



# **Debugging** - Basic techniques and tools

---

**Steven Costiou**

[steven.costiou@inria.fr](mailto:steven.costiou@inria.fr)

RMoD / Inria Lille - Nord Europe

September 2022

# Summary

---

## Methodology

1. Reporting bugs
2. Understanding and fixing bugs

## Basic tools: traces, debuggers, breakpoints

3. Traces
4. Debuggers: basic usage
5. An overview of standard breakpoints

# Reporting Bugs

1

2

3

4

5

# Reporting bugs

---

- ◆ **It is important to report bugs to developers!**
- ◆ **These reports must be as accurate as possible in order to properly reproduce bugs**
- ◆ Why reproducing bugs?
  - ▶ To understand and fix them
  - ▶ To check if they are fixed!



# How to report bugs?

---

## ◆ **Write Bug Reports:**

- ▶ PR (Problem Report), PTR (Problem Technical Report), etc.

## ◆ **How to write a bug report?**

- ▶ « State all relevant facts » [Zeller, Why Programs Fail (2009)]
- ▶ In practice:
  - ▶ it is hard to choose which information to include
  - ▶ It is hard to make that information concise

# What to report in **bug** reports? [Zeller, 09]

---

## ◆ **Technical information**

- ▶ The product (software, library, module...)
- ▶ The product release number, tag or version
- ▶ The operating system
- ▶ The hardware (or system's resources)
  - ▶ Processor, RAM, etc.
- ▶ And everything specific to the run-time infrastructure

## ◆ **The problem report**

## ◆ **A severity: is that a blocking problem?**

# What to report in **problem** reports? [Zeller, 09]

---

## ◆ **A summary of the problem**

- ▶ « A one-line summary that captures the essence of the problem »

## ◆ **A list of steps to reproduce the bug**

- ▶ Describe a minimal set of steps to reproduce the bug

## ◆ **A description of the symptoms**

- ▶ What is the observed behavior and why it is wrong
- ▶ Only neutral facts: avoid humor, sarcasm, or any digression

## ◆ **A description of the expected behavior**

- ▶ What should have happened if the program behaved as expected?

## ◆ **Technical data and artefacts produced by the buggy program**

- ▶ Execution log, core dumps, binaries, etc.

# What's a severity? [Zeller, 09]

---

## ❖ **Describes the impact of the problem on the software development/release process or for users in production**

- ▶ «Blocker or Showstopper»: blocks development, testing, deployment or usage in production
- ▶ «Critical»: program crashes, data loss, memory leaks, or anything that prevents usage of the software
- ▶ «Major»: loss of important functionality
- ▶ «Normal»: standard level of problem in the development process
- ▶ «Minor»: problem for which there is an obvious/easy fix or for which there are easy-to-use workarounds
- ▶ «Trivial or cosmetic»: no real impact (misspelled words, wrong logo...)
- ▶ Etc.

# Reporting bugs in practice

---

## ◆ **We use bug/issue trackers**

- ▶ Tools to report and track bugs
  - ▶ Assign bugs to developers
  - ▶ Follow its evolution
  - ▶ Make pull request with bug fixes
  - ▶ Make code reviews of the bug fixes
- ▶ Mantis, Github, Clearquest, etc.

