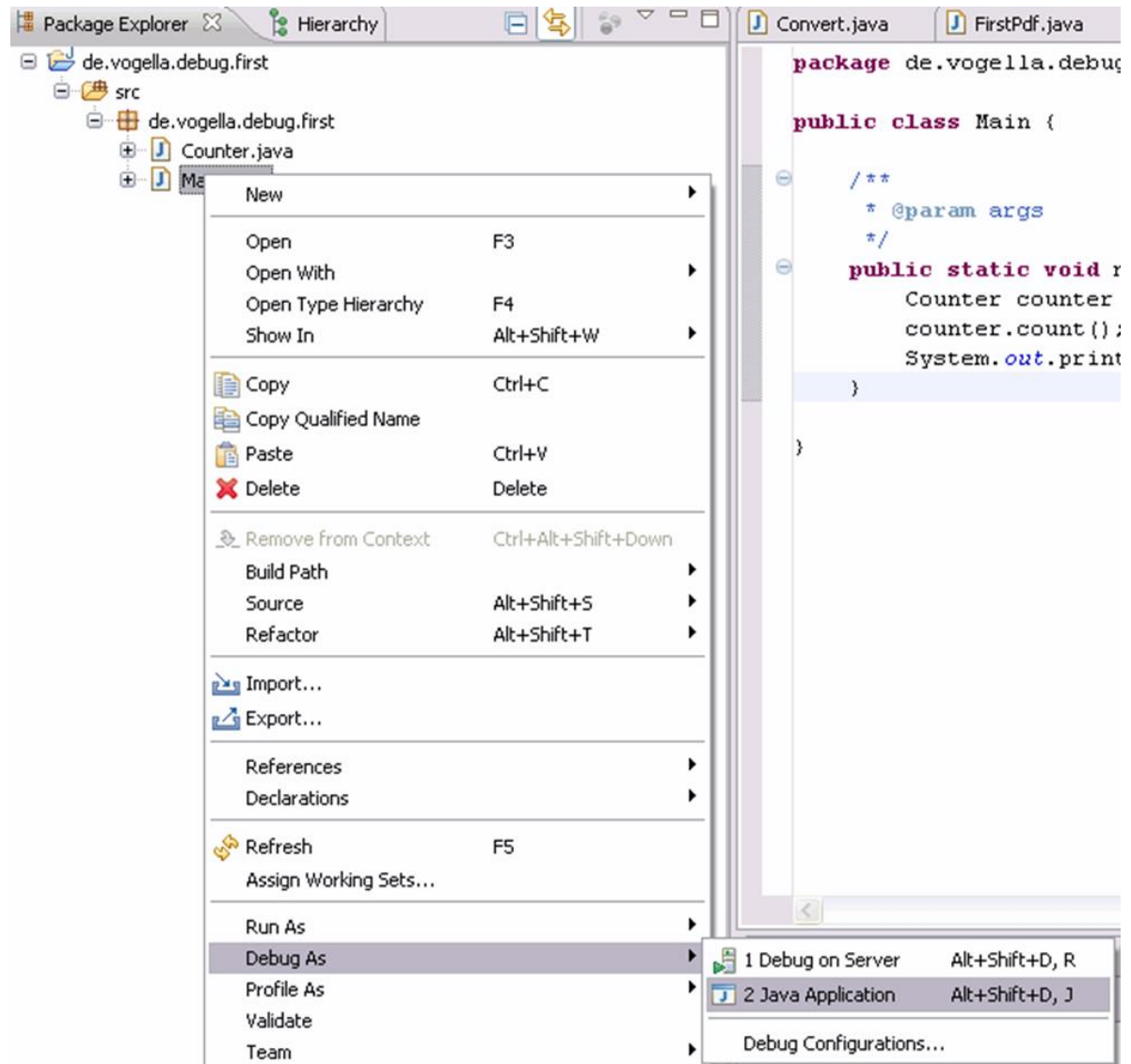
    /**
     * @param args
     */
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.count();
        System.out.println("We have counted " + counter.getResult());
    }
}
```

A blue dot, representing a breakpoint, is placed on the left margin of the line `Counter counter = new Counter();`. The code is color-coded: keywords like `package`, `public`, `class`, `void`, and `main` are in purple; comments are in blue; and identifiers like `args`, `String`, and `getResult` are in blue.

Starting the Debugger (1)

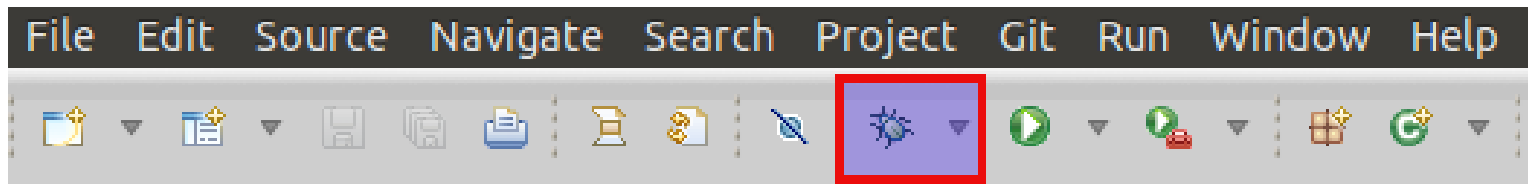
- To debug your application, select a Java file which can be executed, right-click on it and select:

“Debug As” →
“Java Application”.



Starting the Debugger (2)

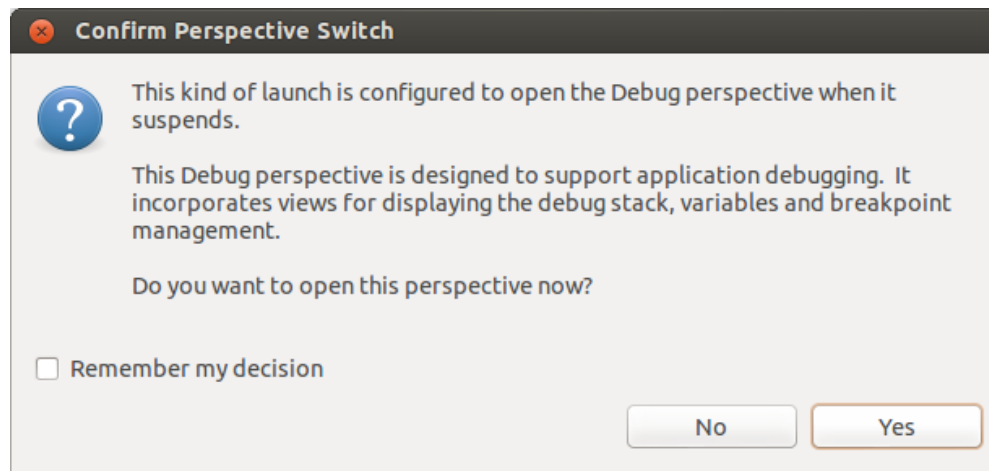
- After you have started the application once via the context menu, you can use the created launch configuration again via the Debug button in the Eclipse toolbar.



