Table of Contents

1. Overview

1.1. 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.

1.2. Debugging support in Eclipse

Table of Contents

1. Overview

1.1. 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.

1.2. 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.

2. Prerequisites

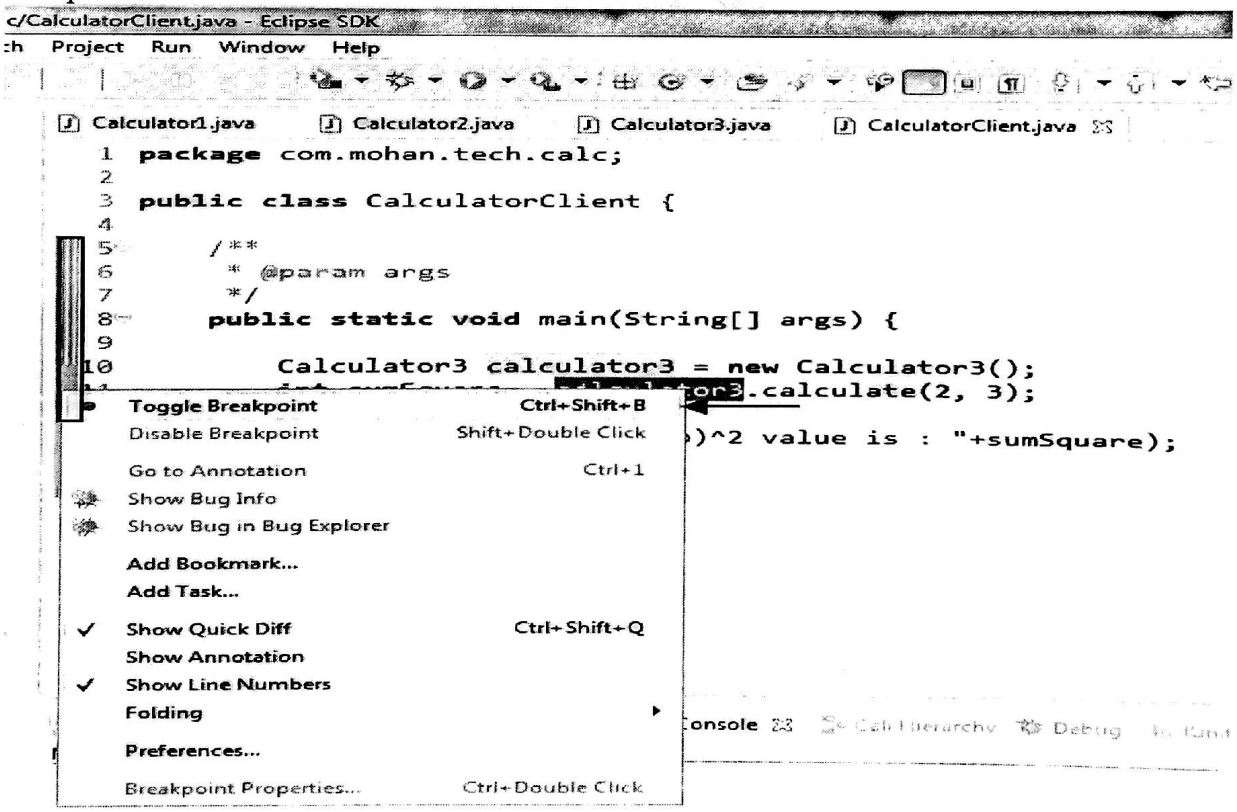
The following assumes you know how to develop simple standard Java programs. This article will focus on how to debug Java applications in Eclipse.

The installation and usage of Eclipse as Java IDE is described in Eclipse Java IDE Tutorial.

3. Debugging in Eclipse

3.1. Setting Breakpoints

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.



For example in the following screen shot we set a breakpoint on the line

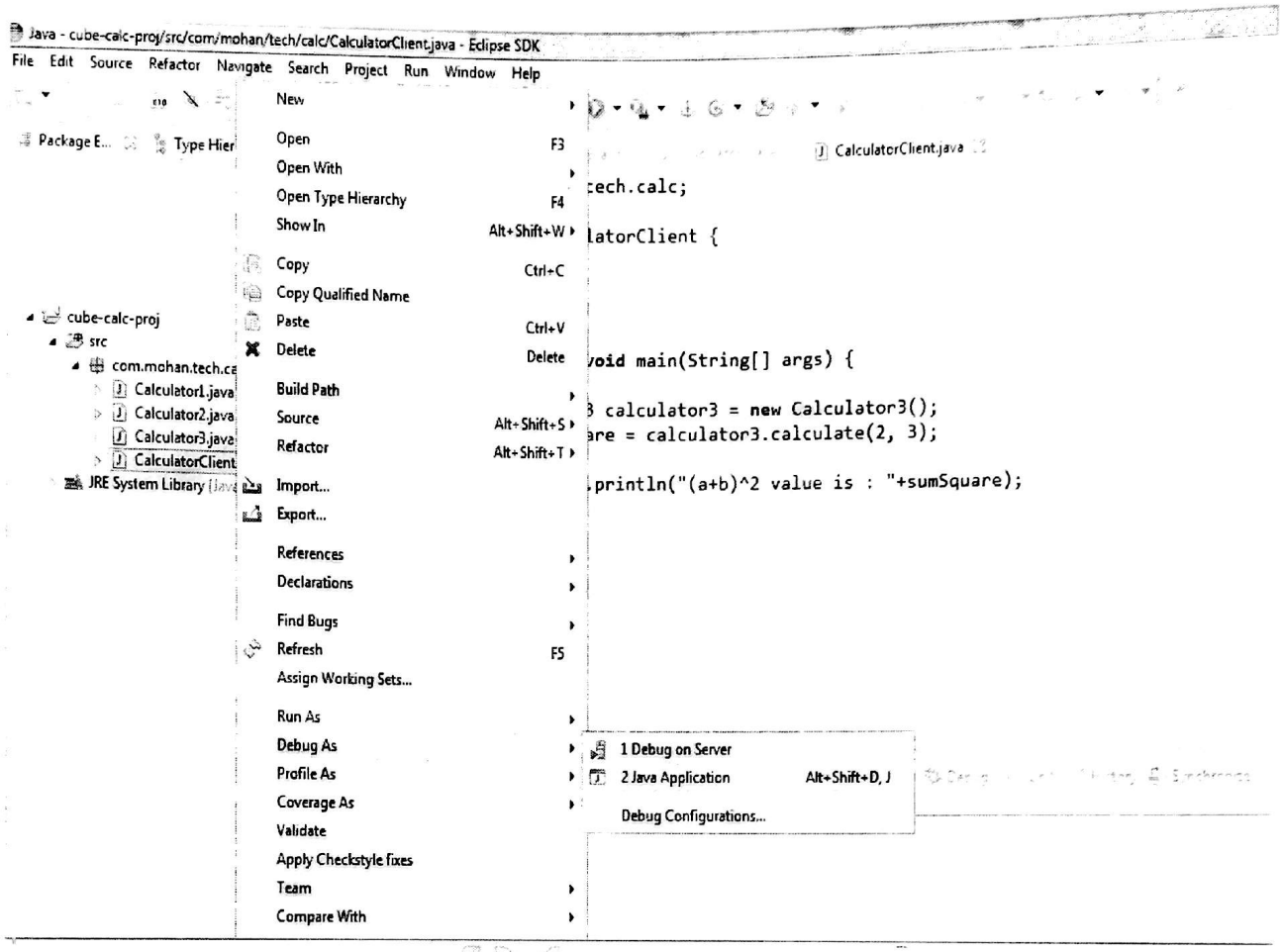
int sumSquare = calculator3.calculate(2, 3);

Calculator1.java | Calculator2.java | Calculator3.java | CalculatorClient.java

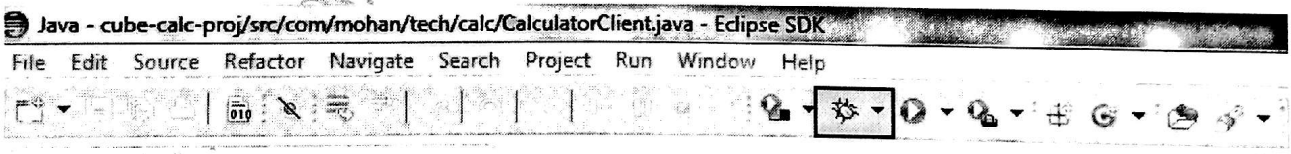
```
1  package com.mohan.tech.calc;
2
3  public class CalculatorClient {
4
5      /**
6       * @param args
7       */
8      public static void main(String[] args) {
9
10         Calculator3 calculator3 = new Calculator3();
11         int sumSquare = calculator3.calculate(2, 3);
12
13         System.out.println("(a+b)^2 value is : "+sumSquare);
14
15     }
16
17 }
18
```

