Debugging

Problemstellung

Für Fehlersuche notwendig: Nachvollziehen, was Code genau macht; Fehler können beim Codelesen übersehen werden und es dauert sehr lange

Lang bekannte Möglichkeit: **Print-Debugging**
Mit `System.out.println();`-Ausgaben im Programmlauf Positionen und Variablenwerte ausgeben

Viele Nachteile:

- Programm wird geändert, obwohl sich an der Ausführlogik nichts ändern sollte
- Programm wird dadurch unübersichtlicher
- Ausgabe unübersichtlich, wenn verschiedene Stellen gleichzeitig etwas ausgeben
- Für zusätzliche Informationen muss jedes Mal das Programm komplett neu gestartet werden

sehr aufwändig falls Benutzereingaben notwendig

Debugger

Debugger ermöglichen eine effiziente Programmuntersuchung ohne Codeänderung während des Programmablaufs:

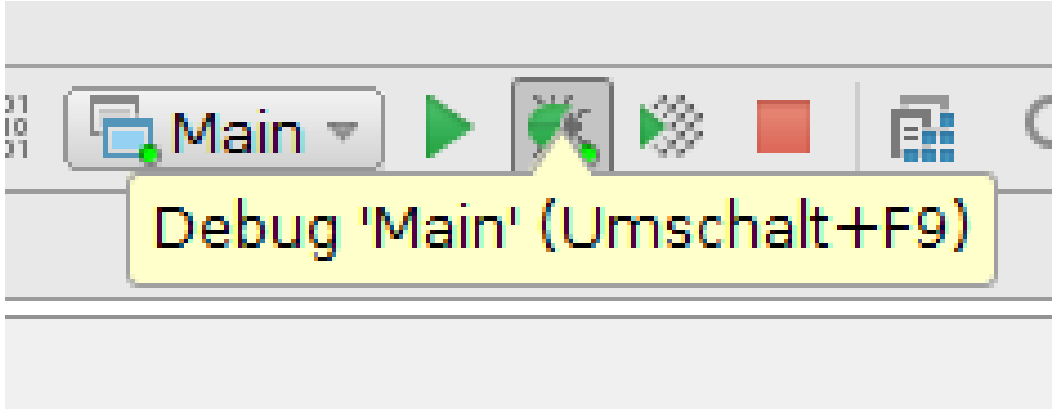
- Programmablauf kann pausiert und fortgesetzt werden
- Es können Unterbrechungspunkte (engl. **Breakpoints**) zum Pausieren an interessanten Stellen des Programms gesetzt werden z.B. in einem `if`-Zweig
- Aktuelle Wert von Variablen können beobachtet werden
- Werte im Programm können für Tests geändert werden
- Exceptions (auch gefangene) können beobachtet werden

Debugger für verschiedene Programmiersprachen arbeiten ähnlich, wir behandeln hier den Debugger von IntelliJ für Java