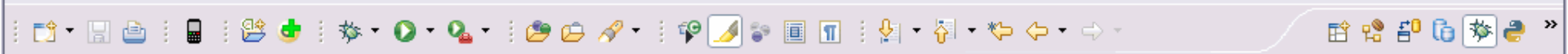
- If you have not defined any breakpoints, this will run your program as normal. To debug the program you need to define breakpoints.

Starting the Debugger (3)

- If you start the debugger, Eclipse asks you if you want to switch to the Debug *perspective* once a stop point is reached. Answer Yes in the corresponding dialog.



- Afterwards Eclipse opens this *perspective*, which looks similar to the following screenshot.



Debug console showing the execution stack:

- Main (1) [Java Application]
- de.vogella.debug.first.Main at localhost:2038
 - Thread [main] (Suspended (breakpoint at line 9 in Main))
 - Main.main(String[]) line: 9
- C:\Program Files\Java\jre6\bin\javaw.exe (06.07.2009 11:30:57)

Variables and Breakpoints panel:

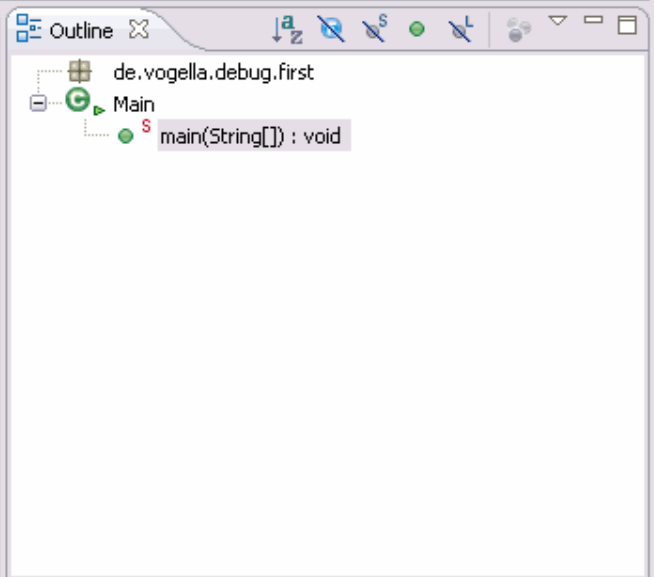
Name	Value
args	String[0] (id=16)

Convert.java de.vogella.jdt.packa Main.java Counter.java >>20

```
package de.vogella.debug.first;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Counter counter = new Counter();
        counter.count();
        System.out.println("We have counted " + counter.getResult());
    }
}
```



Console Tasks

Main (1) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (06.07.2009 11:30:57)

Controlling the program execution (1)

- 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 can use the F5, F6, F7 and F8 key to step through your coding.
- The following picture displays the buttons and their related keyboard shortcuts.

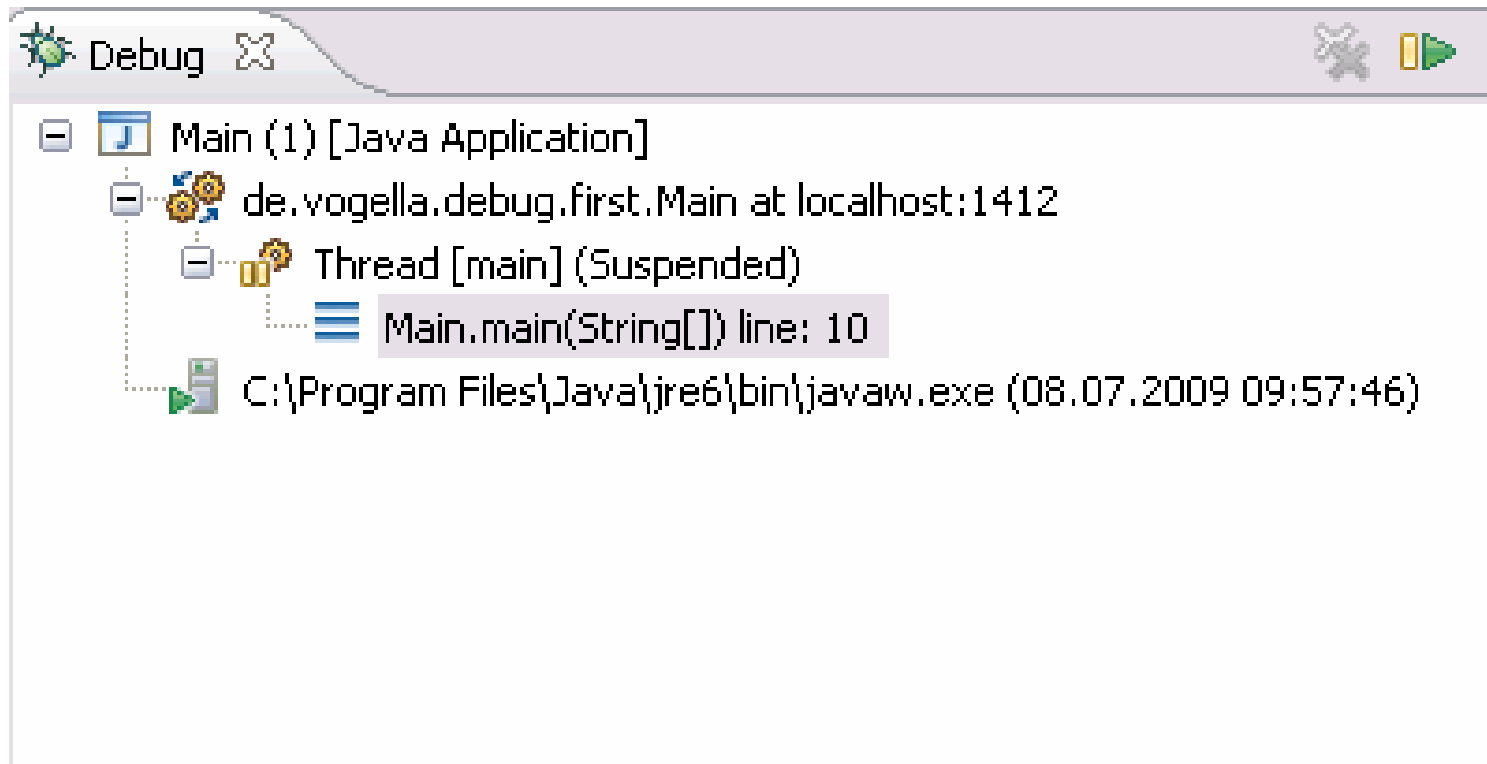


Controlling the program execution (2)

Key	Description
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 it reaches the next breakpoint or watchpoint.