3.2. Starting the Debugger

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

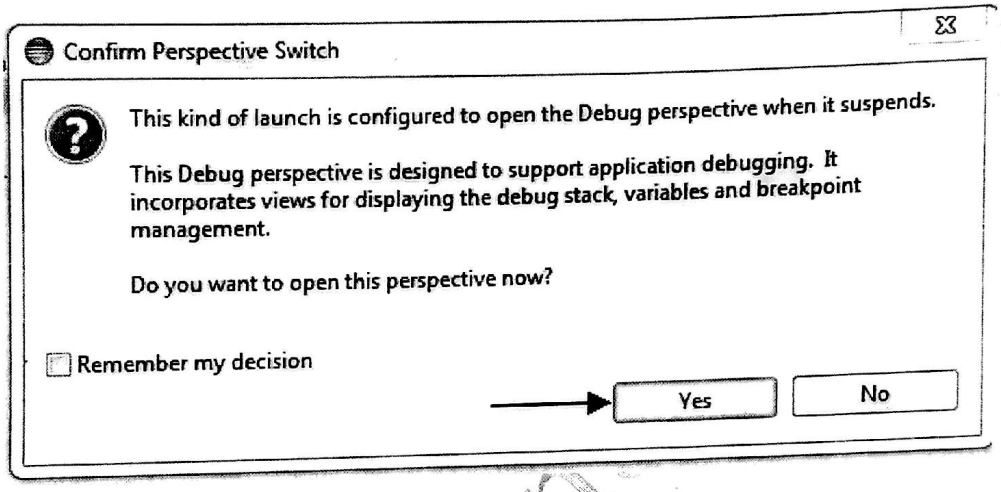


After you have started the application once via the context menu, you can use the created launch configuration again via the Debug button in the Eclipse toolbar.

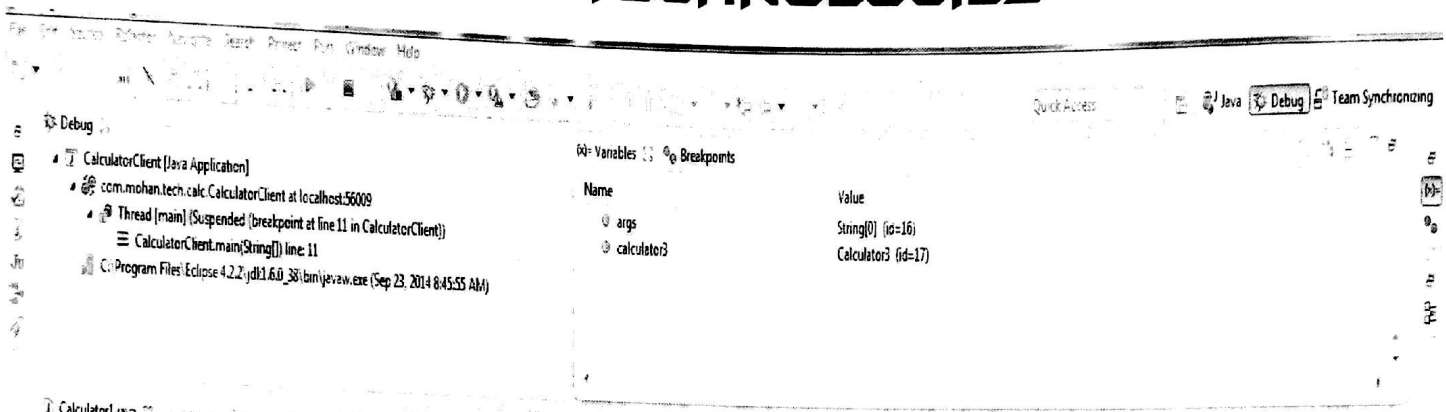


If you have not defined any breakpoints, this will run your program as normal. To debug the program you need to define breakpoints.

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.



Calculator1.java

```
package com.mohan.tech.calc;

public class CalculatorClient {

    /**
     * @param args
     */
    public static void main(String[] args) {

        Calculator3 calculator3 = new Calculator3();
        int sumSquare = calculator3.calculate(2, 3);

        System.out.println("(a+b)^2 value is : "+sumSquare);

    }

}
```

3.3. Controlling the 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.

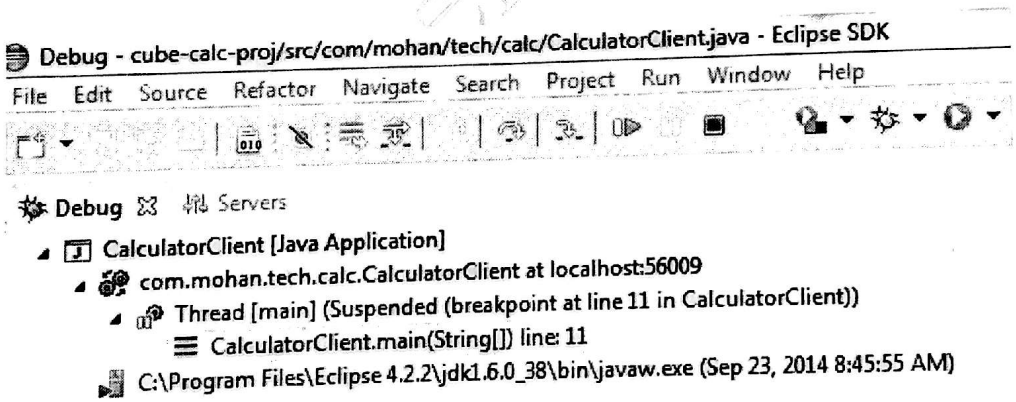
You can use the F5, F6, F7 and F8 key to step through your coding. The meaning of these keys is explained in the following table.

Table 1. Debugging key bindings / shortcuts

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.

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

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 .

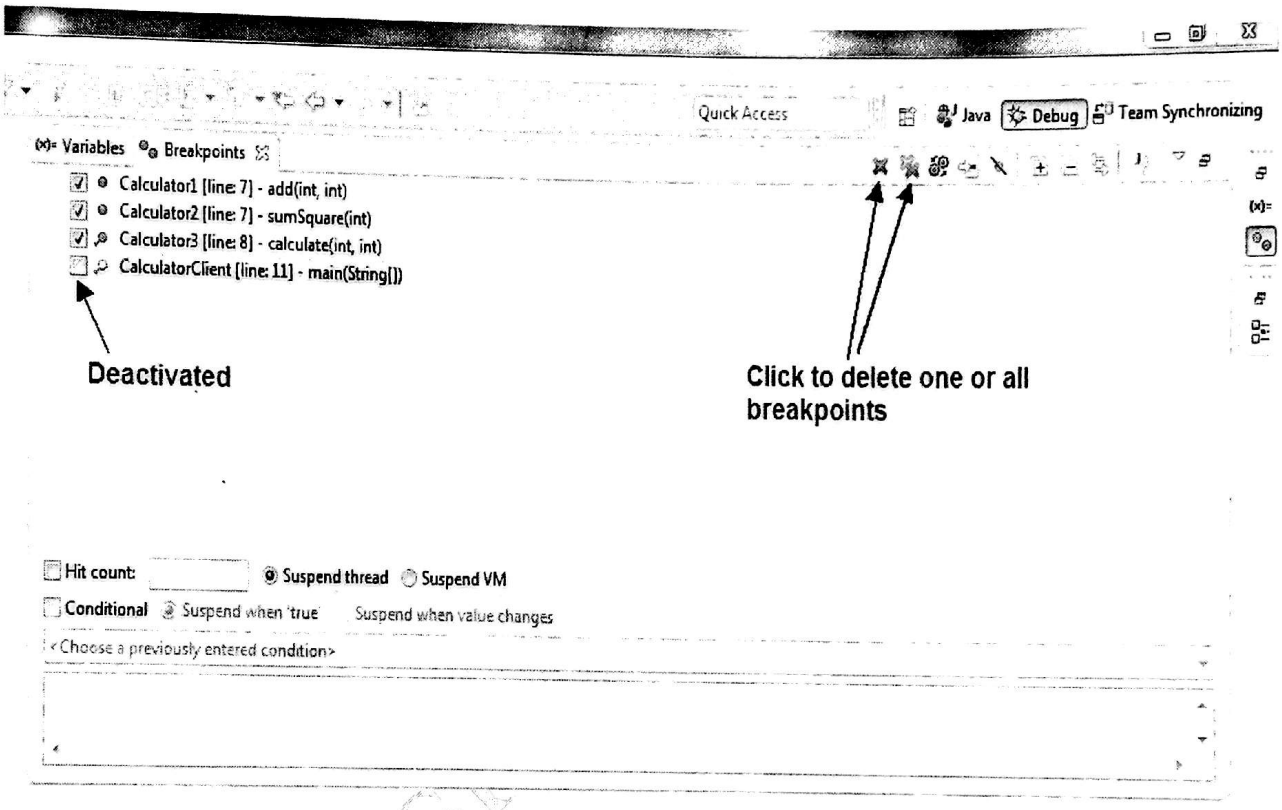


3.4. Breakpoints view and deactivating breakpoints

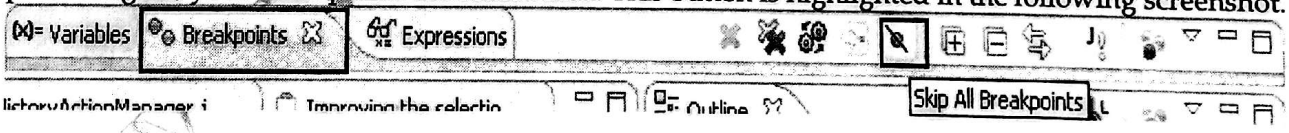
The Breakpoints view allows you to delete and deactivate stop points, i.e. breakpoints and

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.

