Debugging with Eclipse

Matthew Thomas

What is Debugging?

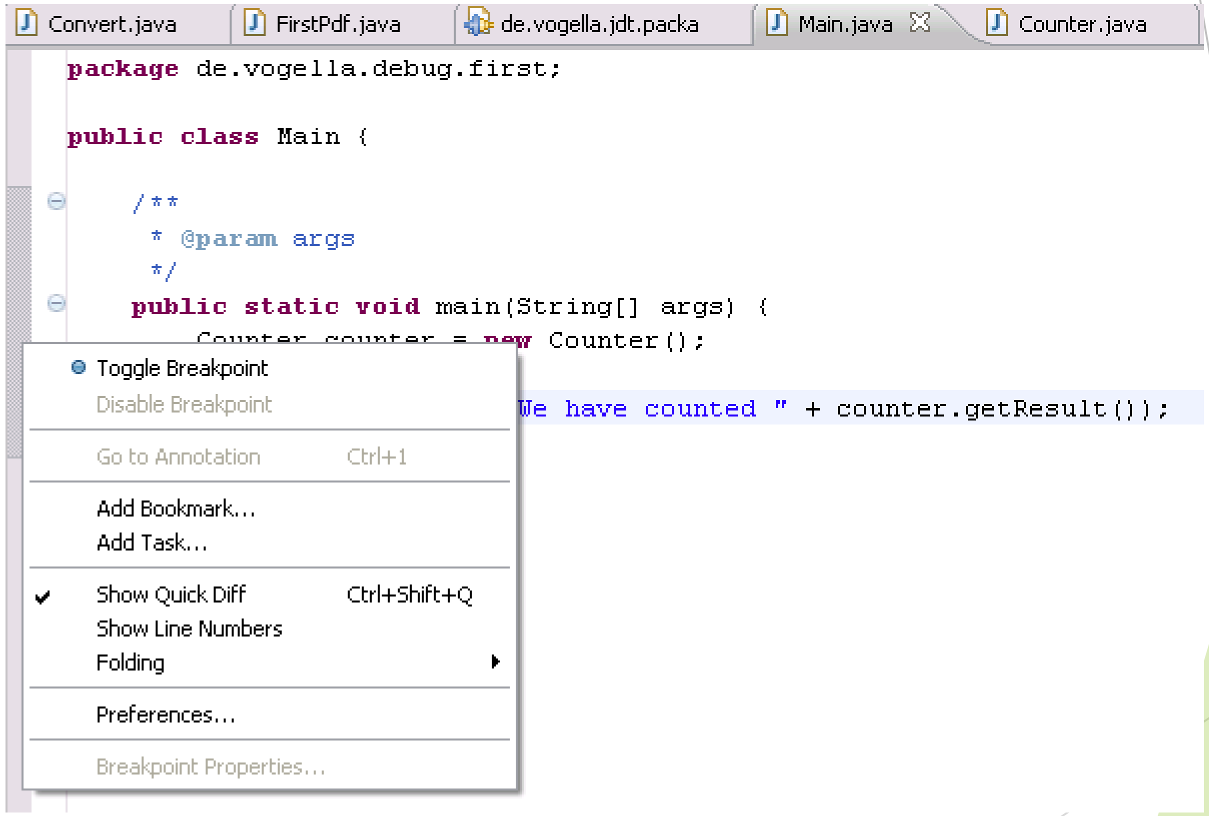
- *Debugging* allows you to run a program interactively while watching the source code and the variables during the execution.
- By *breakpoints* in the source code you specify where the execution of the program should stop. To stop the execution only if a field is read or modified, you can specify *watchpoints* .
- *Breakpoints* and *watchpoints* can be summarized as *stop points*.
- Once the program is stopped you can investigate variables, change their content, etc.

Debugging with 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.

Setting Breakpoints

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



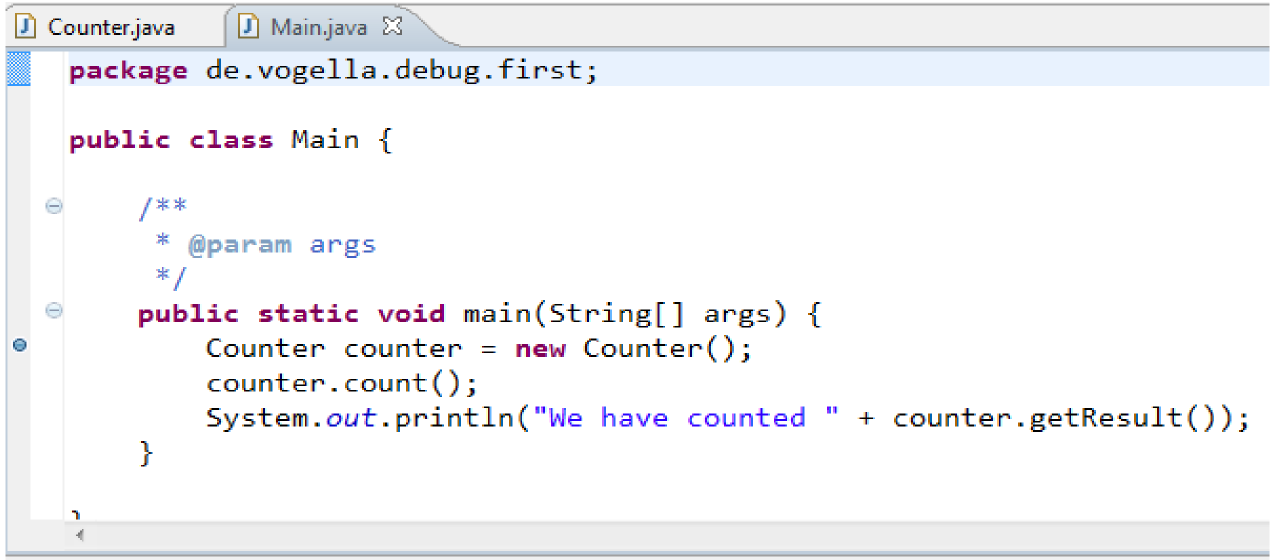
```
package de.vogella.debug.first;

public class Main {

    /**
     * @param args
     */
    public static void main(String[] args) {
        Counter counter = new Counter();
```

```
We have counted " + counter.getResult());
```

Setting Breakpoints

- For example in the following screenshot we set a breakpoint on the line:

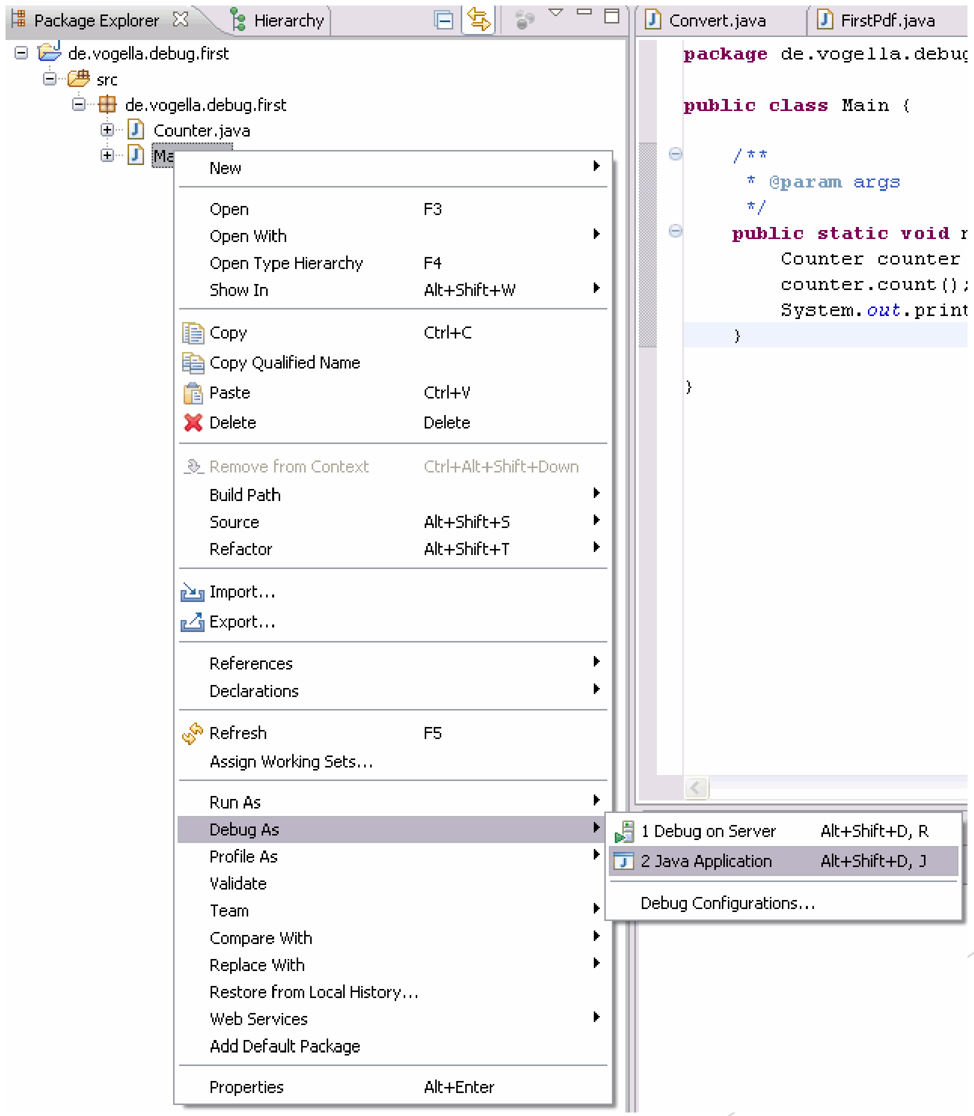
```
Counter counter = new Counter();
```



```
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());
    }

}
```

Starting the Debugger

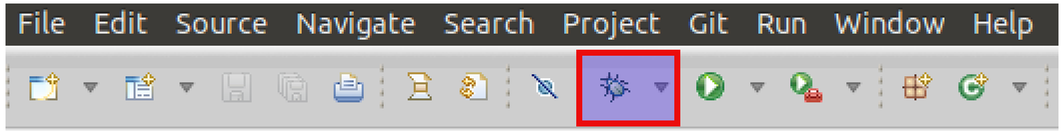
- To debug your application:
  - select a Java file which can be executed
  - right-click on it
  - select Debug As → Java Application.



```
package de.vogella.debu

public class Main {

    /**
     * @param args
     */
    public static void r
        Counter counter
        counter.count();
        System.out.print
    }

}
```

Starting Debugger

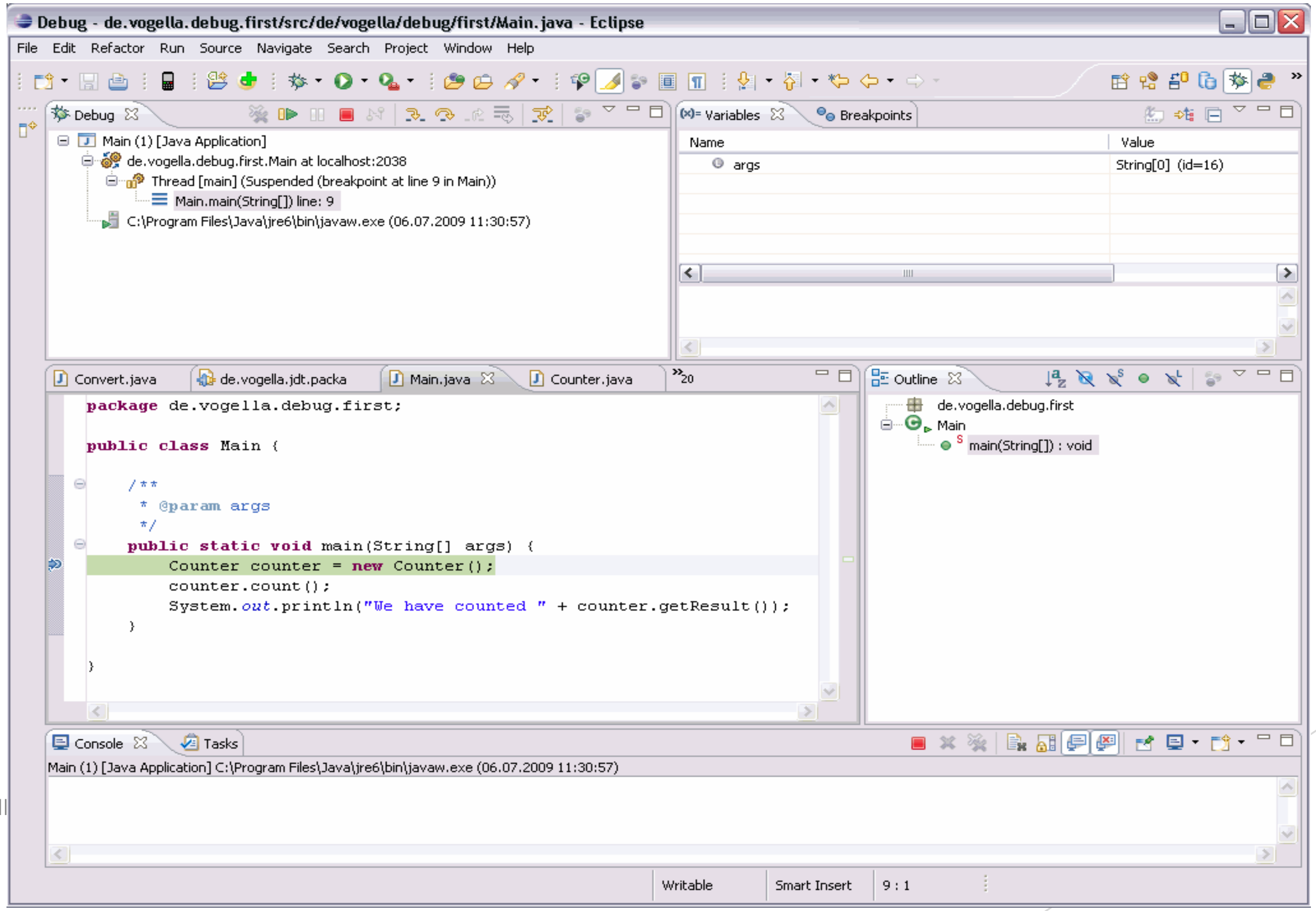
- After you have started the application once via the context menu, you can use the created launch configuration again via the Debug button in the Eclipse toolbar.