Debugger starten

Programm im Debugger starten funktioniert ähnlich, wie normal Programm starten

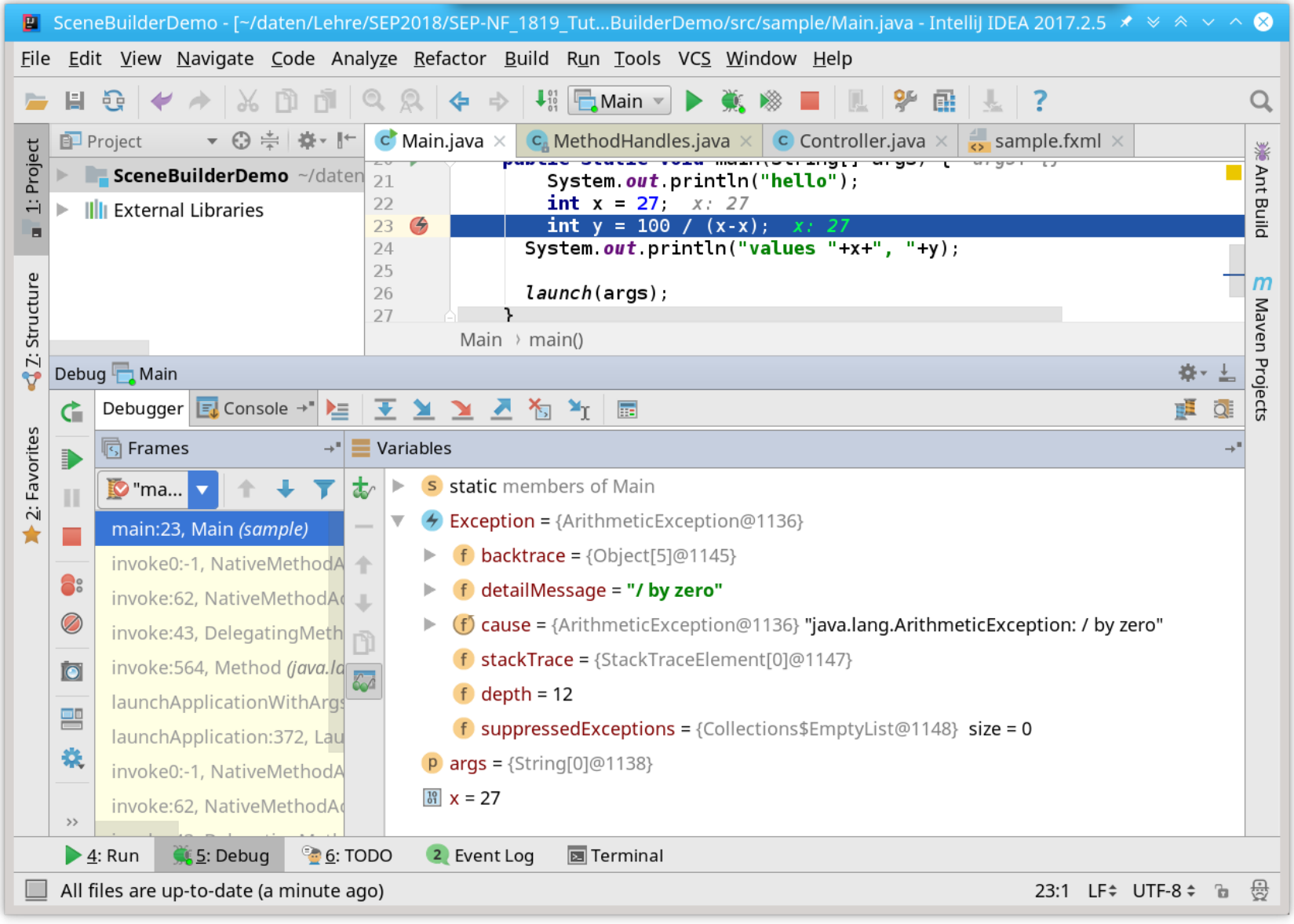
auf den Käfer drücken



Auswirkungen:

- Programmlauf hält an Breakpoints an
- Es steht das Debugmenü zur Verfügung (unter dem Menüpunkt Run): Programm anhalten, schrittweise fortsetzen, ...

Debugger


SceneBuilderDemo - [~/daten/Lehre/SEP2018/SEP-NF_1819_Tut...BuilderDemo/src/sample/Main.java - IntelliJ IDEA 2017.2.5
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Main
Project
SceneBuilderDemo ~/daten
External Libraries
Main.java
MethodHandles.java
Controller.java
sample.fxml
21 System.out.println("hello");
22 int x = 27; x: 27
23 int y = 100 / (x-x); x: 27
24 System.out.println("values "+x+", "+y);
25
26 launch(args);
27 }
Main › main()
Debug Main
Debugger
Console
Frames
Variables
"ma...
main:23, Main (sample)
invoke0:-1, NativeMethodA
invoke:62, NativeMethodAc
invoke:43, DelegatingMeth
invoke:564, Method (java.la
launchApplicationWithArgs
launchApplication:372, Lau
invoke0:-1, NativeMethodA
invoke:62, NativeMethodAc
static members of Main
Exception = {ArithmeticException@1136}
backtrace = {Object[5]@1145}
detailMessage = "/ by zero"
cause = {ArithmeticException@1136} "java.lang.ArithmeticException: / by zero"
stackTrace = {StackTraceElement[0]@1147}
depth = 12
suppressedExceptions = {Collections$EmptyList@1148} size = 0
args = {String[0]@1138}
x = 27
1: Project
7: Structure
2: Favorites
Ant Build
Maven Projects
4: Run
5: Debug
6: TODO
Event Log
Terminal
All files are up-to-date (a minute ago)
23:1 LF UTF-8

Debugmenü

Aktionen im Debugmenü beinhalten unter anderem

Step Over: Nächstes Statement in gleicher Datei, Methodenaufrufe in einem Schritt ausführen