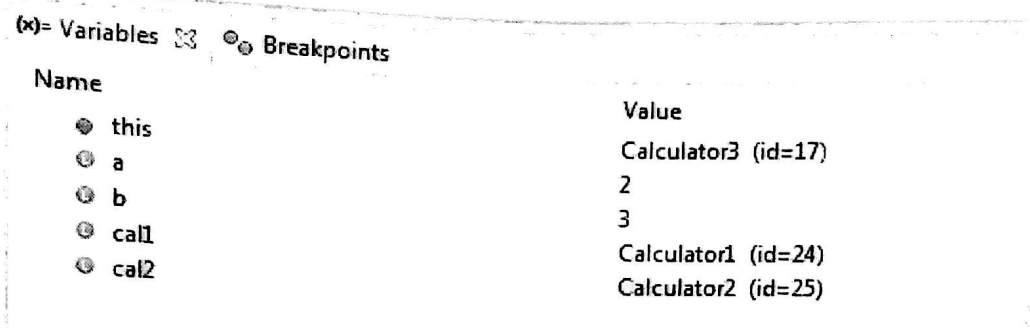


If you want to deactivate all your breakpoints you can press the Skip all breakpoints button. If you press it again, your breakpoints are reactivated. This button is highlighted in the following screenshot.

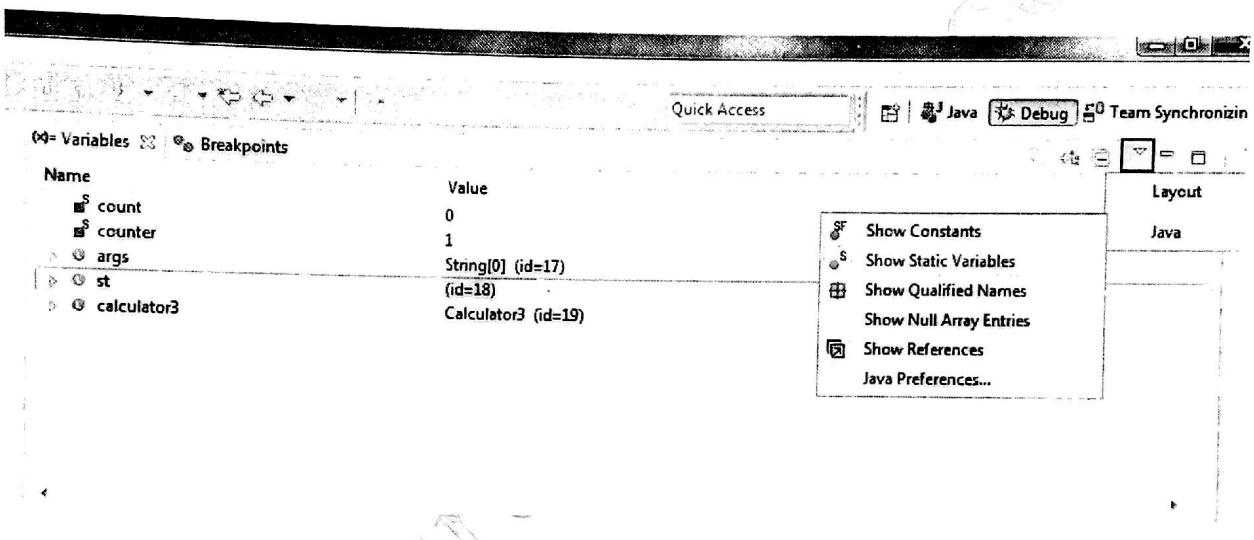


3.5. Evaluating variables in the debugger

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 .



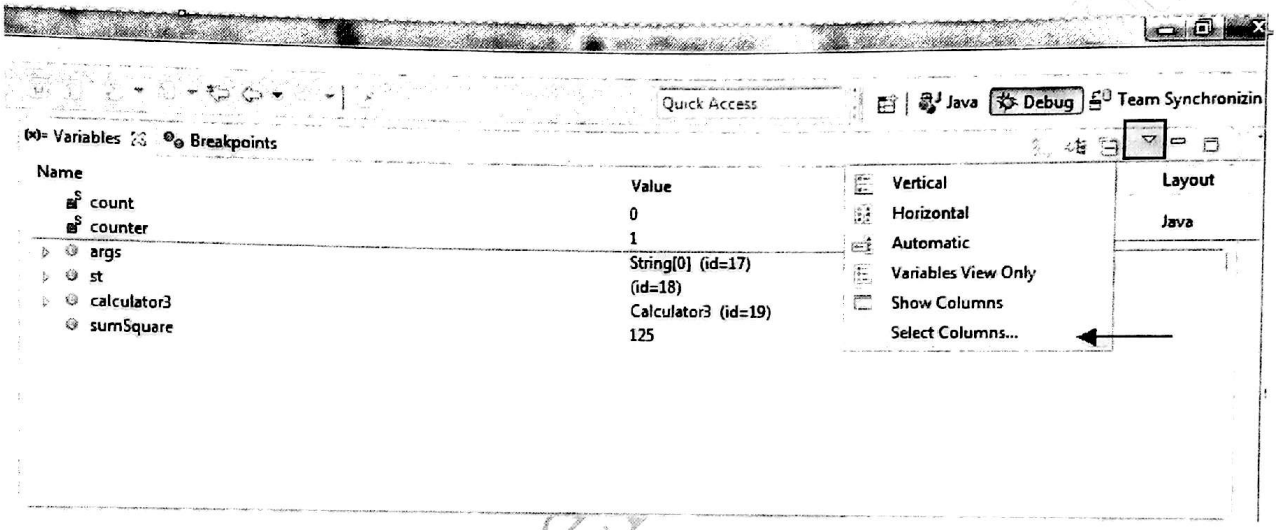
Use the drop-down menu to display static variables.



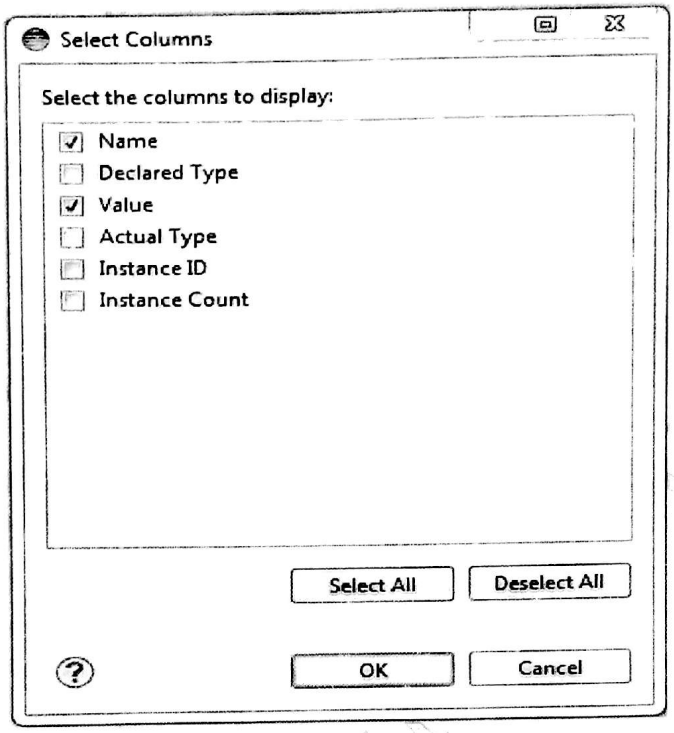
Via the drop-down menu of the Variables view you can customize the displayed columns. For example, you can show the actual type of each variable declaration. For this select Layout → Select Columns... → Type.

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.