# Understanding and fixing bugs

1

2

3

4

5

# Understanding and fixing bugs?

---

## ◆ **This is not a recipe**

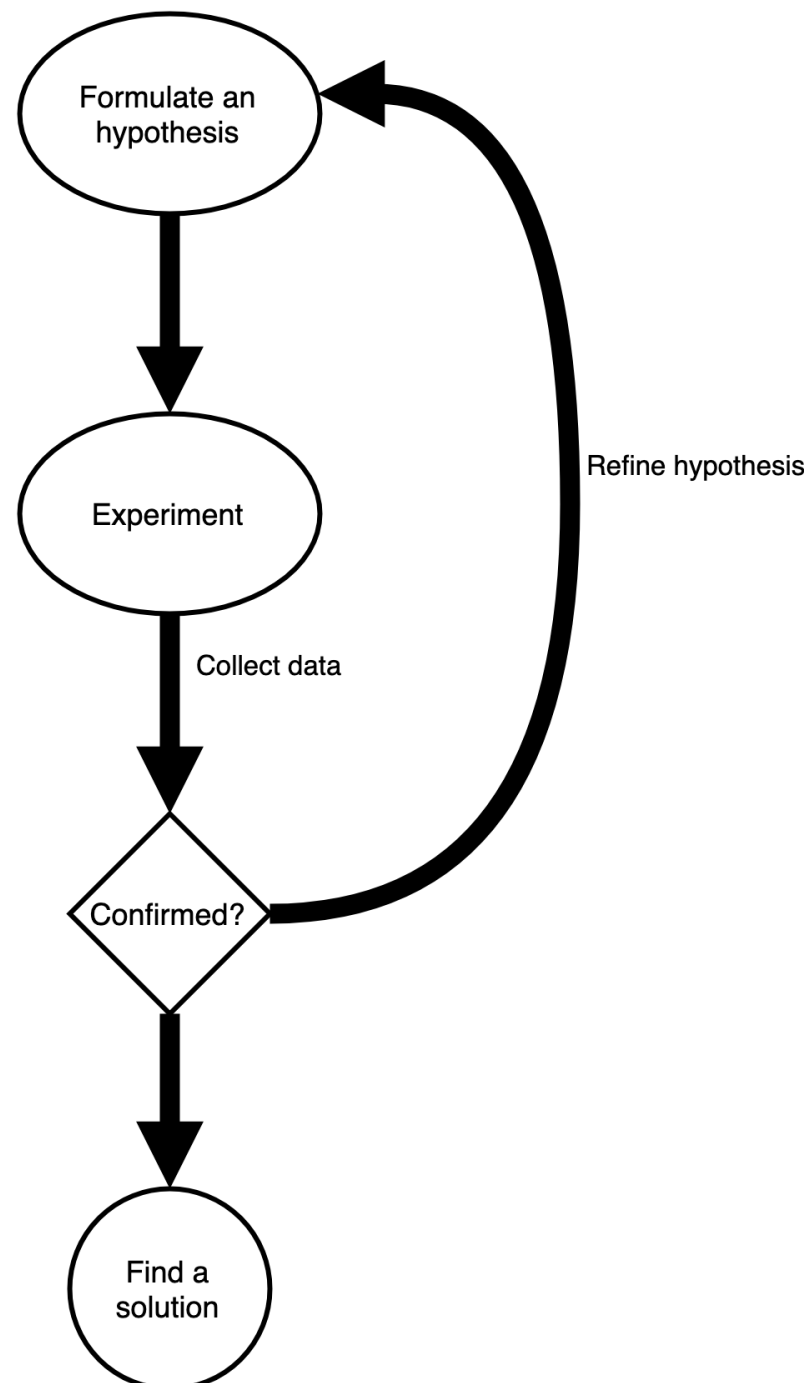
- ▶ The following are general rules and advices
- ▶ This is introductory to more reading
- ▶ Those rules are complementary to practical experience

## ◆ **Good references for detailed methodology**

1. **Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems**, David J. Agans, 2002
2. **Why Programs Fail**, Andreas Zeller, 2009
3. **Effective Debugging**, Diomidis Spinellis, 2016
4. **The Science of Debugging**, Telles and Hsied, 2001