Step Into: Bei Methodenaufrufen in den Rumpf der aufgerufenen Methode gehen

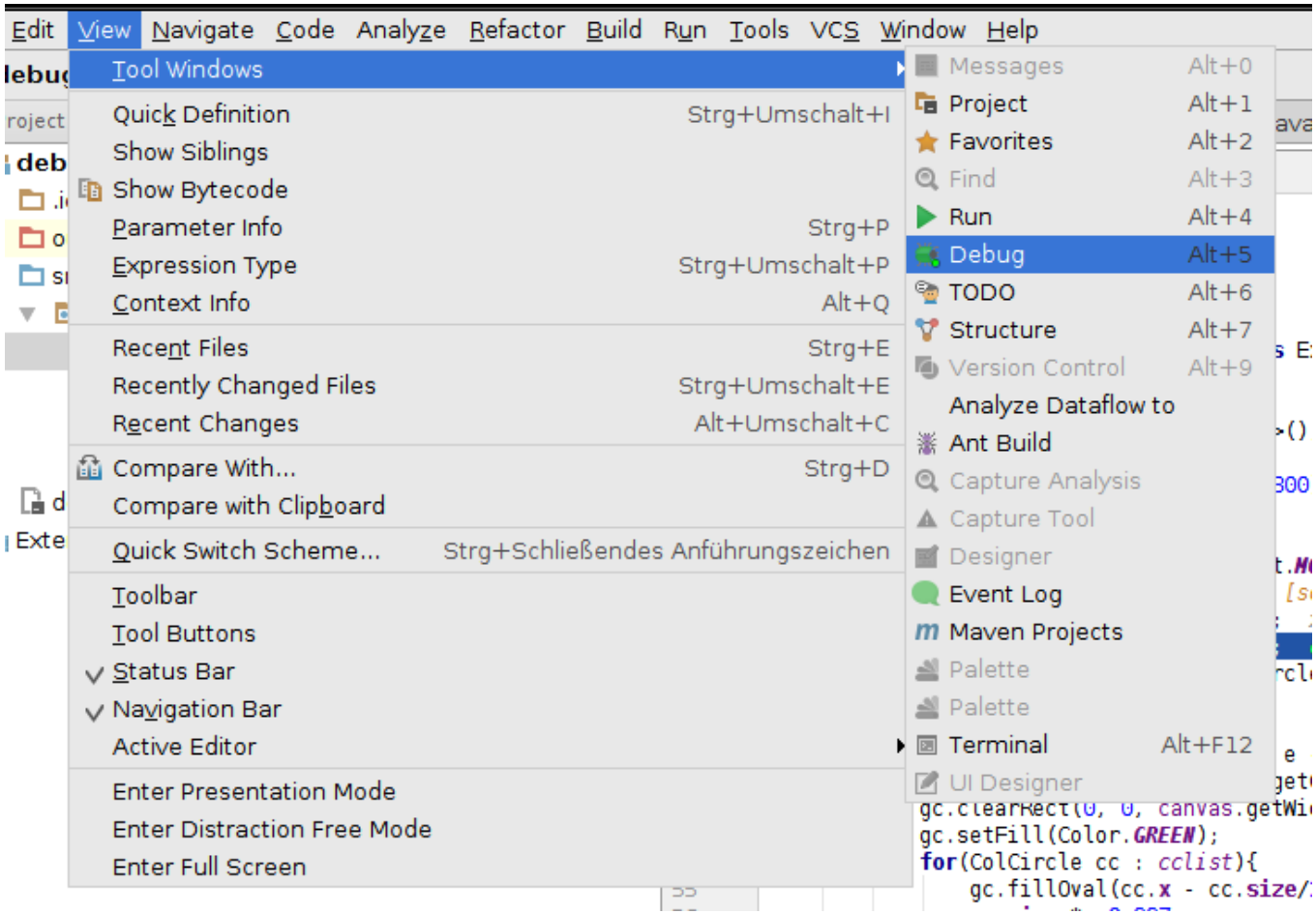
Step Out: so lange weitergehen, bis die Abarbeitung des aktuellen Methodenrumpfes beendet ist

Resume Program: Programm fortsetzen, bis zum nächsten Breakpoint, der ausgelöst wird

Variablen beobachten und setzen

Wenn der Programmablauf pausiert (z.B. weil ein Breakpoint erreicht wurde), kann man im Debug-Fenster Variablen inspizieren und verändern.

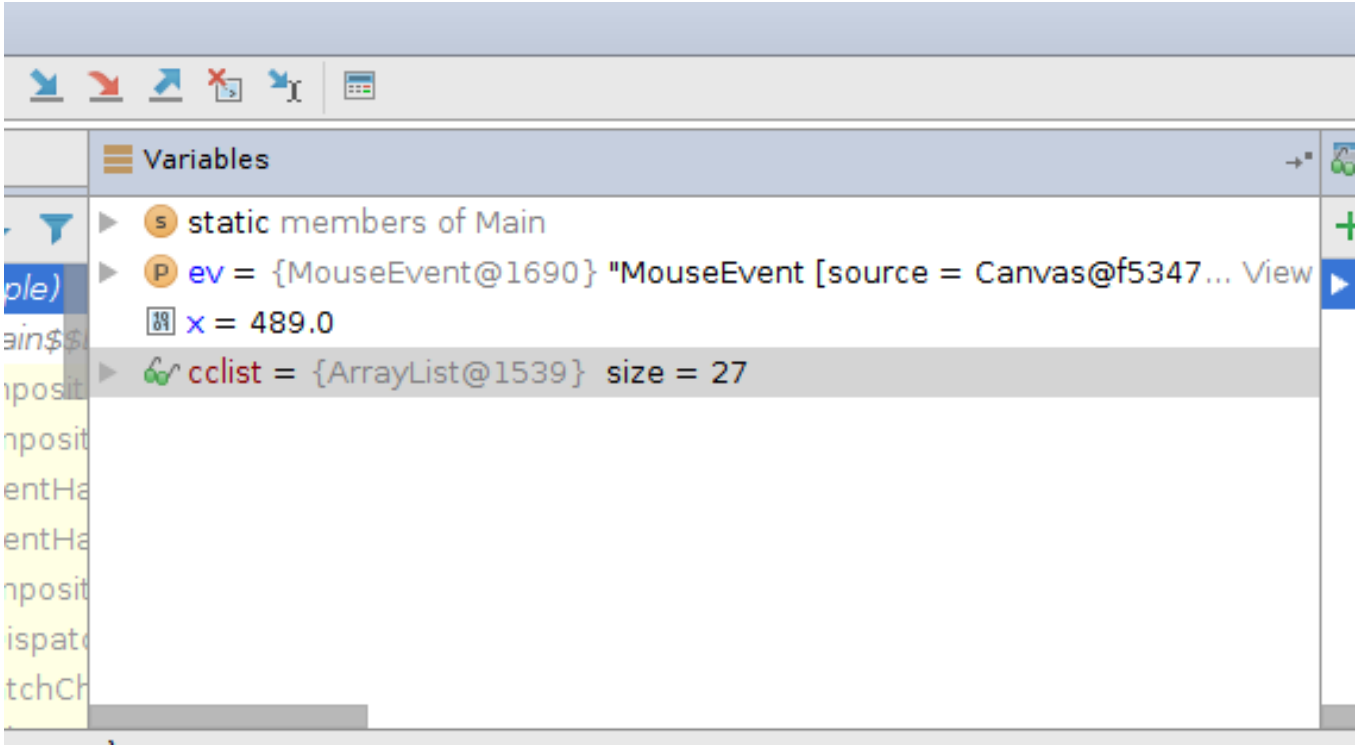
Das Debugfenster erreicht man über das Ansichts-Menü:



Variablen beobachten und setzen

Wenn der Programmablauf pausiert (z.B. weil ein Breakpoint erreicht wurde), kann man im Debug-Fenster Variablen inspizieren und verändern.

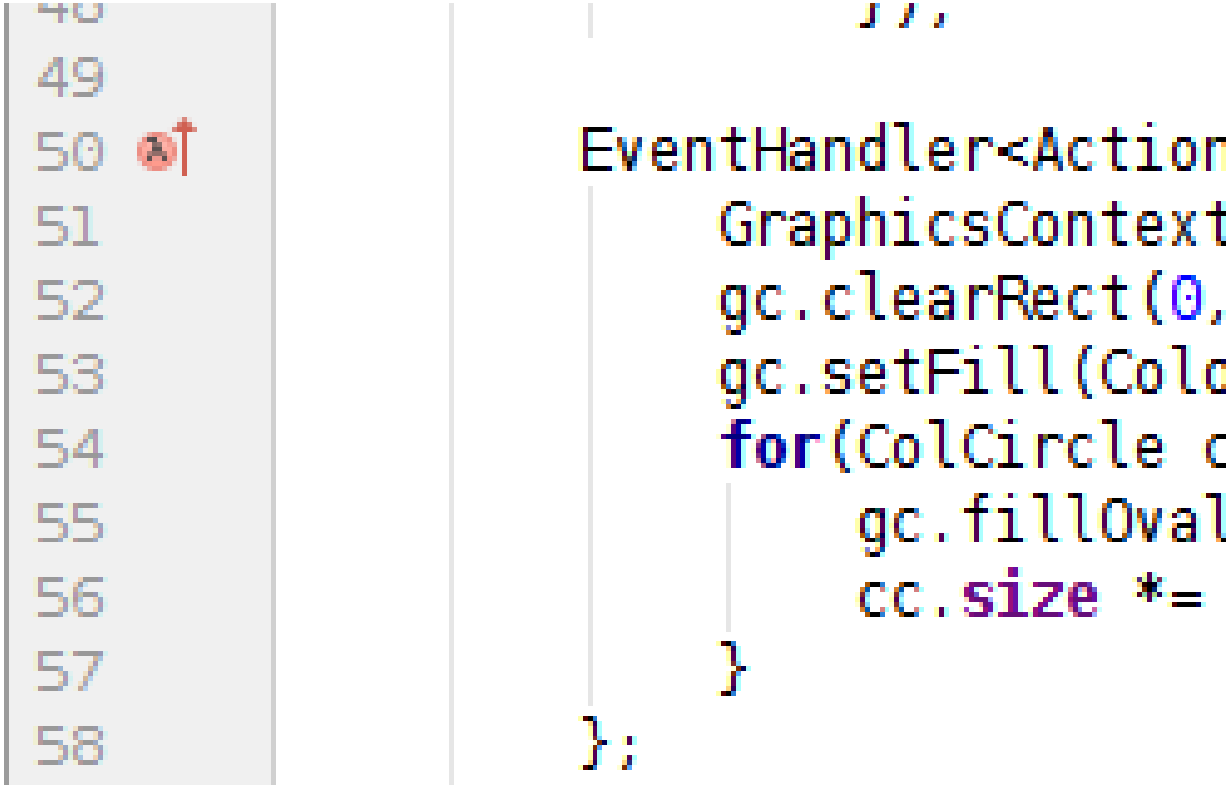
Exemplarischer Inhalt des Debug-Fensters:



Mit Rechtsklick auf die Variablen kann man diese anzeigen und bearbeiten. Beliebige Ausdrücke können ausgewertet werden.