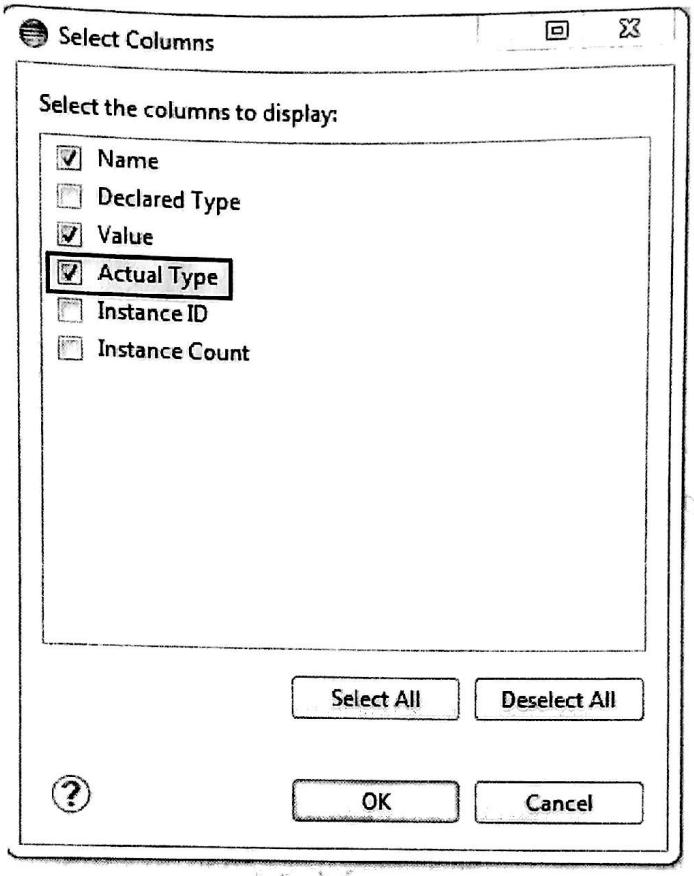
Via the drop-down menu of the Variables view you can customize the displayed columns. For example, you can show the actual type of each variable declaration. For this select Layout → Select Columns... → Type.

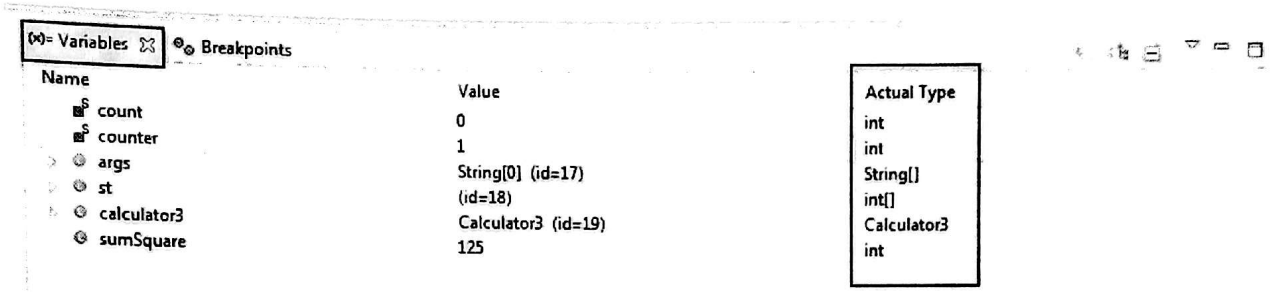


When selecting upon Select Columns..., will get one '**Select Columns**' window like below. Here you can choose what values required.



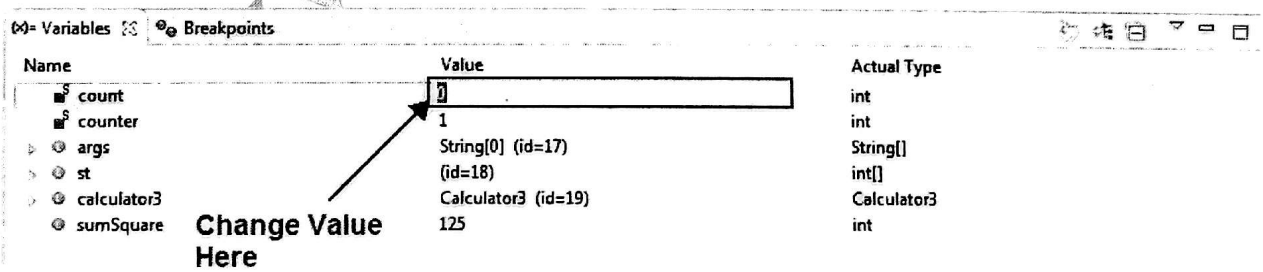
Form the above screen select required checkbox types. So, that the selected value added as a column in the variables view. So, I am going to select the **'Actual Type'** from the **Select columns window.**





3.6. Changing variable assignments in the debugger

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

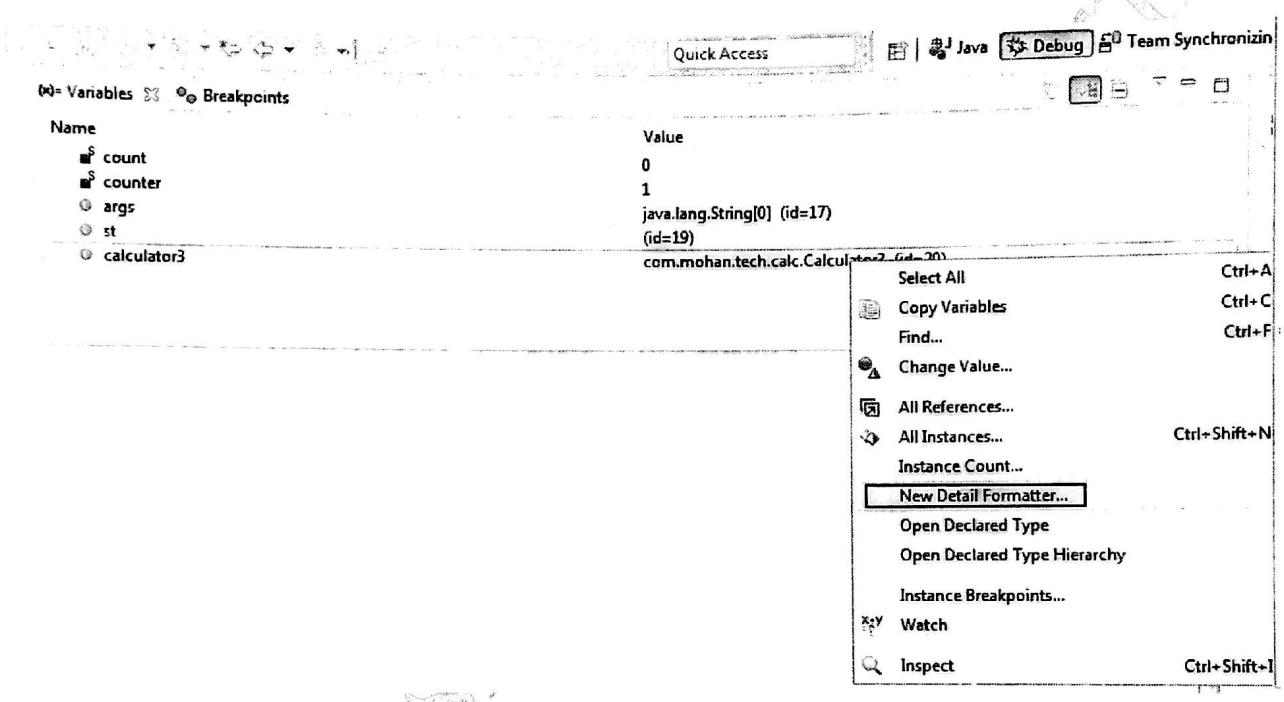


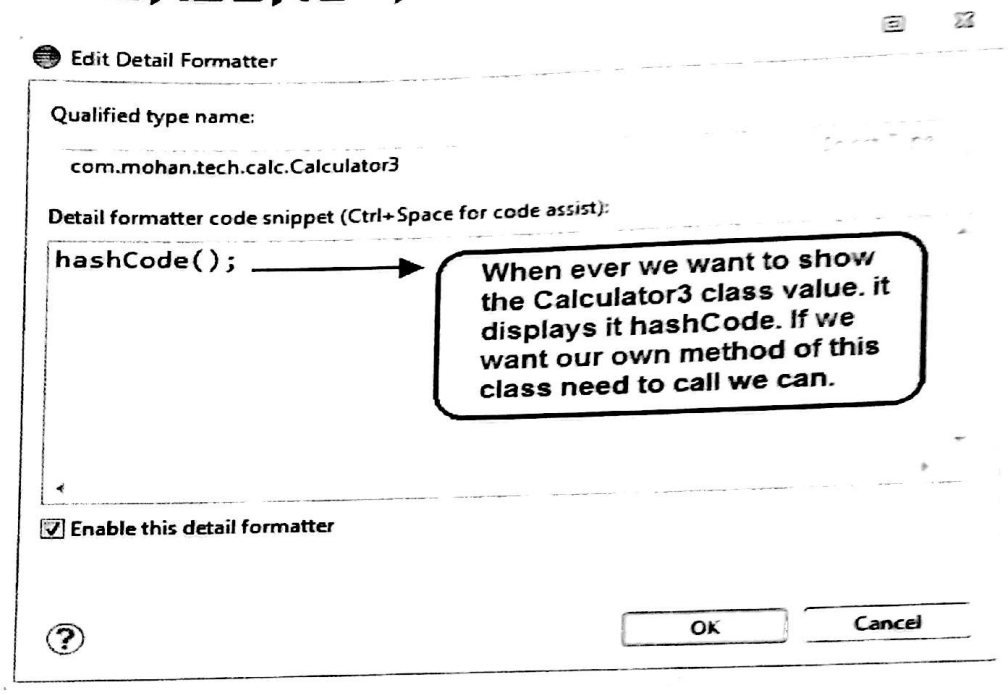
3.7. Controlling the display of the variables with Detail Formatter

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.