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

Debug Perspective

- Afterwards Eclipse opens the *Debug perspective*, which looks similar to the following screenshot:


Debug - de.vogella.debug.first/src/de/vogella/debug/first/Main.java - Eclipse
File Edit Refactor Run Source Navigate Search Project Window Help
Debug
Variables
Breakpoints
Convert.java
de.vogella.jdt.packa
Main.java
Counter.java
»20
Outline
Console
Tasks
Writable
Smart Insert
9 : 1

```
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)
```

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

```java
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());
    }

}
```

```
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)
```

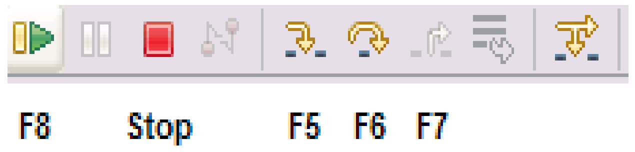
Controlling Program Execution

- 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.

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 is reaches the next breakpoint or watchpoint.

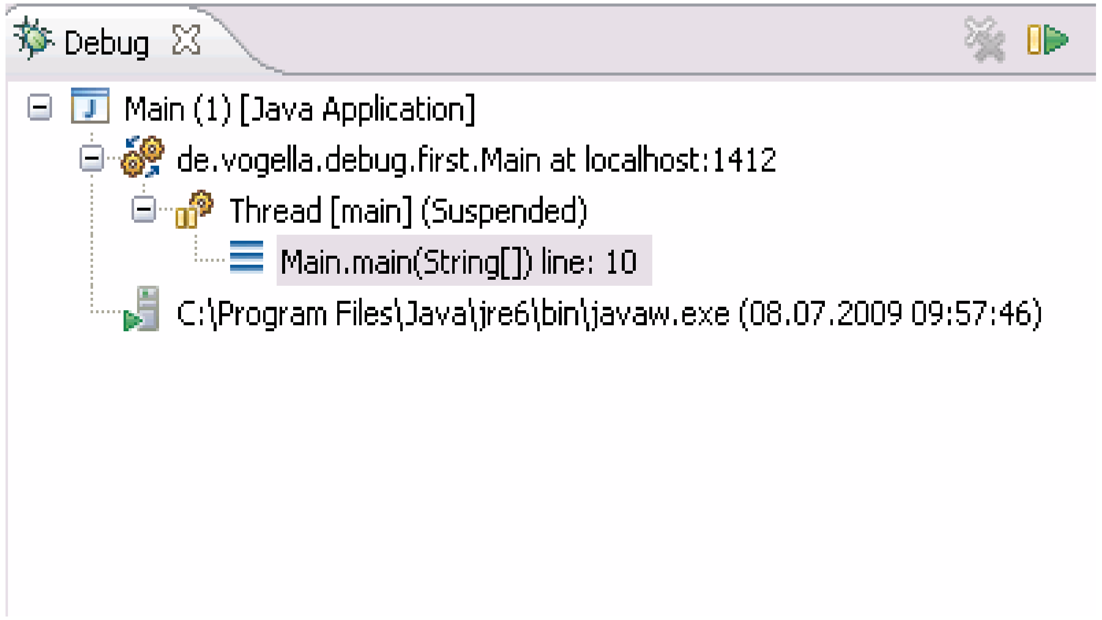
Controlling Program Execution

- The following picture displays the buttons and their related keyboard shortcuts.