Breakpoints setzen

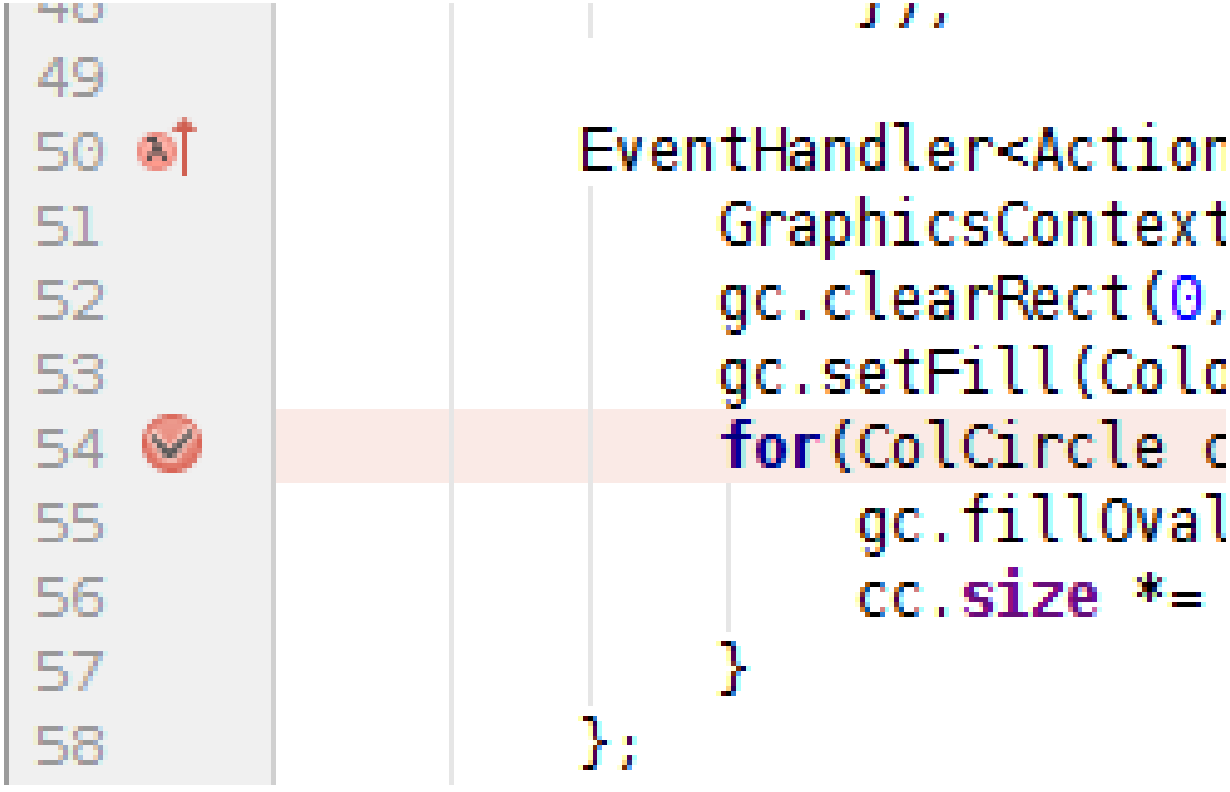
Breakpoints können in IntelliJ durch Mausklick neben der Zeilennummer gesetzt werden



Auswirkung: Programmlauf des Debuggers wird pausiert, sobald ein Statement dieser Zeile ausgeführt werden soll

Breakpoints setzen

Breakpoints können in IntelliJ durch Mausklick neben der Zeilennummer gesetzt werden