For example the toString() method in the Counter class may show meaningless information, e.g. **com.mohan.tech.calc.Calculator3@7e0c2ff5**. To make this output more readable you can right-click on the corresponding variable and select the New Detail Formater entry from the context menu.





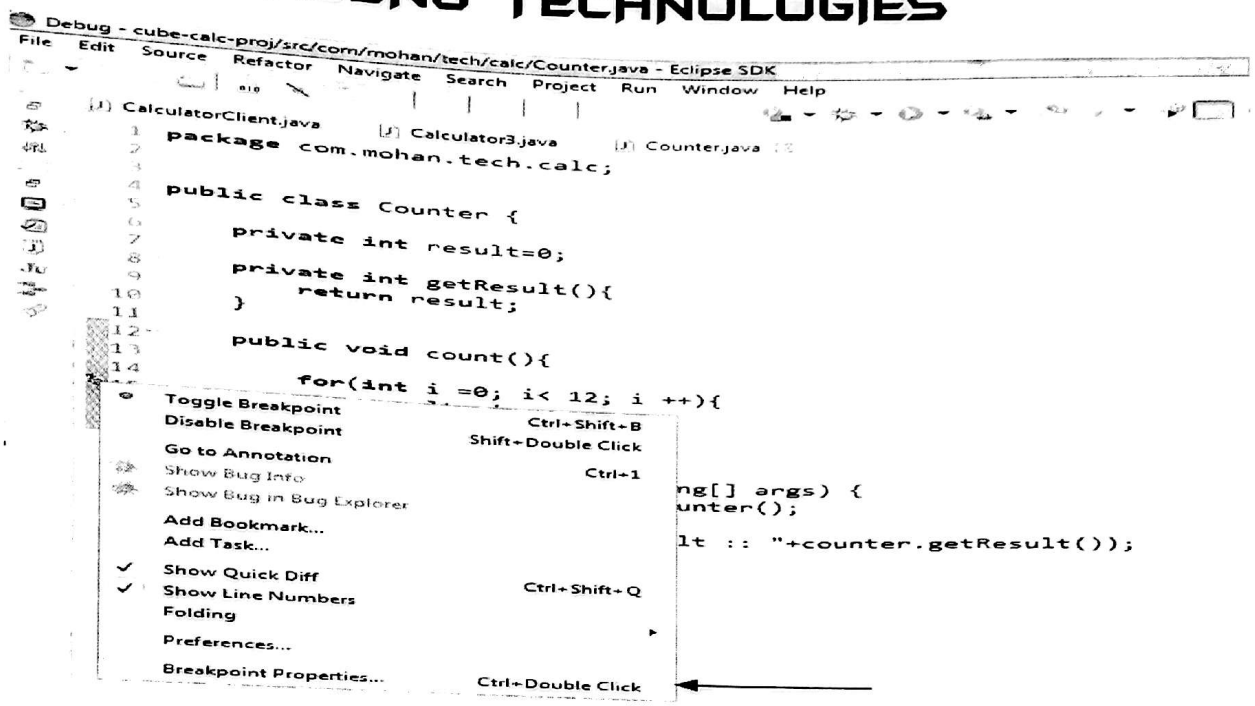
4.1. Breakpoint properties

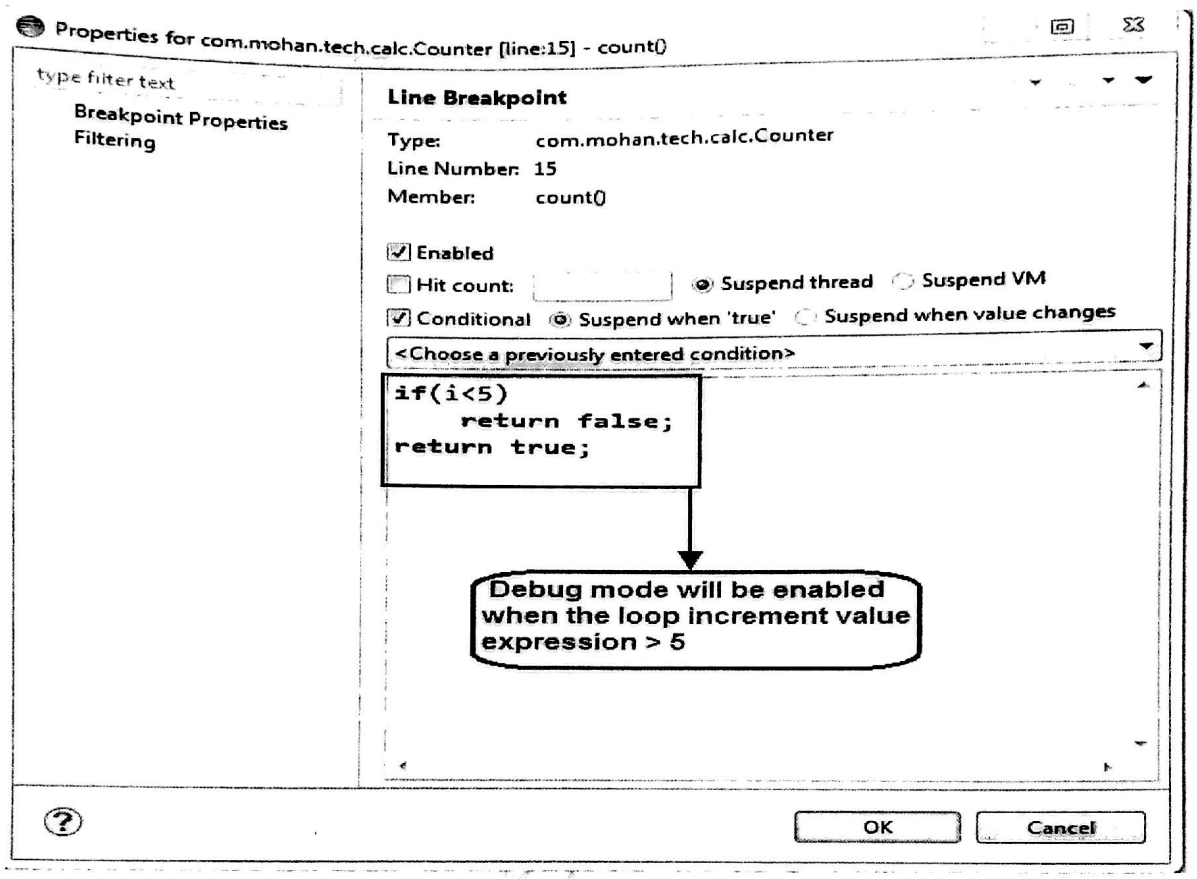
After setting a breakpoint you can select the properties of the breakpoint, via right-click → Breakpoint Properties. Via the breakpoint properties you can define a condition that restricts the activation of this breakpoint.

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.

The following screenshot depicts this setting.