The Call Stack

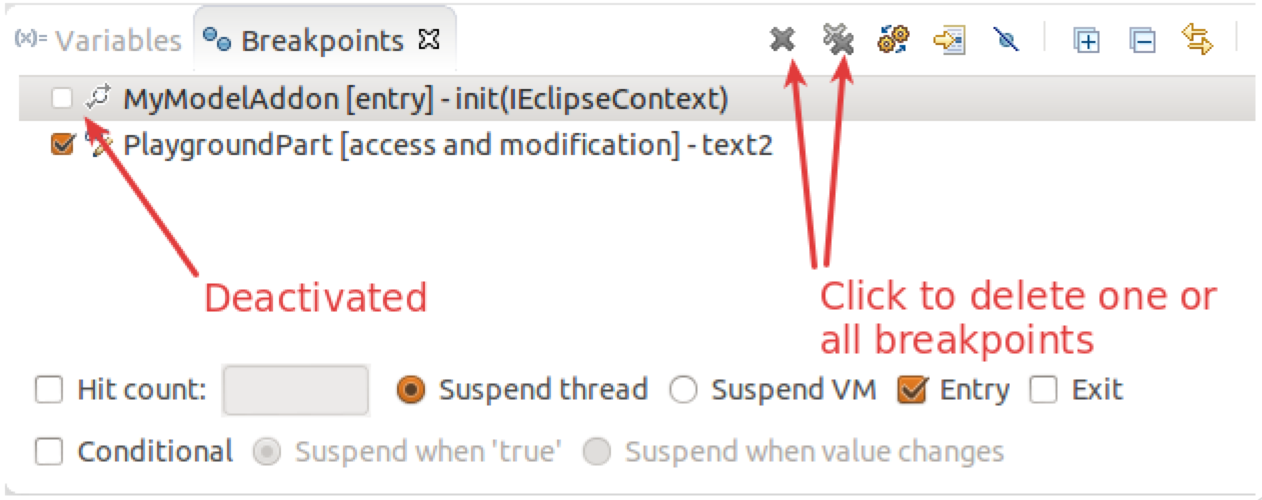
- The *call stack* shows the parts of the program which are currently executed and how they relate to each other. The current stack is displayed in the Debug view .



```
Main (1) [Java Application]
  de.vogella.debug.first.Main at localhost:1412
    Thread [main] (Suspended)
      Main.main(String[]) line: 10
  C:\Program Files\Java\jre6\bin\javaw.exe (08.07.2009 09:57:46)
```

Breakpoints View

- 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.

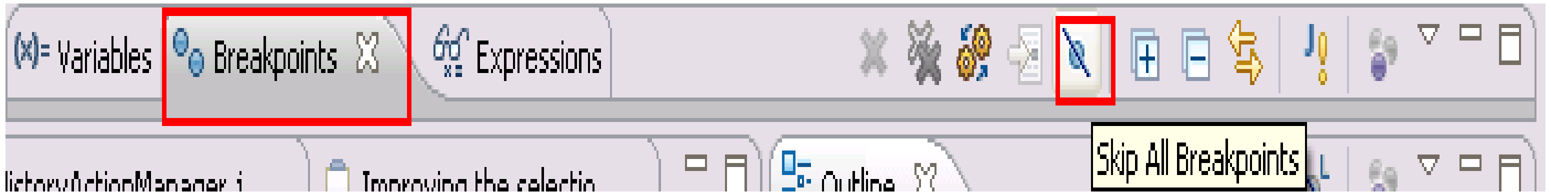


[ ] MyModelAddon [entry] - init(IEclipseContext)
[✓] PlaygroundPart [access and modification] - text2

[ ] Hit count: [● Suspend thread] ○ Suspend VM [✓ Entry] [ ] Exit
[ ] Conditional ○ Suspend when 'true' ○ Suspend when value changes

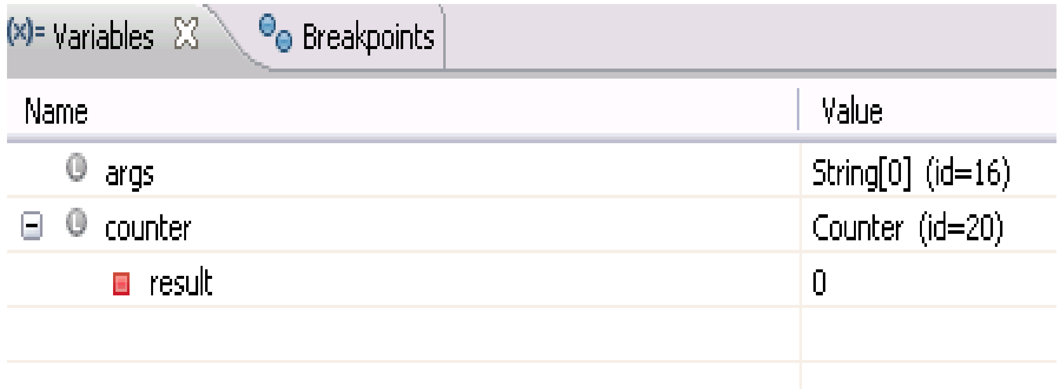
Deactivating All Breakpoints

- 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.



Evaluating Variables

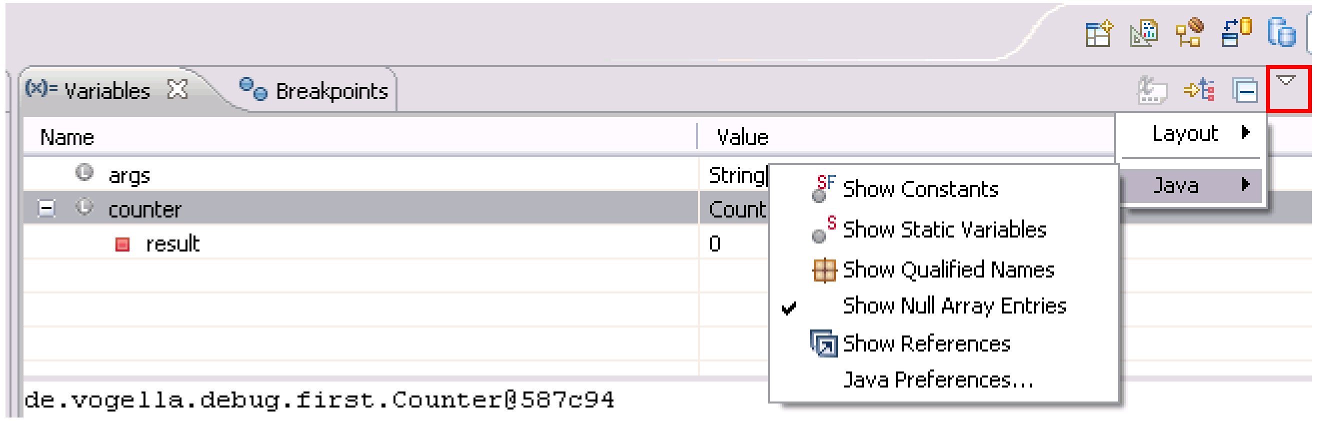
- The Variables view displays fields and local variables from the current executing stack.