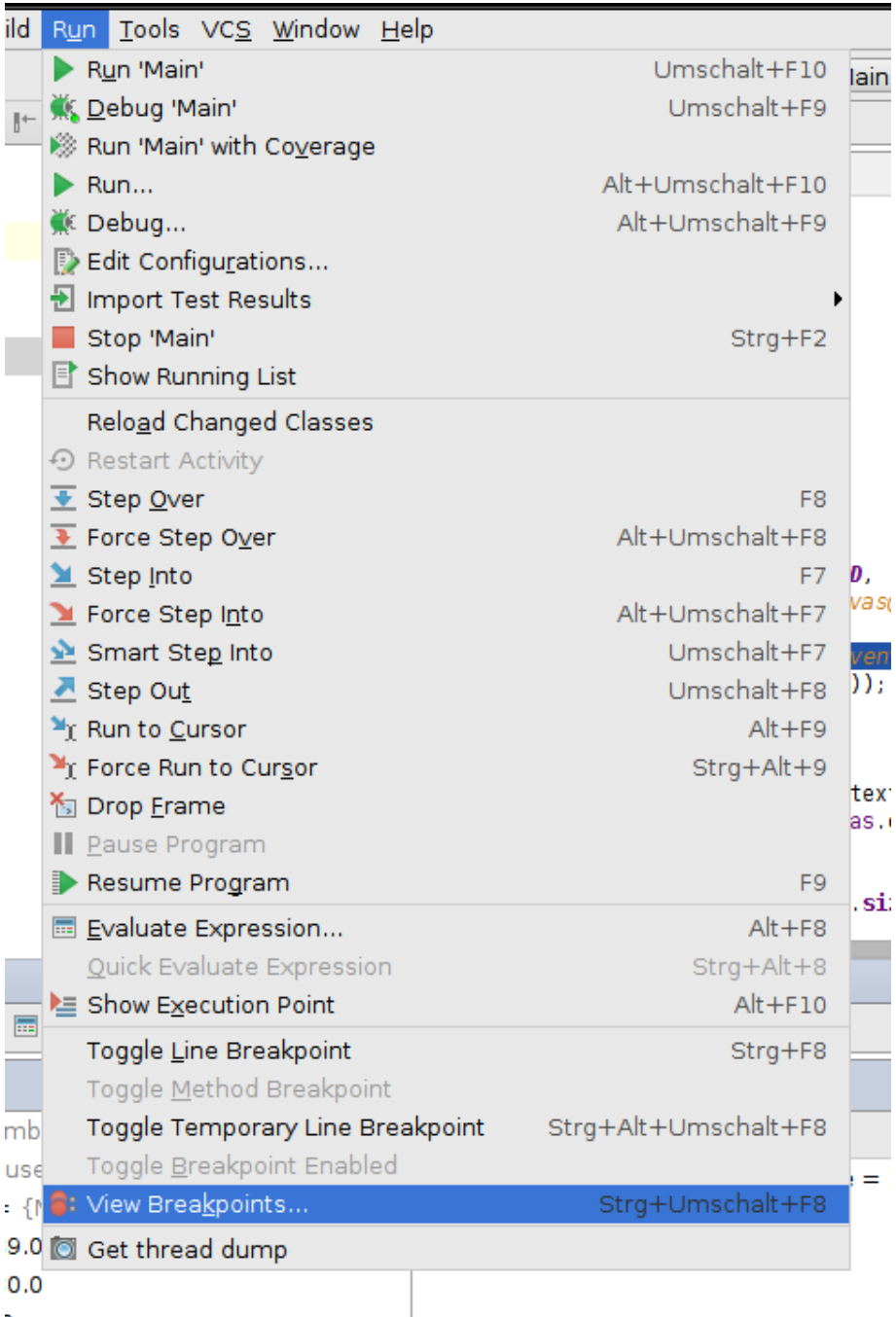


Auswirkung: Programmlauf des Debuggers wird pausiert, sobald ein Statement dieser Zeile ausgeführt werden soll