# Overview: « simplified » simplified scientific method

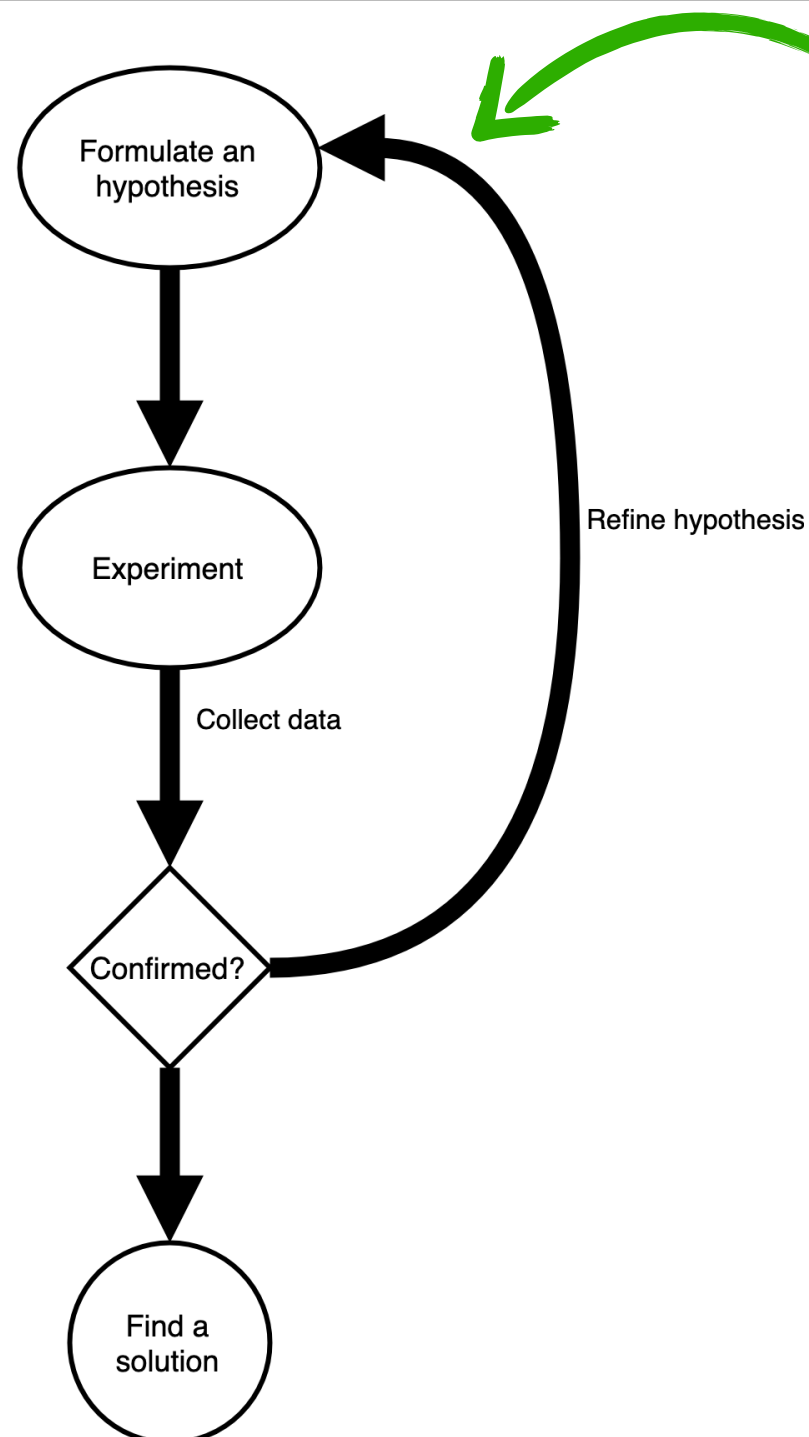
---





# Overview: « simplified » simplified scientific method

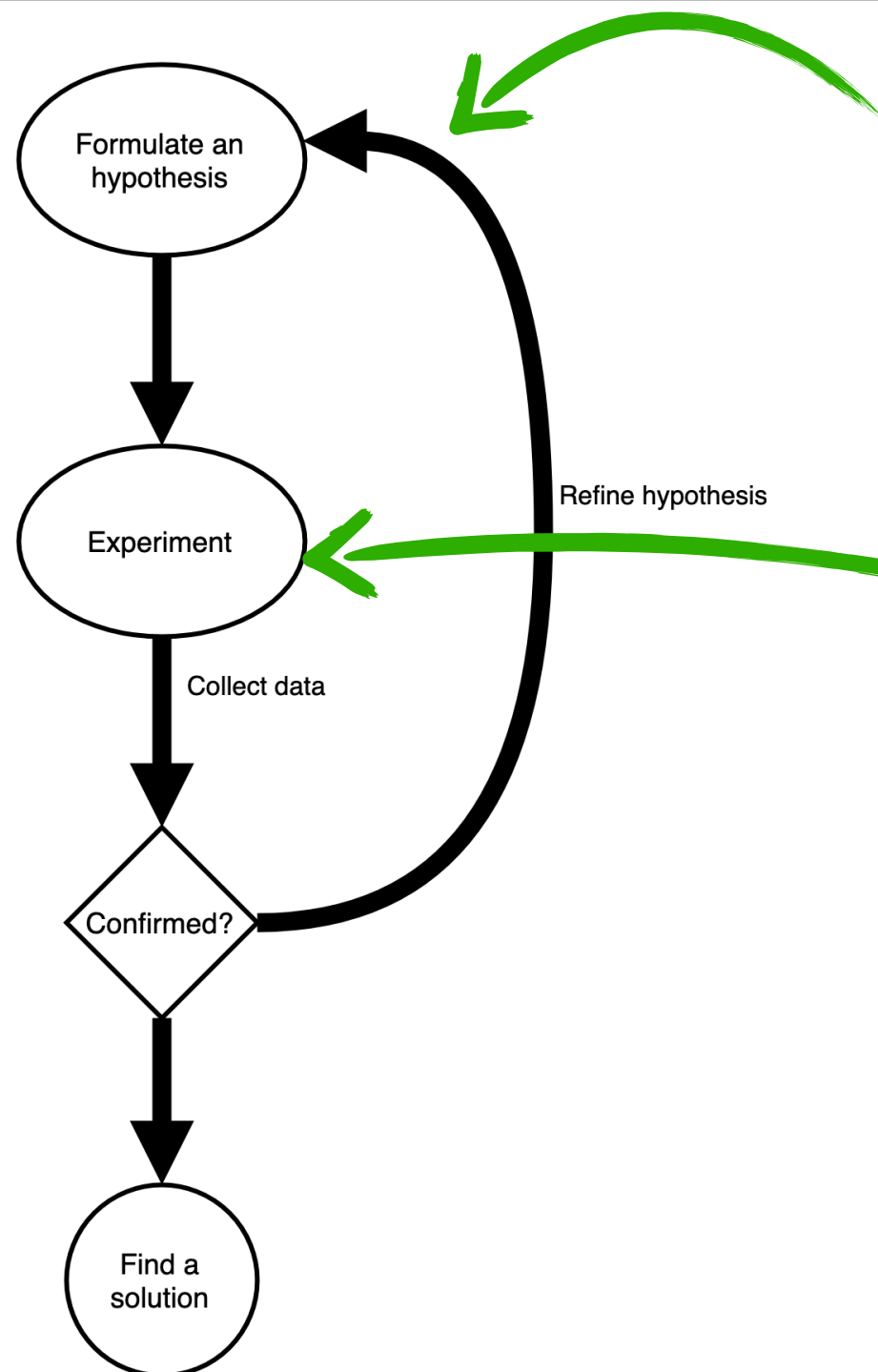
---