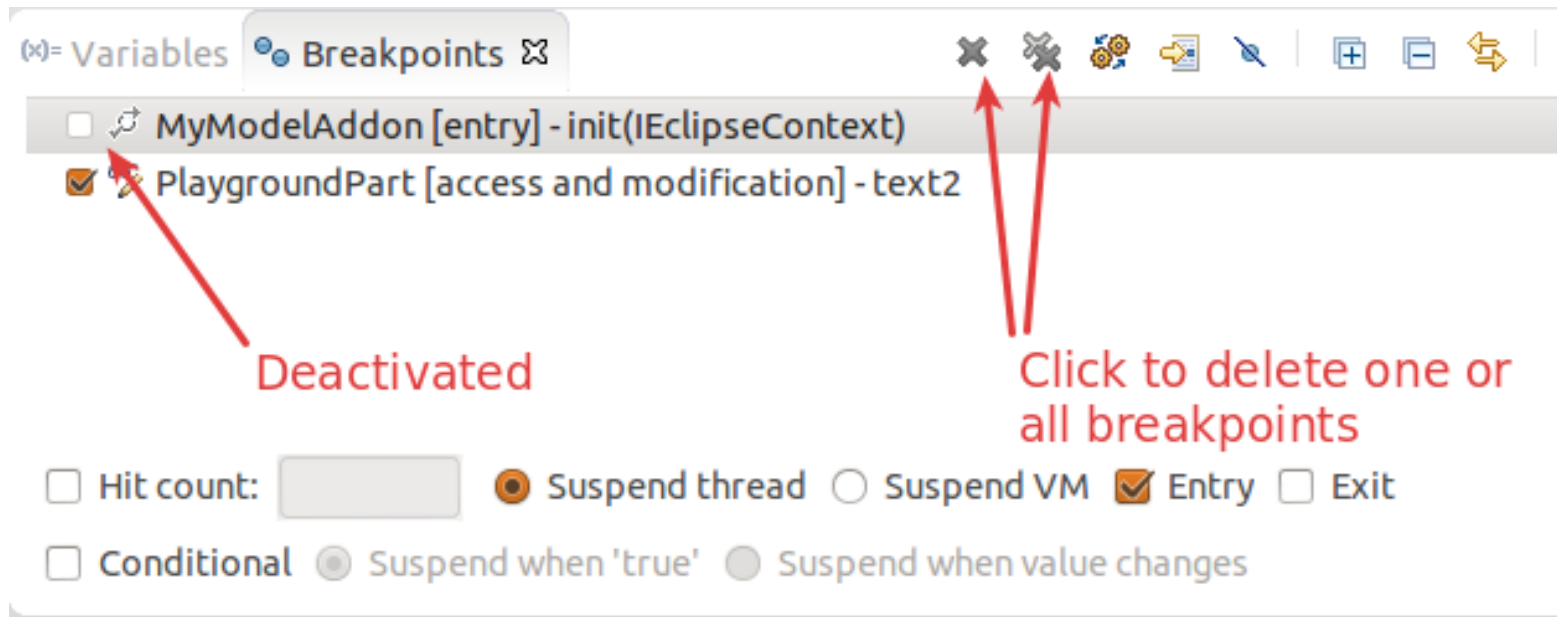
Controlling the program execution (3)

- 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 and deactivating breakpoints (1)

- The Breakpoints view allows you to delete and deactivate stop points, i.e. breakpoints and watchpoints and to 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.



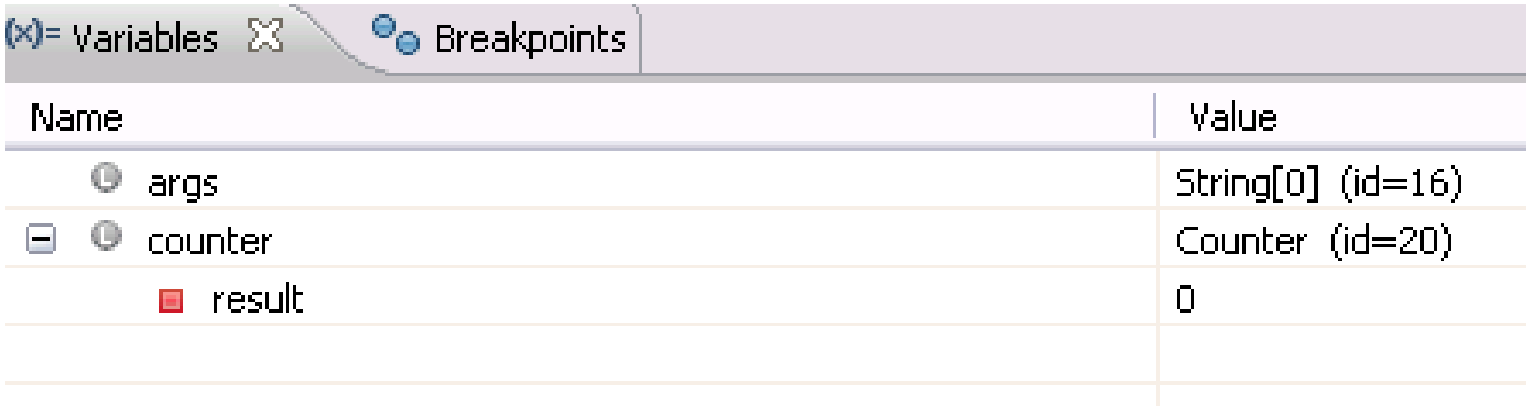
Breakpoints view and deactivating breakpoints (2)

- If you want to deactivate all your breakpoints you can press the Skip all breakpoints button. If you press it again, your breakpoints are reactivated. This button is highlighted in the following screenshot.