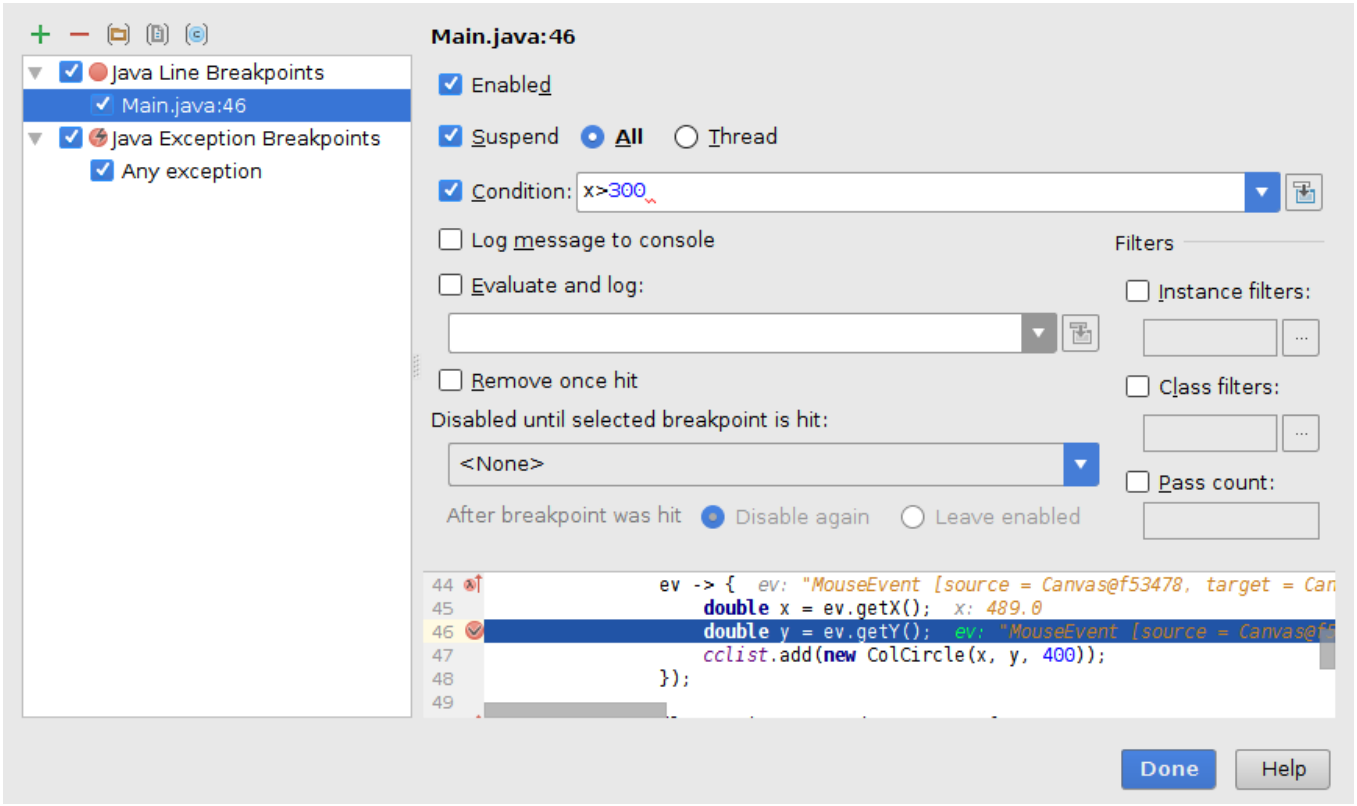
Breakpointliste

Im Debugmenü kann eine Liste der Breakpoints aufgerufen werden.

Beinhaltet gesetzte Breakpoints, sowie Breakpoints für Exceptions (selbst, wenn sie gefangen werden)



Breakpoints konfigurieren

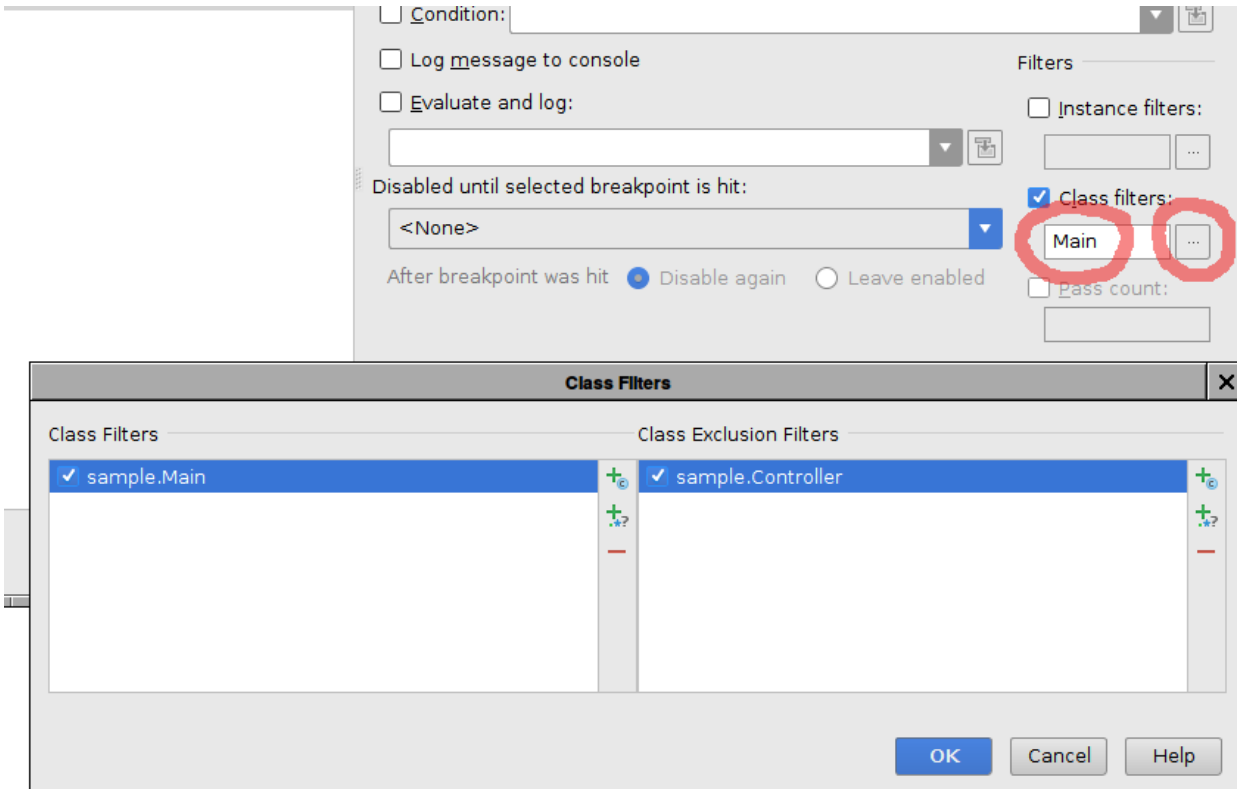


In der Breakpointliste können Bedingungen gesetzt werden, wann ein Breakpoint aktiv ist.

Beispiel im Bild: Ausführung wird nur dann angehalten, wenn bei Ausführung der Stelle $x > 300$ gilt.