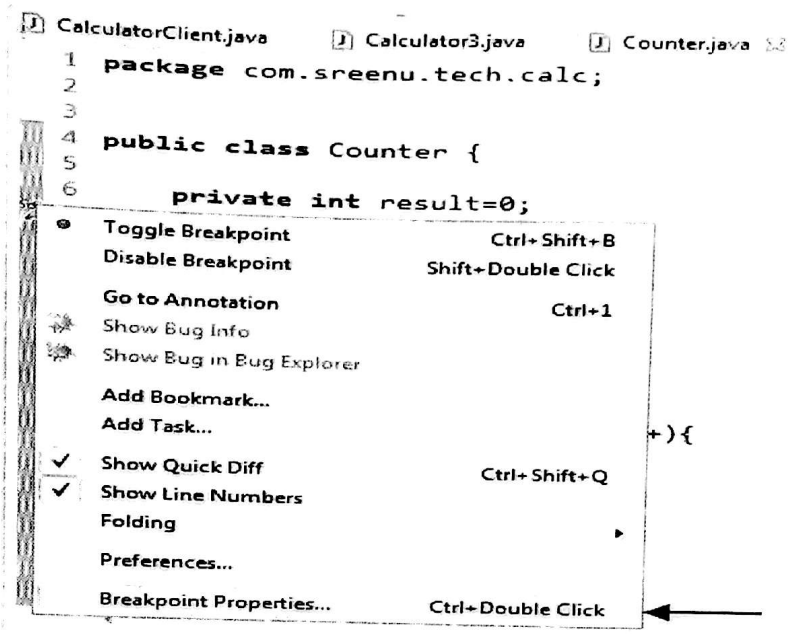
4.2. Watchpoint

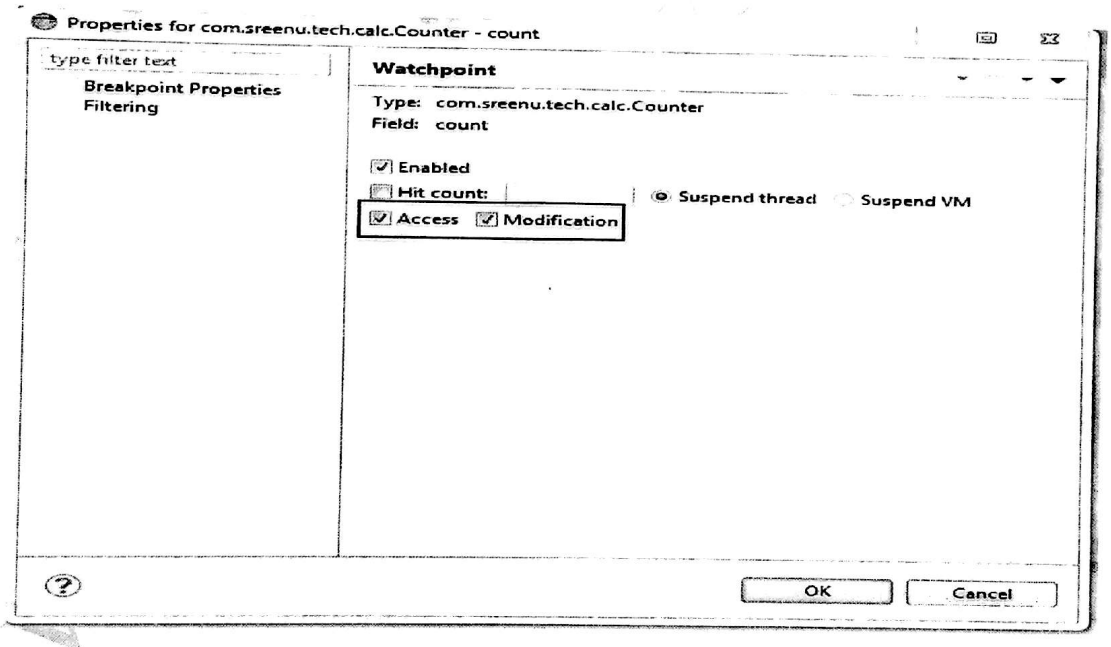
A watchpoint is a breakpoint set on a field. The debugger will stop whenever that field is read or changed.

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 configure if the execution should stop during read access (Field Access) or during write access (Field Modification) or both.

CalculatorClient.java | Calculator3.java | Counter.java

```
1  package com.sreenu.tech.calc;
2
3
4  public class Counter {
5
6      private int result=0;
7      int count =0;
8
9      private int getResult(){
10         return result;
11     }
12
13     public void count(){
14
15         for(int i =0; i< 12; i ++){
16             result+=i;
17             if(i == 8)
18                 count++;
19
20         }
21     }
22
```

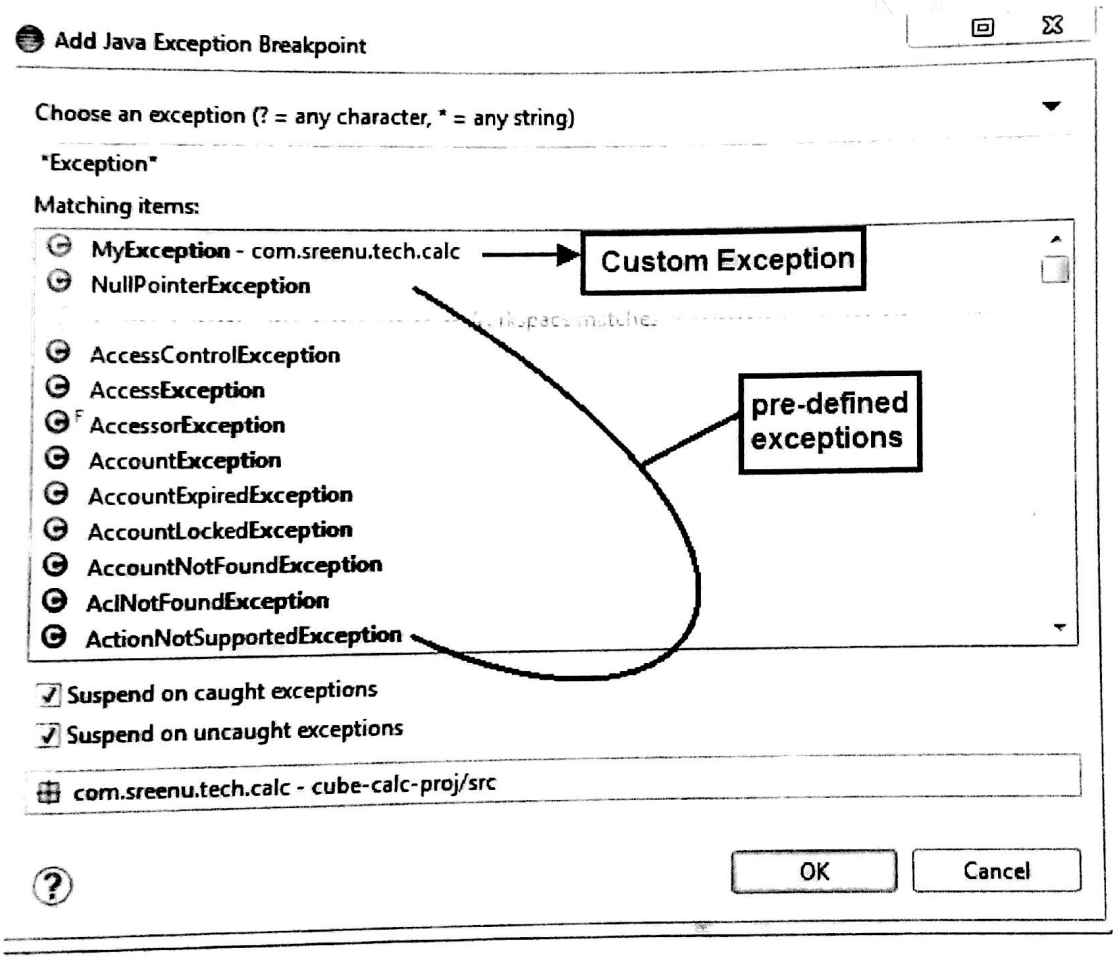




4.3. Exception breakpoints

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.

When we click on the above marked Icon then, the below window will be populated. From this window we can choose any of the pre-defined exceptions or custom exceptions which are configured in the application.



CalculatorClient.java | Calculator3.java | Counter.java | MyException.java

```
1  package com.sreenu.tech.calc;
2
3
4  public class Counter {
5
6      private int result=0;
7      int count =0;
8
9      private int getResult(){
10         return result;
11     }
12
13     public void count(){
```

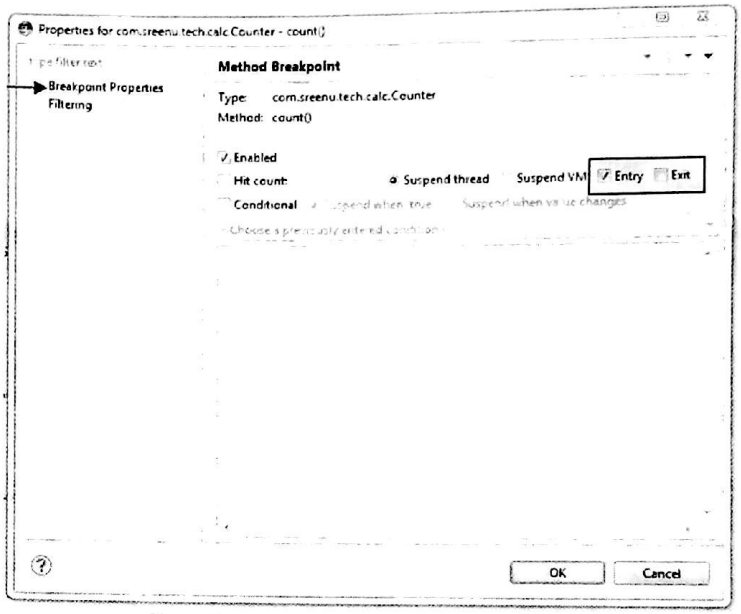
Toggle Breakpoint Ctrl+Shift+B
Disable Breakpoint Shift+Double Click
Go to Annotation Ctrl+1
Show Bug Info
Show Bug in Bug Explorer
Add Bookmark...
Add Task...
Show Quick Diff Ctrl+Shift+Q
Show Line Numbers
Folding
Preferences...
Breakpoint Properties... Ctrl+Double Click

```
od begin...");
+){

od leaving...");

ng[] args) {

unter();

String value:: "+str.toString());

ion n){
n);
30         }
31         System.out.println("Result :: "+counter.getResult());
```