Evaluating variables in the debugger (1)

- 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 .

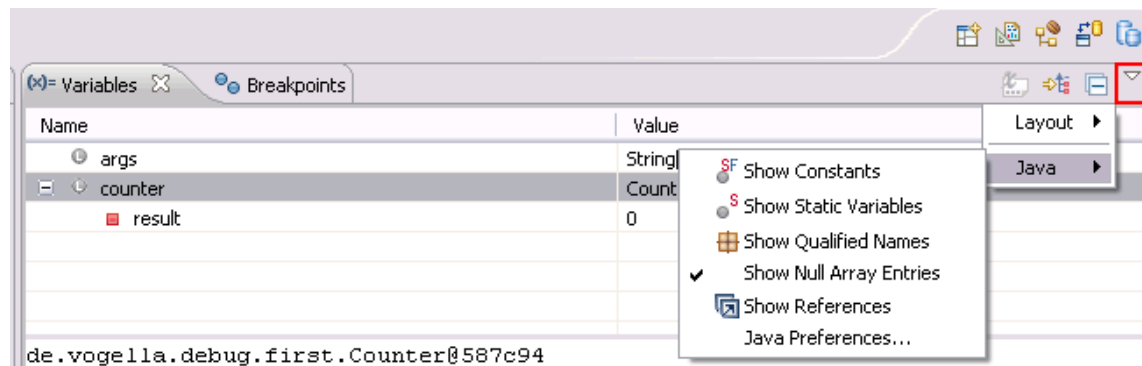


The screenshot shows a debugger interface with two tabs: 'Variables' and 'Breakpoints'. The 'Variables' tab is active and displays a table of variables. The table has two columns: 'Name' and 'Value'. The variables listed are 'args', 'counter', and 'result'. 'args' is a String array, 'counter' is a Counter object, and 'result' is an integer.

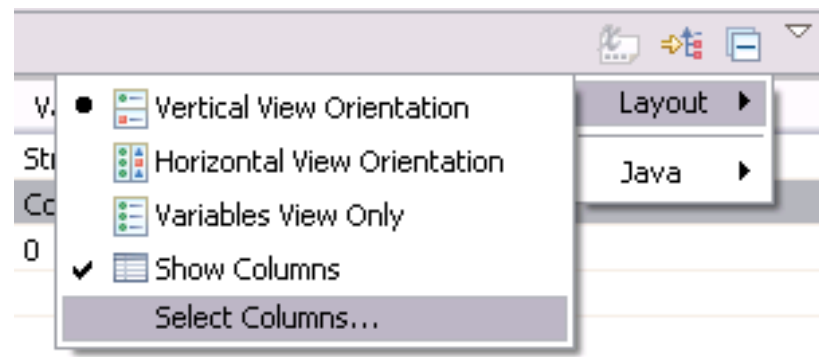
Name	Value
args	String[0] (id=16)
counter	Counter (id=20)
result	0

Evaluating variables in the debugger (2)

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



- 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... → Type.



Changing variable assignments in the debugger

- The Variables view allows you to change the values assigned to your variable at runtime.

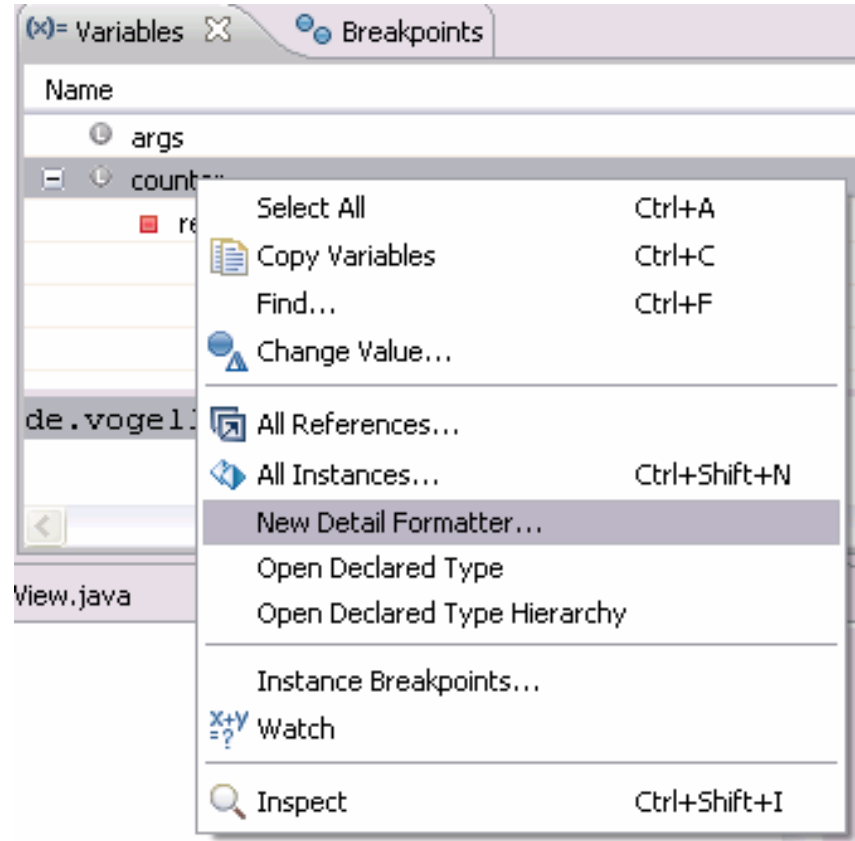
The screenshot shows a debugger interface with two tabs: "Variables" and "Breakpoints". The "Variables" tab is active, displaying a table with the following data:

Name	Value
args	String[0] (id=15)
sum	40

A red arrow points from the text "Change value here" to the value "40" in the "sum" row. The value "40" is also displayed in a separate box at the bottom left of the image.