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

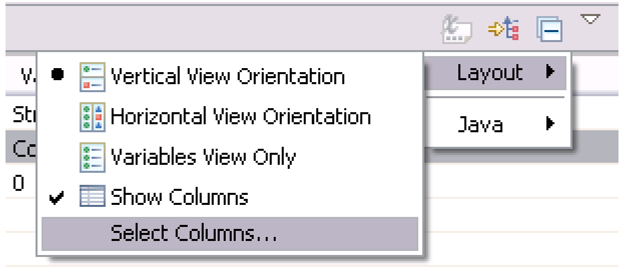
Evaluating Variables

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



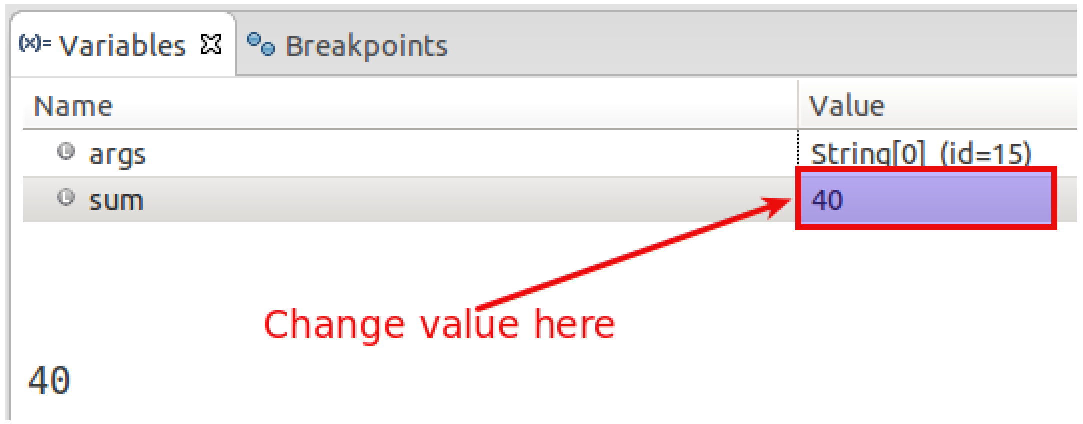
```
de.vogella.debug.first.Counter@587c94
```

- 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

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

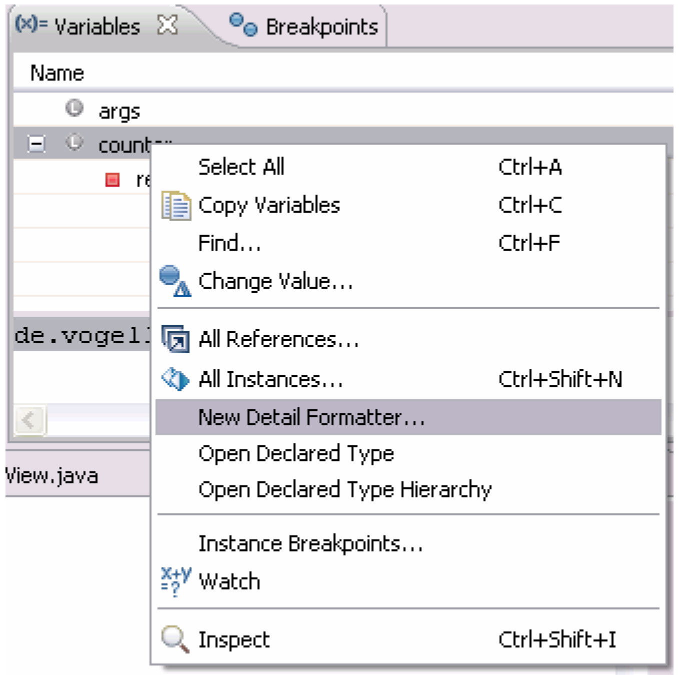


Name	Value
args	String[0] (id=15)
sum	40

```
40
```

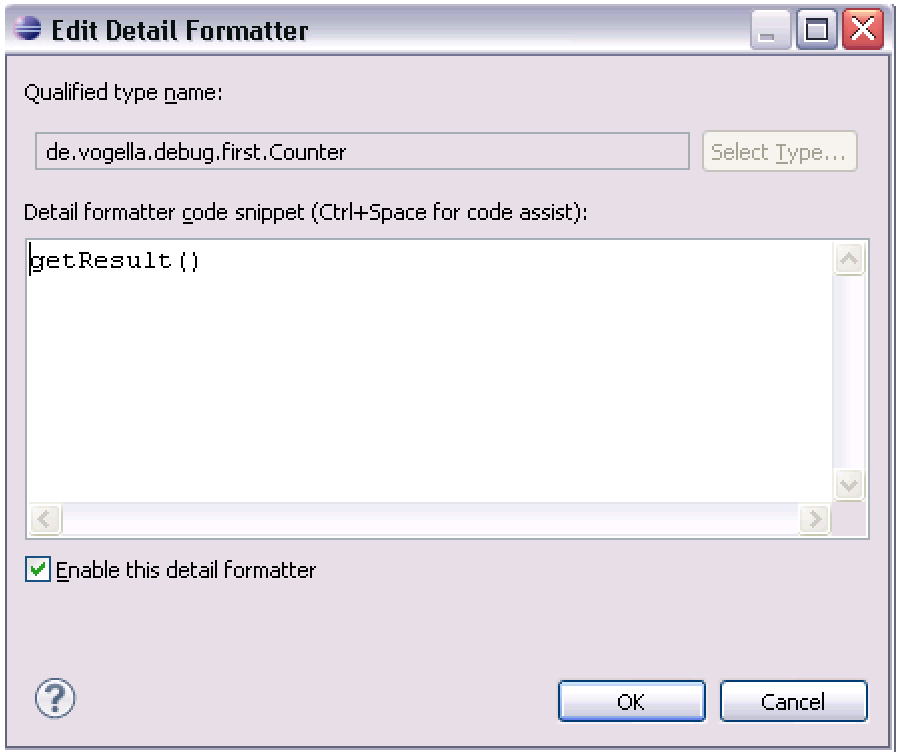
Controlling Display of Variables

- 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. 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 Display of Variables

- Then you can use any class method to display as output.
- For example, the *getResult()* method of this class can be used.