#64/3RT, SR Nagar, Near Community Hall, Hyderabad
Contacts: +91-8019697596, 040-40061799 Email-id: sreenutechnologies@gmail.com
www. Sreenutech.com

CalculatorClient.java | Calculator3.java | Counter.java | MyException.java

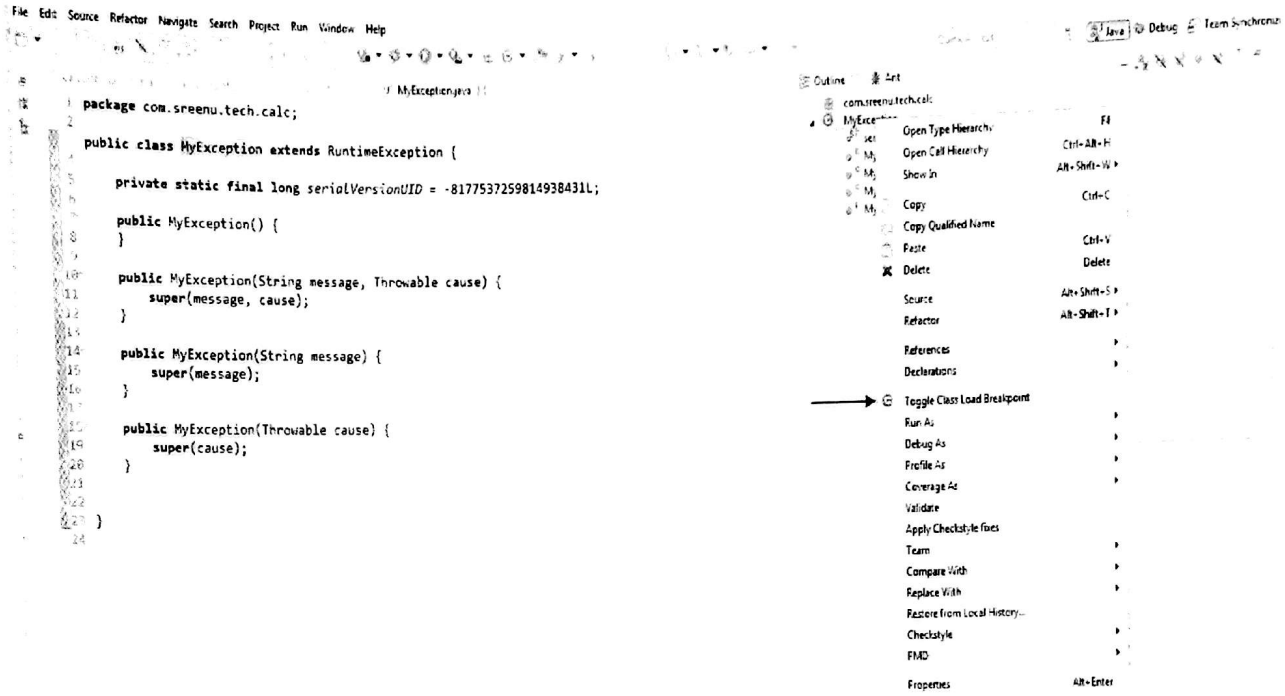
```
package com.sreenu.tech.calc;


public class Counter {

    private int result=0;
    int count =0;

    private int getResult(){
        return result;
    }

    public void count(){
        System.out.println("method begin...");
        for(int i =0; i< 12; i ++){
            result+=i;
        }
        System.out.println("method leaving...");
    }

    public static void main(String[] args) {
        String str = null;
        Counter counter = new Counter();
        counter.count();
        try{
            System.out.println("String value:: "
        }catch(NullPointerException n){
            throw new MyException(n);
        }
        System.out.println("Result :: "+counter.
    }
}
```



4.5. 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.



Now corresponding to the class name **'Toggle class load Breakpoint'** appeared.

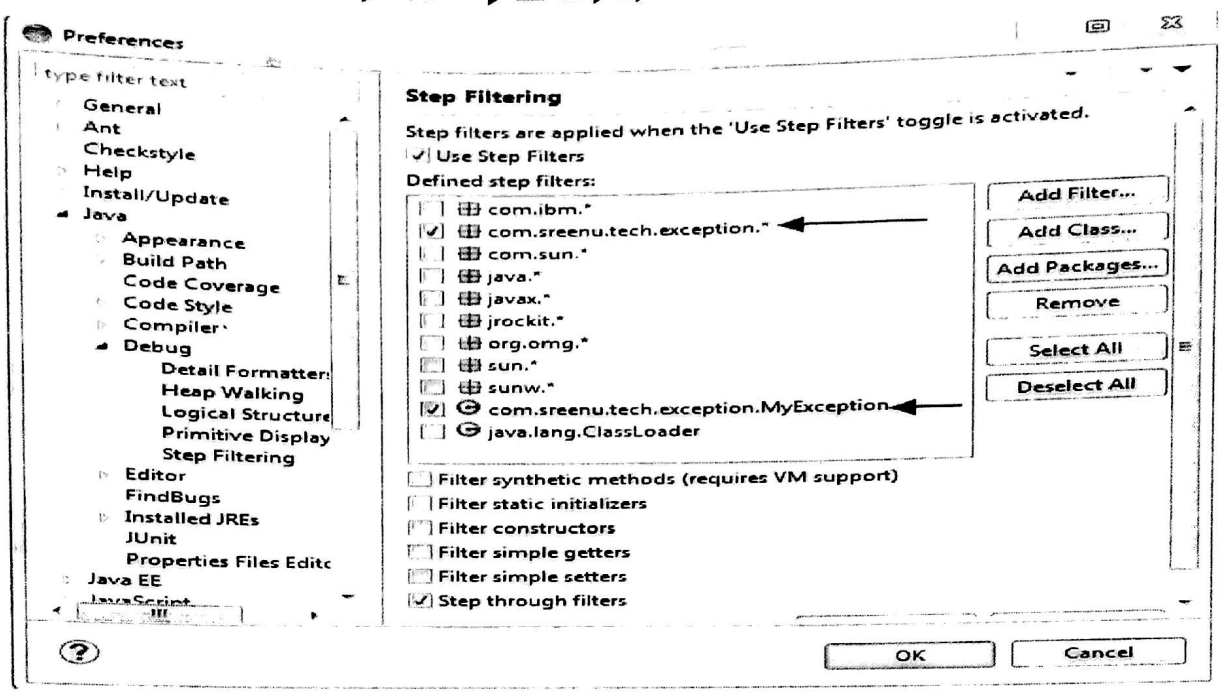
CalculatorClient.java | Calculator3.java | Counter.java | MyException.java

```
1  package com.sreenu.tech.calc;
2
3  public class MyException extends RuntimeException {
4
5      private static final long serialVersionUID = -8177537259814938431L;
6
7      public MyException() {
8      }
9
10     public MyException(String message, Throwable cause) {
11         super(message, cause);
12     }
13
14     public MyException(String message) {
15         super(message);
16     }
17
18     public MyException(Throwable cause) {
19         super(cause);
20     }
21
22
23 }
```

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

4.6. 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.