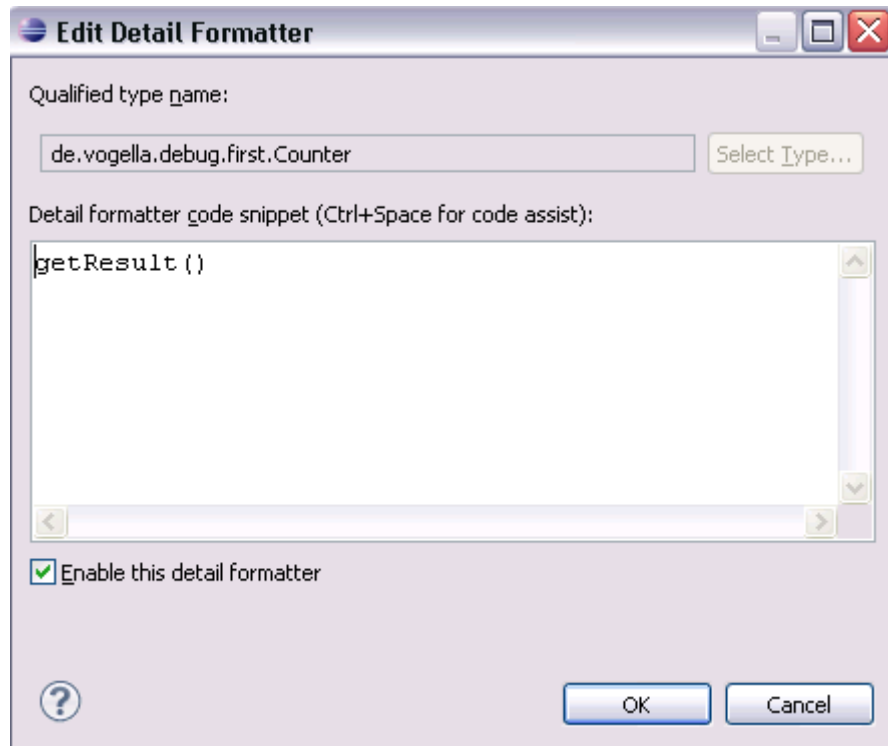
Controlling the display of the variables with Detail Formatter (1)

- 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.
- For example the `toString()` method in the `Counter` class may show meaningless information, e.g. `de.vogella.combug.first.Counter@587c94`. 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.



Controlling the display of the variables with Detail Formatter (2)

- 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.



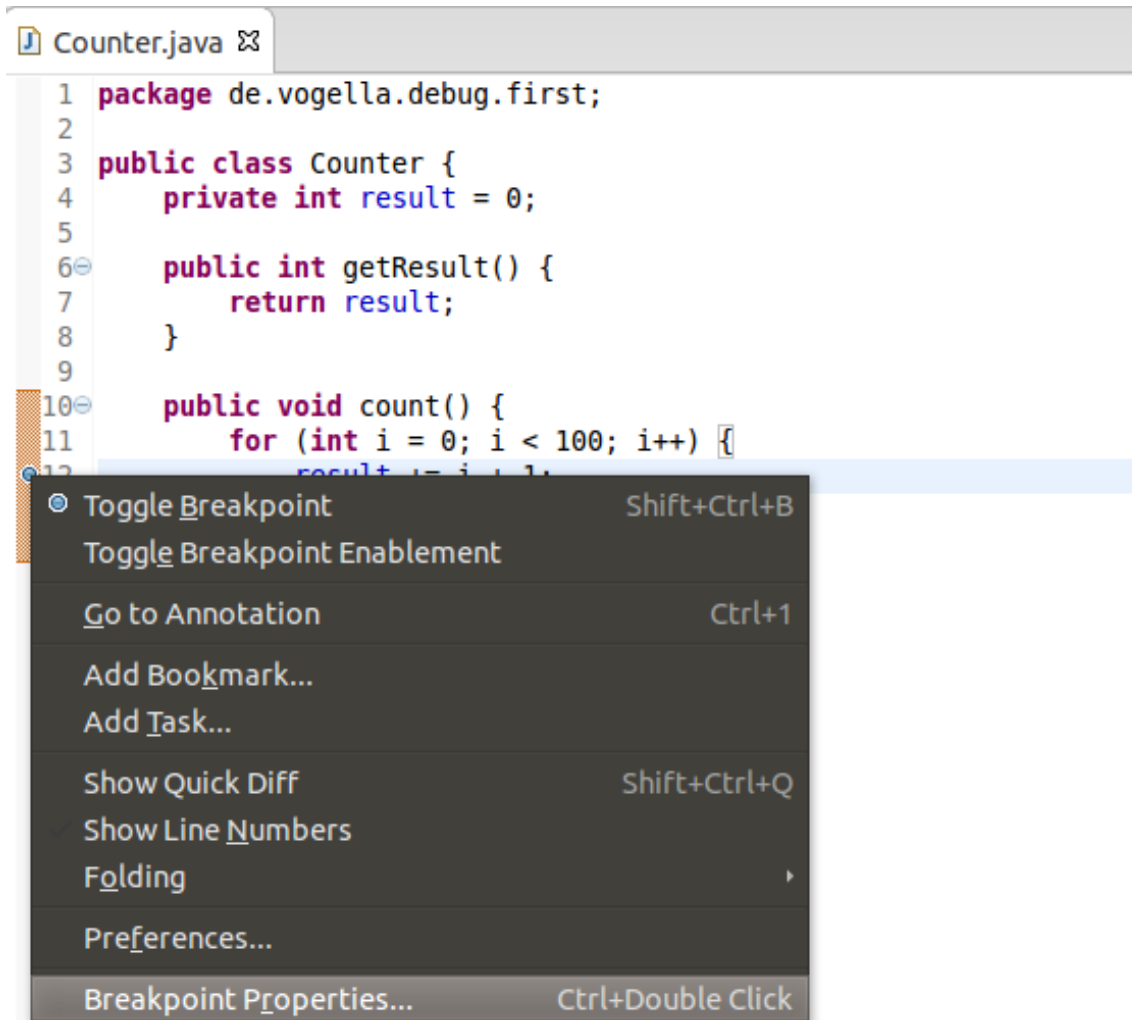
Advanced Debugging

The following section shows more options you have for debugging.