Qualified type name:

de.vogella.debug.first.Counter

Detail formatter code snippet (Ctrl+Space for code assist):

```
getResult()
```

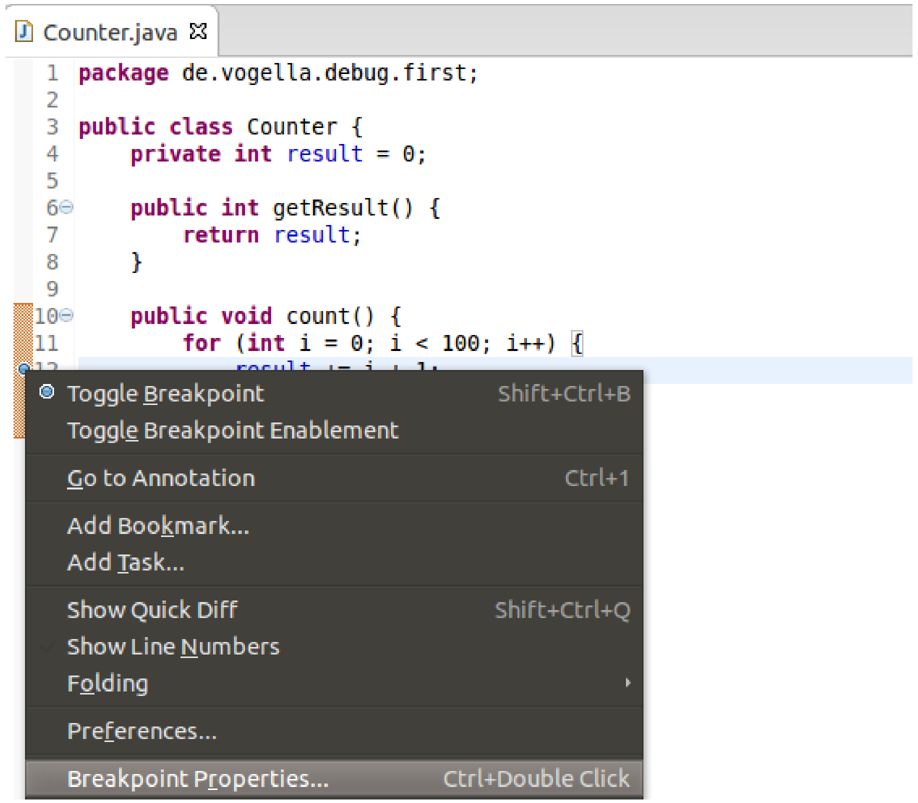
[✓] Enable this detail formatter

Breakpoint Properties

- 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.
- For instance, every breakpoint has a *Hit Count* property. The application is stopped once the breakpoint has been reached the number of times defined in the hit count.
    - As an example, you can specify that a breakpoint should only become active after it has reached 12 or more times via its *Hit Count* property.

Conditional Breakpoints

- For each breakpoint, you can also create a conditional expression. The execution of the program only stops at the breakpoint, if the condition evaluates to true. Can be used for additional logging, as the code that specifies the condition is executed every time the program execution reaches that point.

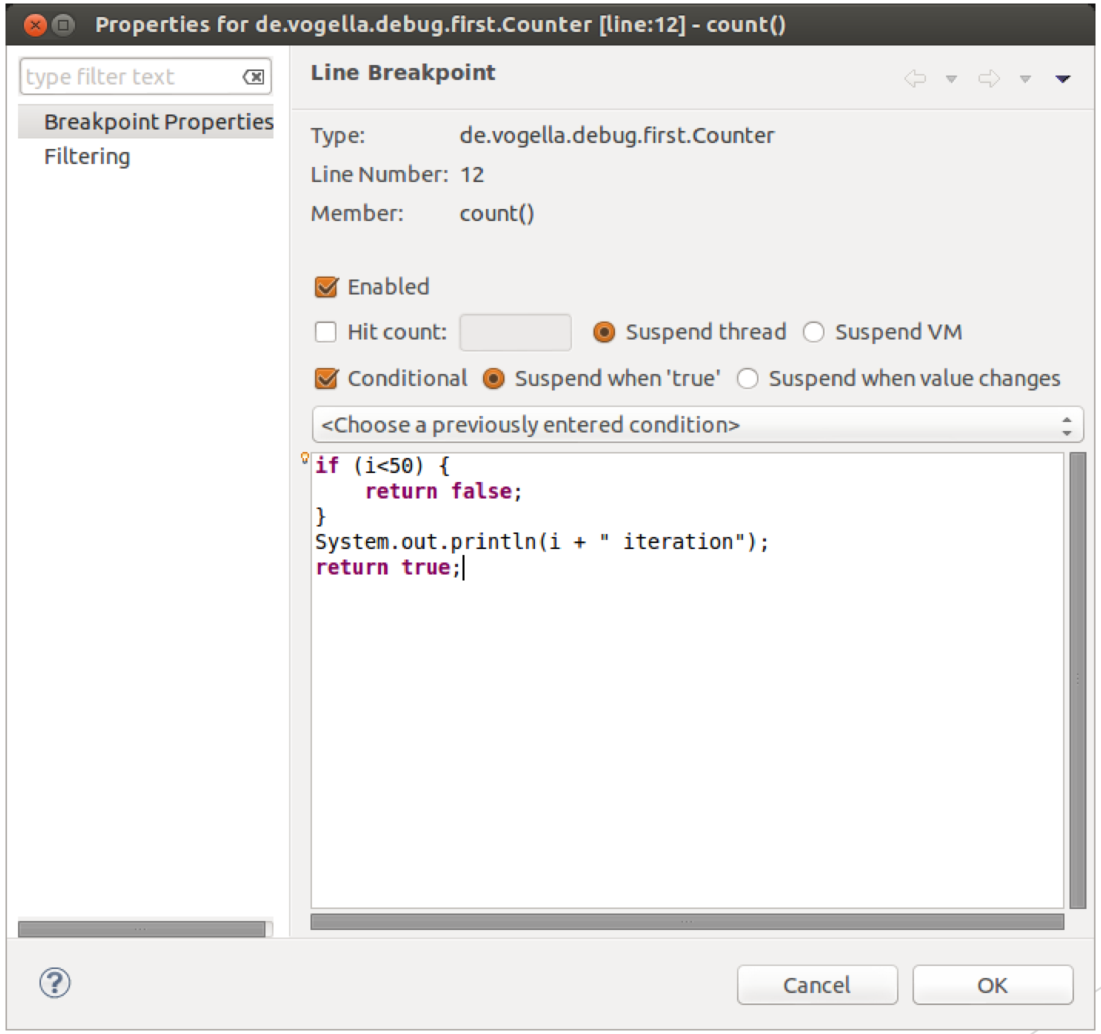


```java
package de.vogella.debug.first;

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;
```

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

Conditional Breakpoints

- For example, the following conditional breakpoint will not do anything until the value of the variable *i* is equal to or greater than 50.
- At that point, it will print out the value of *i* and suspend the running of the program like a typical breakpoint.