Breakpointliste – Class filters

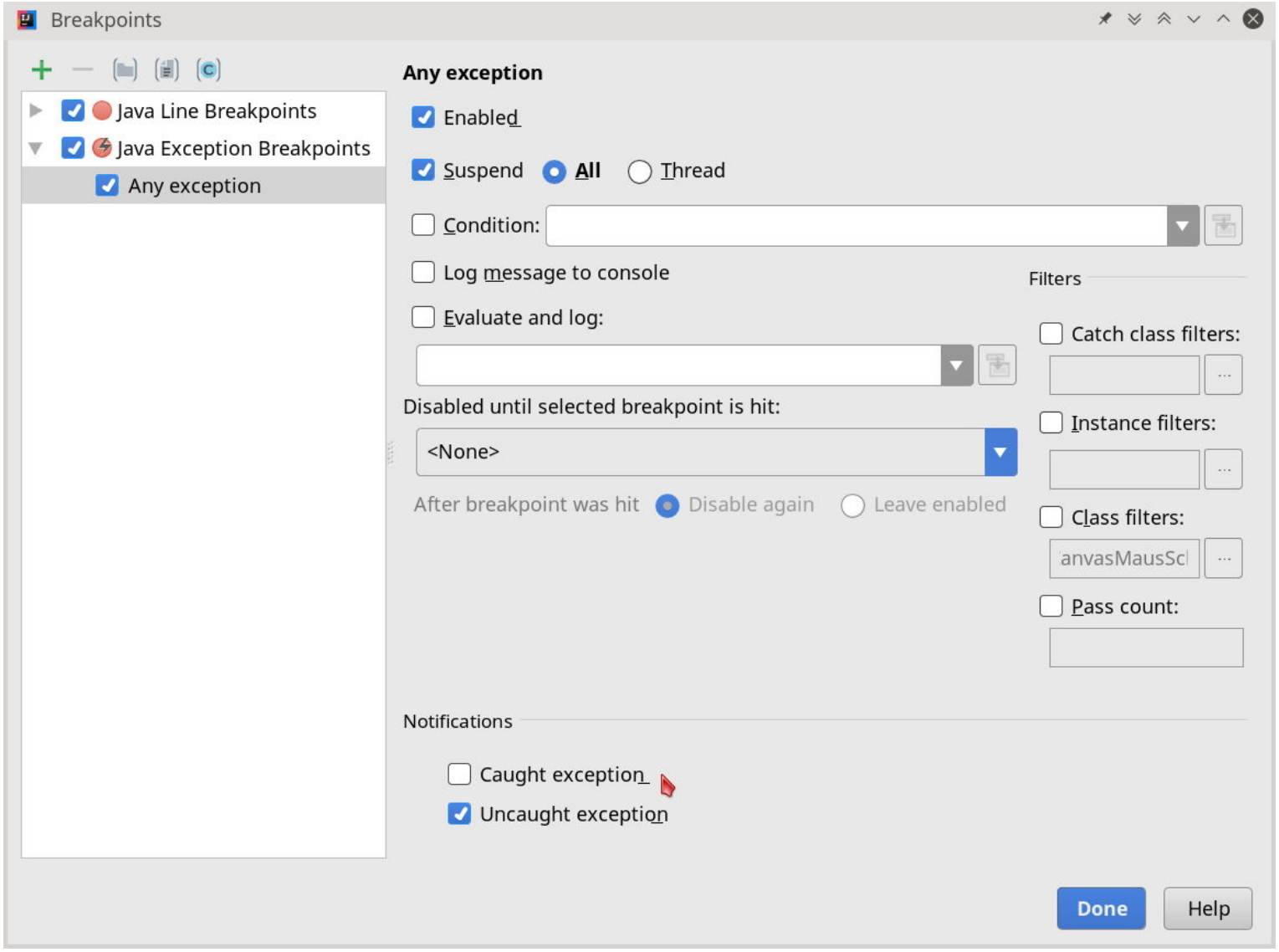
Unter `Run/View Breakpoints` kann man auch **Exception Breakpoints** setzen. Hier ist es oft sinnvoll, über **Class filters** die gewünschten Klassen anzugeben:



Z.B. löst JavaFX beim Start sinnloserweise viele interne Exceptions aus, welche man mit Klassenfiltern unterdrücken kann.

Exception Breakpoints – Un-/Caught Exceptions only

Eine weitere Möglichkeit ist es, nur bei ungefangenen Ausnahmen anzuhalten:



Zusammenfassung

Kennt man den Wert von Variablen versteht man genauer, warum ein Programm ein gewisses Verhalten zeigt

Debugger zu verwenden hat viele Vorteile gegenüber einer reinen Textausgabe:

- Es muss kein Quellcode geändert werden
  starker Vorteil bei Zusammenarbeit in Gruppe
- Programm kann übersichtlicher nachvollzogen werden
- Man während des Programmablaufs entscheiden, welche Werte interessant sind
- Man kann Werte auch testweise ändern