Breakpoint properties (1)

- After setting a breakpoint you can select the properties of the breakpoint, via right-click → “Breakpoint Properties”. Via the breakpoint properties you can define a condition that restricts the activation of this breakpoint.
- You can for example specify that a breakpoint should only become active after it has reached 12 or more times via the Hit Count property.
- You can also create a conditional expression. The execution of the program only stops at the breakpoint, if the condition evaluates to true. This mechanism can also be used for additional logging, as the code that specifies the condition is executed every time the program execution reaches that point.

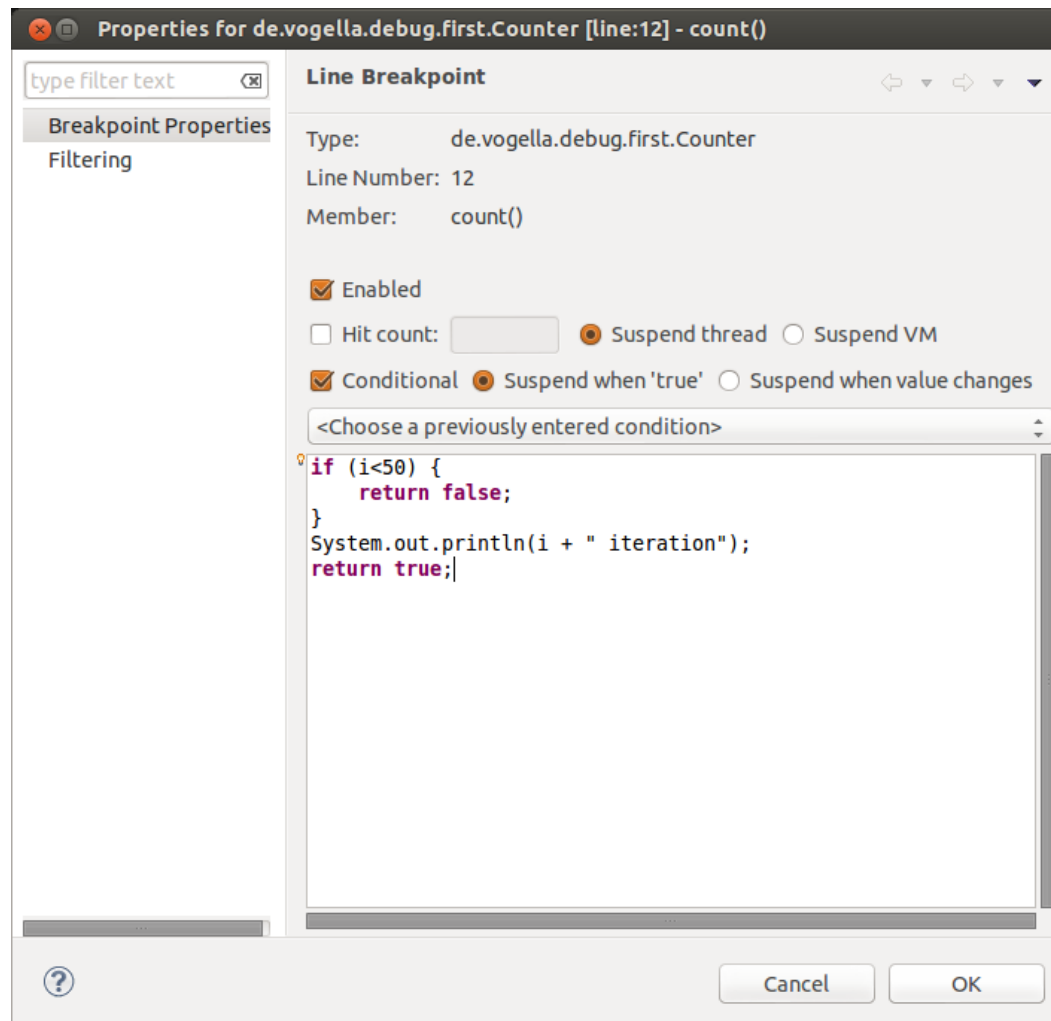
Breakpoint properties (2)



```
1 package de.vogella.debug.first;
2
3 public class Counter {
4     private int result = 0;
5
6     public int getResult() {
7         return result;
8     }
9
10    public void count() {
11        for (int i = 0; i < 100; i++) {
12            result = i + 1;
13        }
14    }
15 }
```

- Toggle Breakpoint Shift+Ctrl+B
- Toggle Breakpoint Enablement
- Go to Annotation Ctrl+1
- Add Bookmark...
- Add Task...
- Show Quick Diff Shift+Ctrl+Q
- Show Line Numbers
- Folding ▶
- Preferences...
- Breakpoint Properties... Ctrl+Double Click

Breakpoint properties (3)



Watchpoint (1)

- A watchpoint is a breakpoint set on a field. The debugger will stop whenever that field is read or changed.
- You can set a watchpoint by double-clicking on the left margin, next to the field declaration. In the properties of a watchpoint you can configure if the execution should stop during read access (Field Access) or during write access (Field Modification) or both.

Watchpoint (2)

The image shows a screenshot of an IDE with a Java code editor and a dialog box. The code editor displays the following code:

```
package de.vogella.debug.first;  
  
public class Counter {  
    private int result=0;  
    public int getResult() {
```

The dialog box, titled "Properties for de.vogella.debug.first.Counter - result", is open over the code. It has a left sidebar with "Breakpoint Properties" and "Filtering" options. The main area is titled "Watchpoint" and contains the following settings:

- Type: de.vogella.debug.first.Counter
- Field: result
- Enabled
- Hit Count: [empty text box]
- Suspend on:
 - Field Access
 - Field Modification
- Suspend Policy: Suspend Thread [dropdown arrow]

The "Field Access" and "Field Modification" options are highlighted with a red box. At the bottom of the dialog box, there are "OK" and "Cancel" buttons.

Exception breakpoints

- You can set breakpoints which are triggered when exceptions in your Java source code are thrown. To define an exception breakpoint click on the Add Java Exception Breakpoint button icon in the Breakpoints view toolbar.