Properties for de.vogella.debug.first.Counter [line:12] - count()

type filter text

Breakpoint Properties
Filtering

Line Breakpoint

Type: de.vogella.debug.first.Counter
Line Number: 12
Member: count()

[✓] Enabled
[ ] Hit count: [✓ Suspend thread] Suspend VM
[✓] Conditional [✓ Suspend when 'true'] Suspend when value changes

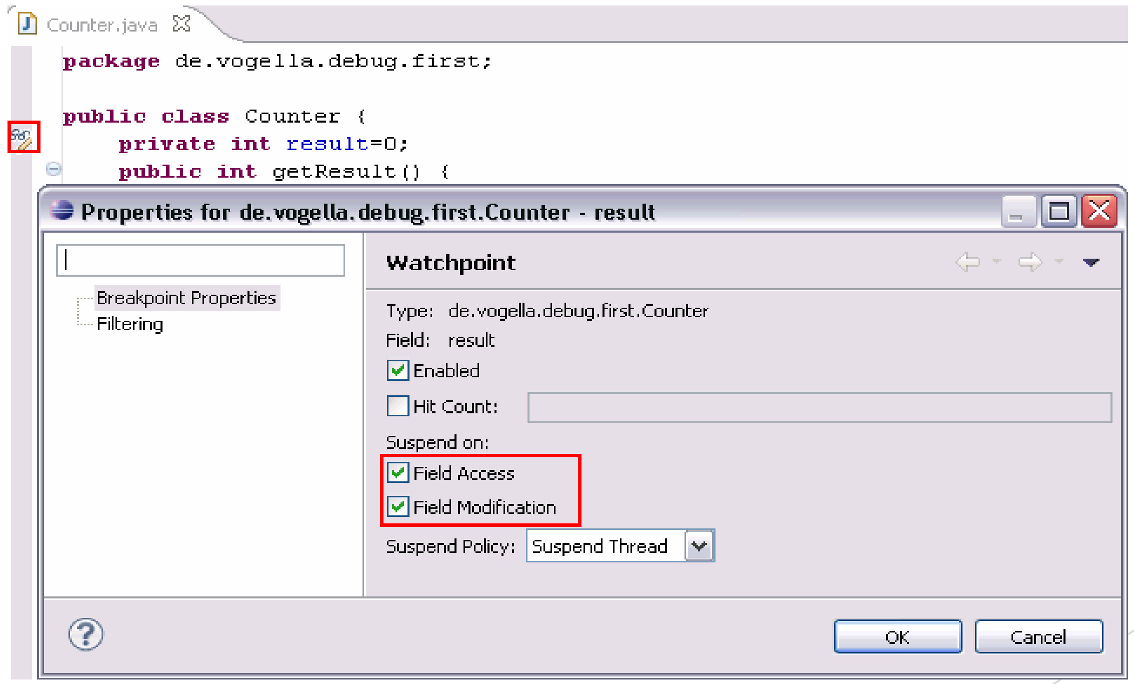
<Choose a previously entered condition>

```
if (i<50) {
    return false;
}
System.out.println(i + " iteration");
return true;
```

Cancel OK

Watchpoint

- 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 specify whether to stop during read access (Field Access) or during write access (Field Modification) or both.