Base your hypothesis on:

- what you observe
- what you know about the system
- a preliminary analysis of the system

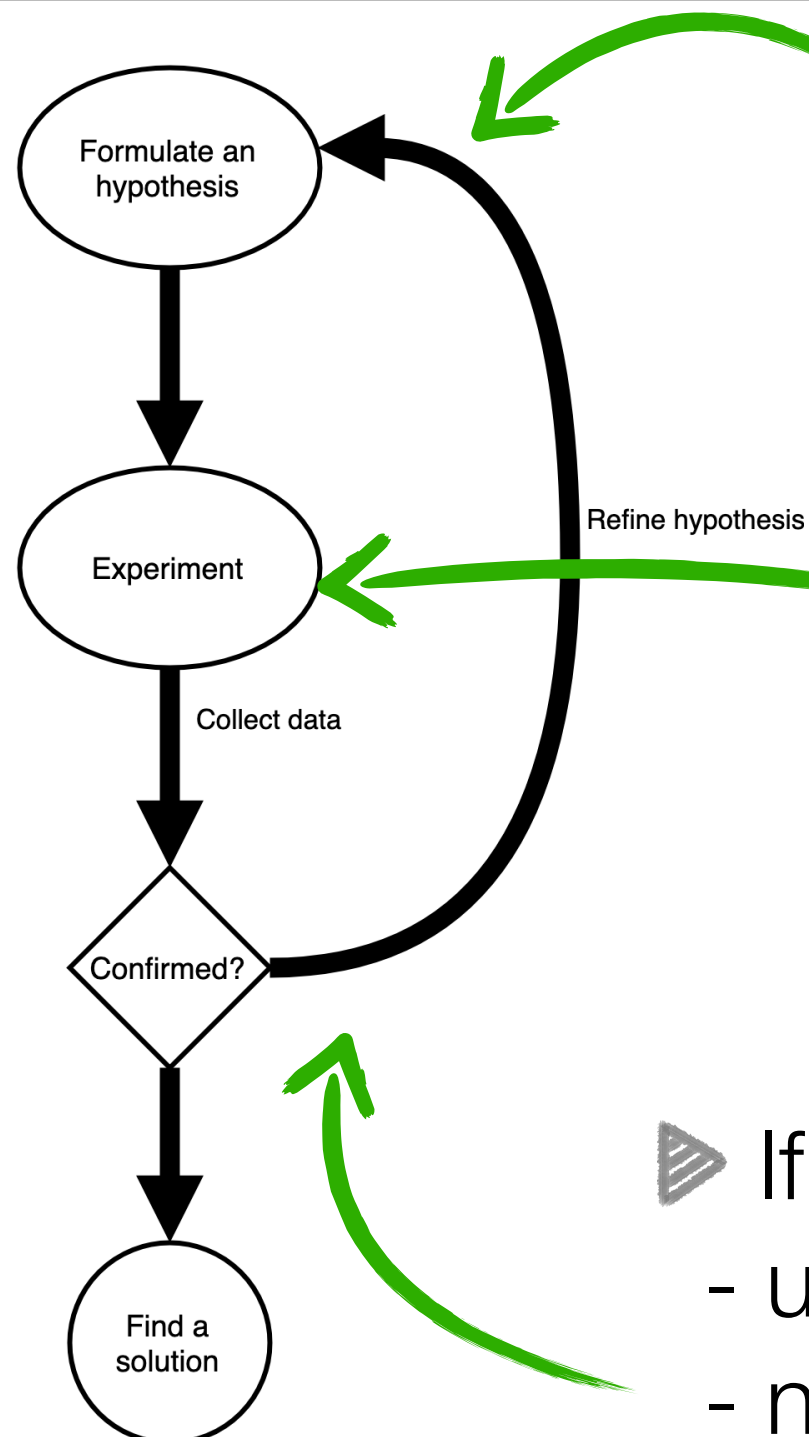
# Overview: « simplified » simplified scientific method