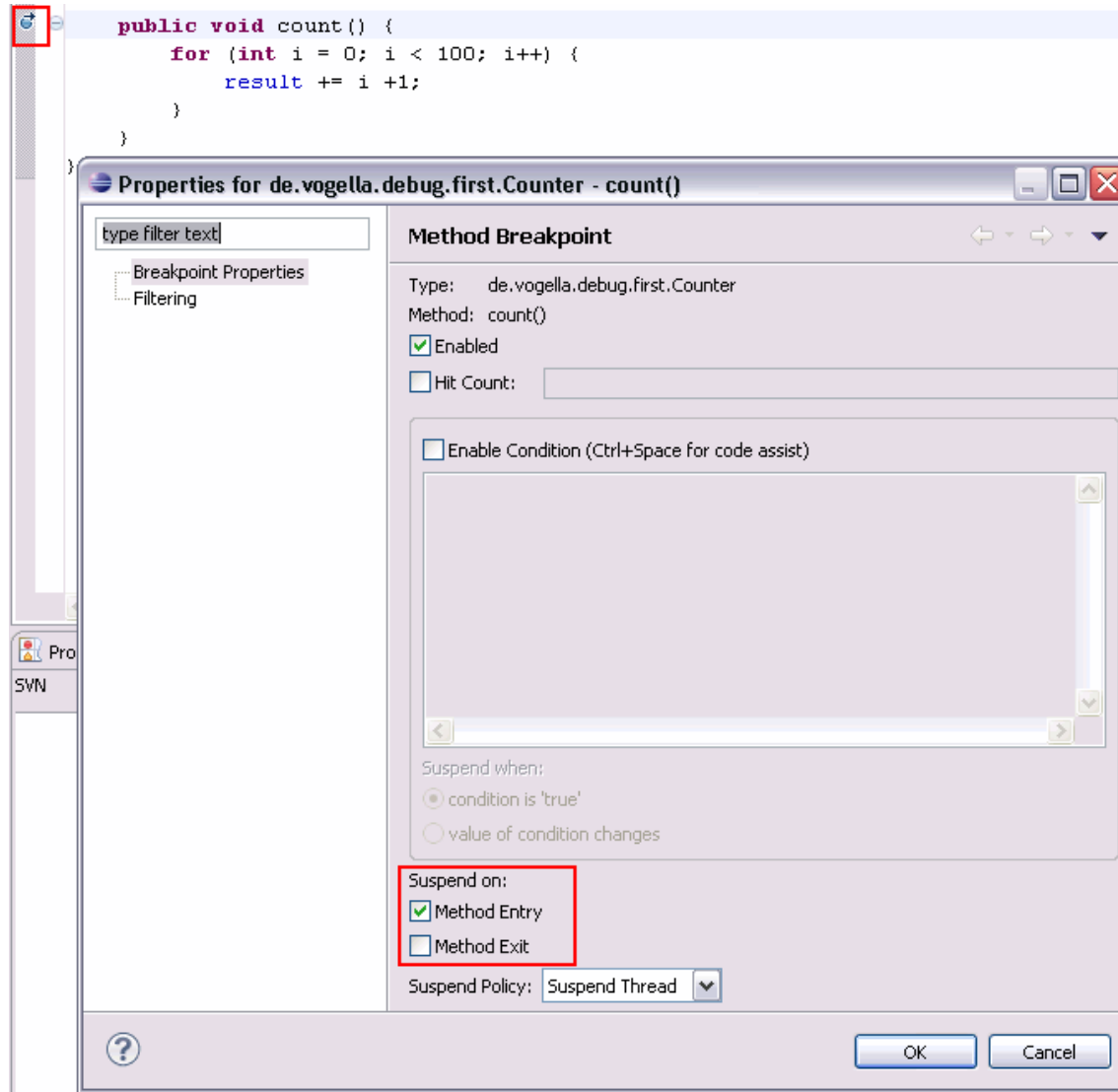


- You can configure if the debugger should stop at caught or uncaught exceptions.

Method breakpoint (1)

- A method breakpoint is defined by double-clicking in the left margin of the editor next to the method header.
- You can configure if you want to stop the program before entering or after leaving the method.

Method breakpoint (2)



Breakpoints for loading classes (1)

- A class load breakpoint stops when the class is loaded.
- To set a class load breakpoint, right-click on a class in the Outline view and choose the Toggle Class Load Breakpoint option.
- Alternative you can double-click in the left border of the Java editor beside the class definition.

Breakpoints for loading classes (2)

The screenshot displays the Eclipse IDE interface during a debug session. The main window title is "Debug - SixSigmaRCP/src/sixsigmarcp/ApplicationWorkbenchAdvisor.java - Eclipse". The menu bar includes File, Edit, Pydev, Run, Source, Refactor, Navigate, Search, Project, Window, and Help. The toolbar contains various icons for file operations, debugging, and navigation.

The Breakpoints view (top right) shows a list of breakpoints:

- Exception: caught and uncaught
- RuntimeException: caught and uncaught
- DBReader [line: 60] - read()
- PrintHandler [line: 136] - createReport(IRequirementList)
- RequirementsReader [line: 17] - readRequirements(GenericReade
- RequirementsReader [line: 36] - convertReq(ResultSet)

The code editor (bottom left) shows the source code of ApplicationWorkbenchAdvisor.java:

```
1 package sixsigmarcp;
2
3 import java.io.IOException;
4
25
26 public class ApplicationWorkbenchAdvisor {
27
28     // private final static Logger logger
29     // Logger.getLogger(ApplicationWorkbenchAdvisor.class);
30
31     public void initialize(IWorkbenchConfigurer configurer) {
32         super.initialize(configurer);
33         configurer.setSaveAndRestore(false);
34     }
35 }
```

The Outline view (bottom right) shows the class hierarchy for sixsigmarcp, including ApplicationWorkbenchAdvisor and its methods: initialize(IWorkbenchConfigurer), createWorkbenchWindowAdvisor(), getInitialWindowPerspective(), preStartup(): void, and preShutdown(): boolean.

A context menu is open over the ApplicationWorkbenchAdvisor class in the Outline view, showing the following options:

- Open Type Hierarchy (F4)
- Open Call Hierarchy (Ctrl+Alt+H)
- Show In (Alt+Shift+W)
- Copy (Ctrl+C)
- Copy Qualified Name
- Paste (Ctrl+V)
- Delete (Delete)
- Remove from Context (Ctrl+Alt+Shift+Down)
- Source (Alt+Shift+S)
- Refactor (Alt+Shift+T)
- References
- Declarations
- Toggle Class Load Breakpoint**

Step Filter

- You can define that certain packages should be skipped in debugging. This is for example useful if you use a framework for testing but don't want to step into the test framework classes.