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

public class Counter {
    private int result=0;
    public int getResult() {
```

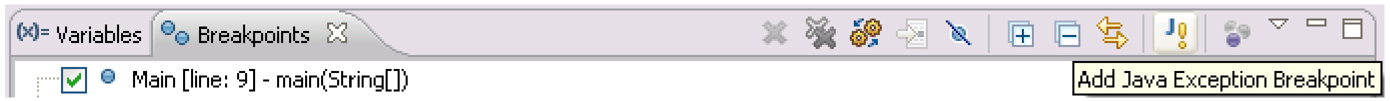
Properties for de.vogella.debug.first.Counter - result

Watchpoint

Type: de.vogella.debug.first.Counter
Field: result
[✓] Enabled
[ ] Hit Count:
Suspend on:
[✓] Field Access
[✓] Field Modification
Suspend Policy: Suspend Thread

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.

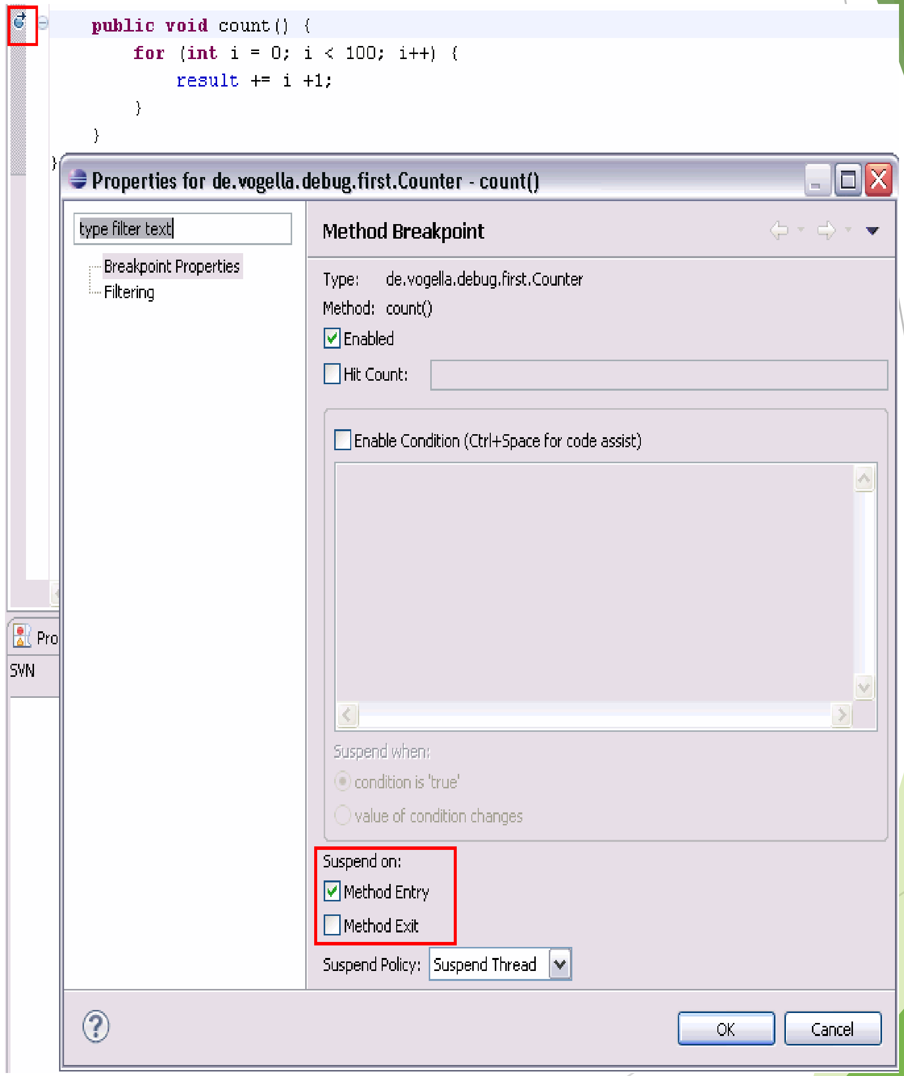


Main [line: 9] - main(String[])

- That way, you can configure the debugger to stop at caught or uncaught exceptions.

Method Breakpoints

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



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

Properties for de.vogella.debug.first.Counter - count()

Method Breakpoint

Type: de.vogella.debug.first.Counter

Method: count()

[✓] Enabled

[ ] Hit Count:

[ ] Enable Condition (Ctrl+Space for code assist)

Suspend when:

- ○ condition is 'true'
- ○ value of condition changes

Suspend on:

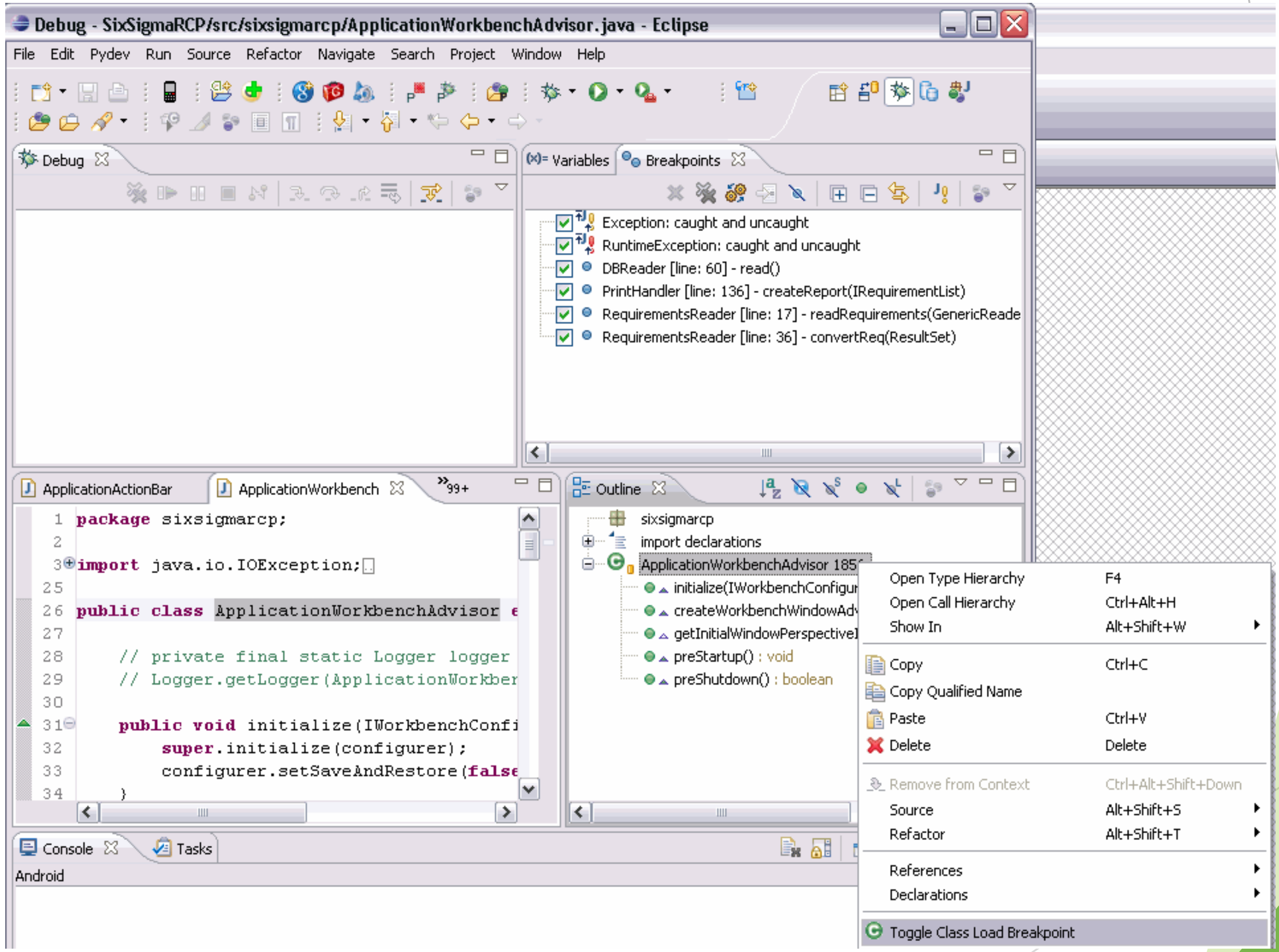
[✓] Method Entry

[ ] Method Exit

Suspend Policy: Suspend Thread

Breakpoints for Loading Classes

- 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.
- Or you can double-click in the left border of the Java editor beside the class definition.



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)

```
package sixsigmarcp;

import java.io.IOException;

public class ApplicationWorkbenchAdvisor e

    // private final static Logger logger
    // Logger.getLogger(ApplicationWorkber

    public void initialize(IWorkbenchConfi
        super.initialize(configurer);
        configurer.setSaveAndRestore(false
    }
```

sixsigmarcp
import declarations
ApplicationWorkbenchAdvisor 185
initialize(IWorkbenchConfigur
createWorkbenchWindowAdv
getInitialWindowPerspectiveI
preStartup() : void
preShutdown() : boolean

Acknowledgements

- Slide show is based on information found in the article **Java Debugging with Eclipse - Tutorial** by **Lars Vogel,** Version 2.3, copyright © 2009, 2010, 2011, 2012, 2013 vogella GmbH 24.01.2013
- The article can be found at: http://www.vogella.com/tutorials/EclipseDebugging/article.html