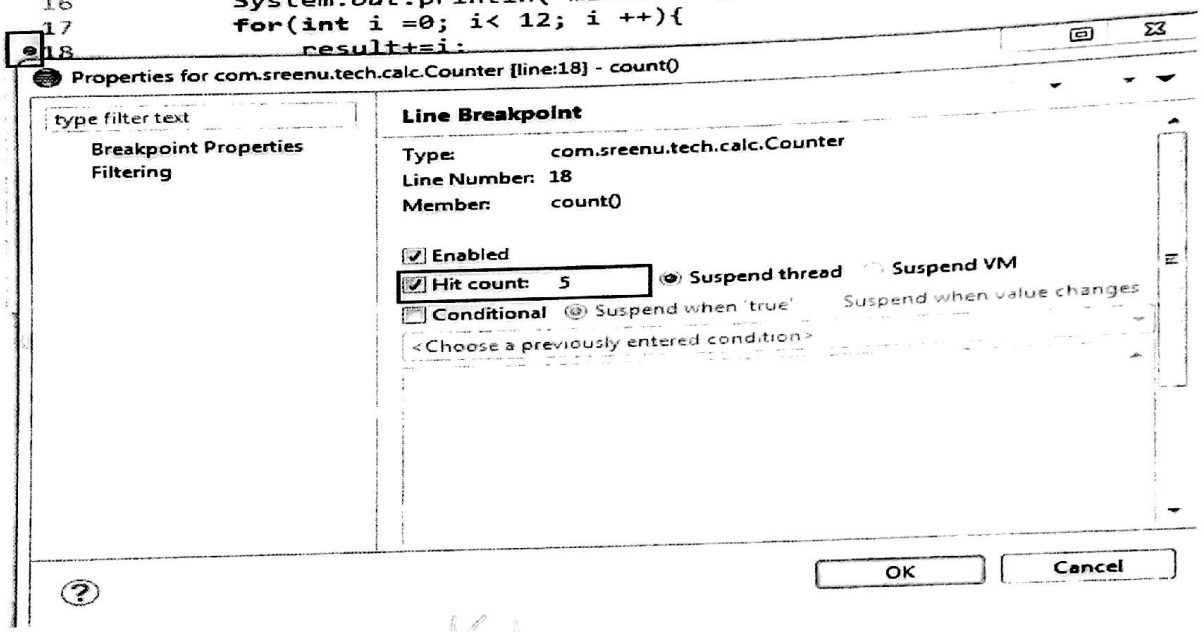


4.7. Hit Count

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.

CalculatorClient.java Calculator3.java Counter.java MyException.java

```
7
8       private int result=0;
9       int count =0;
10
11      private int getResult(){
12          return result;
13      }
14
15      public void count(){
16          System.out.println("method begin...");
17          for(int i =0; i< 12; i ++){
18              result+=i;
```



4.8. Remote debugging

Eclipse allows you to debug applications which runs on another Java virtual machine or even on another machine.

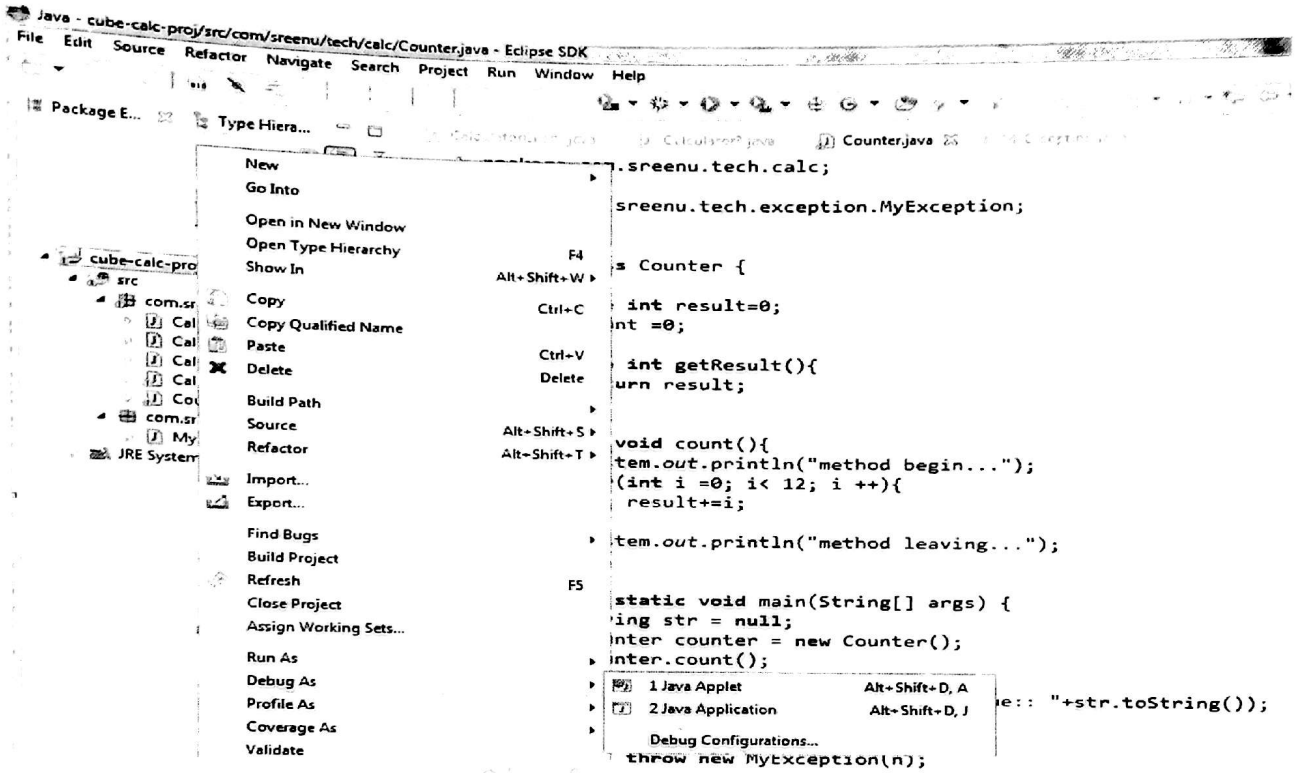
To enable remote debugging you need to start your Java application with certain flags, as demonstrated in the following code example.

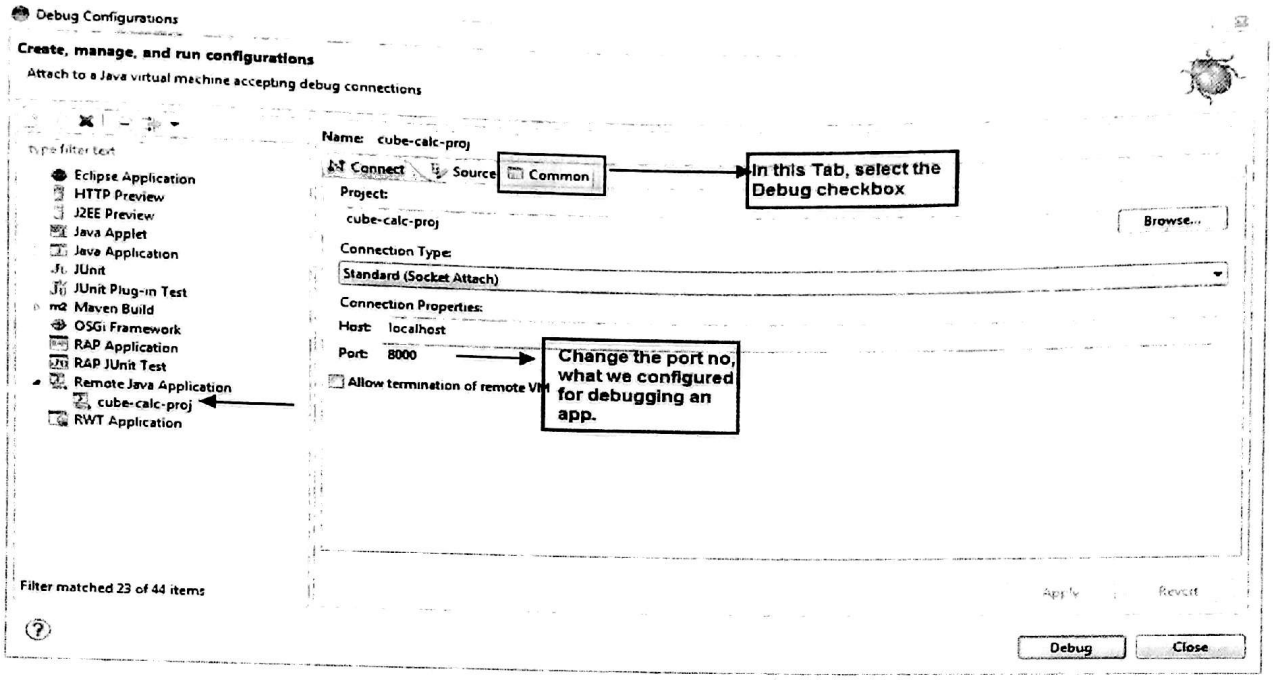
```
java -Xdebug -Xnoagent \
-Djava.compiler=NONE \
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=5005
```

In you Eclipse IDE you can enter the hostname and port to connect for debugging via the Run

→ Debug Configuration... menu.

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 as depicted in the following screenshot.





Note

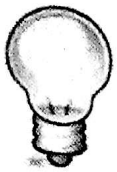
Remote debugging requires that you have the source code of the application which is debugged available in your Eclipse IDE.

4.9. Drop to frame

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.