- These packages can be configured via the Window → Preferences → Java → Debug → Step Filtering menu path.

Hit Count

- For every breakpoint you can specify a hit count in its properties. The application is stopped once the breakpoint has been reached the number of times defined in the hit count.

Drop to frame (1)

- Eclipse allows you to select any level (frame) in the call stack during debugging and set the JVM to restart from that point.
- This allows you to rerun a part of your program. Be aware that variables which have been modified by code that already run will remain modified.
- To use this feature, select a level in your stack and press the Drop to Frame button in the toolbar of the Debug view .

Note: Fields and external data may not be affected by the reset. For example if you write a entry to the database and afterward drop to a previous frame, this entry is still in the database.

- The following screenshot depicts such a reset. If you restart your for loop, the field result is not set to its initial value and therefore the loop is not executed as without resetting the execution to a previous point.

Drop to frame (2)

The screenshot shows an IDE with a Java application being debugged. The call stack on the left shows the following frames:

- Main [Java Application]
- de.vogella.debug.first.Main at localhost:51447
 - Thread [main] (Suspended)
 - Counter.count() line: 12 (highlighted)
 - Main.main(String[]) line: 10
- C:\Program Files\Java\jre7\bin\javaw.exe (Dec 16, 2011 11:06:10 AM)

The 'Drop To Frame' button is highlighted over the 'Counter.count() line: 12' frame.

The Variables window on the right shows the following variables:

Name	Value
this	Counter (id=19)
result	44
i	0

The source code for Counter.java is shown below, with the line 'result += i + 1;' highlighted:

```
package de.vogella.debug.first;

public class Counter {
    private int result = 0;

    public int getResult() {
        return result;
    }

    public void count() {
        for (int i = 0; i < 100; i++) {
            result += i + 1;
        }
    }
}
```

Remote debugging (1)

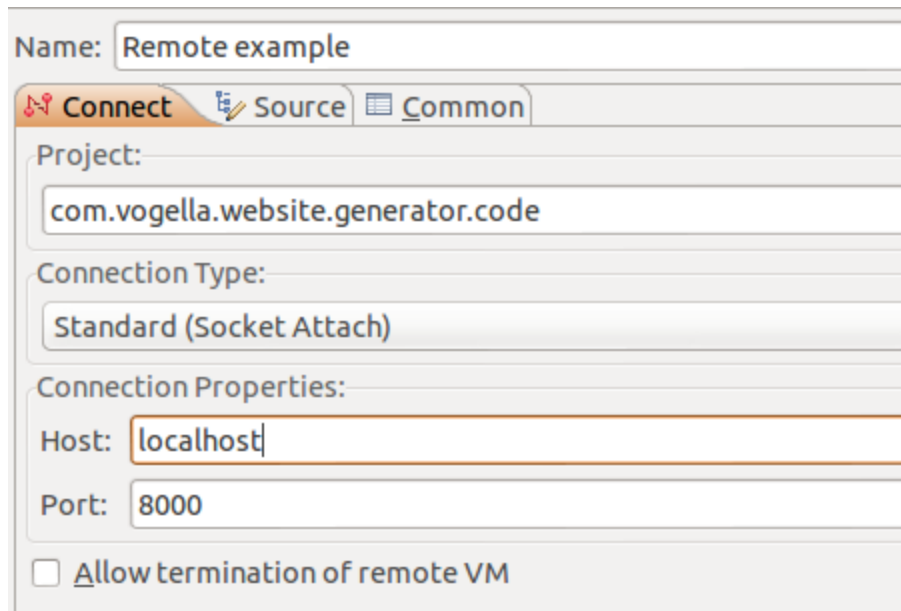
- Eclipse allows you to debug applications which runs on another Java virtual machine or even on another machine.
- To enable remote debugging you need to start your Java application with certain flags, as demonstrated in the following code example.

```
java -Xdebug -Xnoagent \  
-Djava.compiler=NONE \  
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
```

- In you Eclipse IDE you can enter the hostname and port to connect for debugging via the Run → Debug Configuration... menu.

Remote debugging (2)

- Here you can create a new debug configuration of the Remote Java Application type. This configuration allows you to enter the hostname and port for the connection.



The screenshot shows the Eclipse IDE configuration dialog for a Remote Java Application. The dialog has a title bar with the name "Remote example". Below the title bar are three tabs: "Connect" (selected), "Source", and "Common". The "Connect" tab contains the following fields:

- Name:** Remote example
- Project:** com.vogella.website.generator.code
- Connection Type:** Standard (Socket Attach)
- Connection Properties:**
 - Host:** localhost
 - Port:** 8000
- Allow termination of remote VM

Note:

Remote debugging requires that you have the source code of the application which is debugged available in your Eclipse IDE.

Exercise:

Create a Project for debugging

Some references:

- **Java Debugging with Eclipse - Tutorial**

<http://www.vogella.com/tutorials/EclipseDebugging/article.html>

- **Effective Java Debugging with Eclipse**

<http://eclipsesource.com/blogs/2013/01/08/effective-java-debugging-with-eclipse/>

- **Top 10 Java Debugging Tips with Eclipse**

<http://javapapers.com/core-java/top-10-java-debugging-tips-with-eclipse/>

- **Using the Eclipse Debugger for Beginning Programmers**

<http://agile.csc.ncsu.edu/SEMaterials/tutorials/eclipse-debugger/>

- **Again! – 10 Tips on Java Debugging with Eclipse**

<https://blog.codecentric.de/en/2013/04/again-10-tips-on-java-debugging-with-eclipse/>

- **Eclipse And Java: Free Video Tutorials**

<http://eclipsetutorial.sourceforge.net/debugger.html>

- **Debugging Android applications - Tutorial**

<http://www.vogella.com/tutorials/AndroidDebugging/article.html>

- **Using DDMS**

<http://developer.android.com/tools/debugging/ddms.html>