- Base your hypothesis on:
- what you observe
  - what you know about the system
  - a preliminary analysis of the system

- Experiment and collect data:
- test your hypothesis correct
  - understand the system better

# Overview: « simplified » simplified scientific method



- Base your hypothesis on:
  - what you observe
  - what you know about the system
  - a preliminary analysis of the system

- Experiment and collect data:
  - test your hypothesis correct
  - understand the system better

- If your hypothesis is wrong or incomplete:
  - use your new understanding to refine it
  - narrow down your experimental setup to obtain more precise data

# Overview: different steps

---

## ◆ **Observe the bug**

- ▶ Make the program fail: reproduce the bug
- ▶ Simplify the problem: reduce it to the smallest set of conditions

## ◆ **Narrow down the search**

- ▶ Observe the smallest entity / reduce the search space
- ▶ Use assertions to validate program state at key points of the program's execution

## ◆ **Fix the bug**

- ▶ Write tests and execute tests suites to ensure non regression
- ▶ Write assertions in the code to ensure the bug does not reappear

# Observe the bug (1)

---

## ◆ **The objective is to control the program to force bug reproduction**

- ▶ Reproduce the faulty environment
  - ▶ The bug may only happen in a specific context
  - ▶ The bug may happen in different environments (e.g., production & dev)

# Observe the bug (1)

---

## ◆ The objective is to control the program to force bug reproduction

- ▶ Reproduce the faulty environment
  - ▶ The bug may only happen in a specific context
  - ▶ The bug may happen in different environments (e.g., production & dev)
- ▶ Reproduce the faulty execution
  - ▶ Control inputs: force the program to use specific values
  - ▶ Control behavior: force the program to use a specific mode or configuration
  - ▶ **This should not be random:** you must understand the program and formulate reasonable hypotheses about the possible problem's origin

# Observe the bug (2)

---

## ◆ **Help yourself and others to reproduce the bug**

- ▶ Write a step-by-step procedure
  - ▶ Applying the procedure guarantees bug reproduction
  - ▶ Usually in bug reports
  - ▶ Sometimes it is not possible to do otherwise

# Observe the bug (2)

---

## ◆ **Help yourself and others to reproduce the bug**

### ▶ Write a step-by-step procedure

- ▶ Applying the procedure guarantees bug reproduction
- ▶ Usually in bug reports
- ▶ Sometimes it is not possible to do otherwise

### ▶ Write a unit test

- ▶ Implement the minimal conditions for bug reproduction
- ▶ Very powerful tool when it can be done
- ▶ Executing the test reproduces the bug: **you can observe it on demand!**
- ▶ This test should be included in the program's test suite and executed each time a change is done in the program (continuous integration, release...)



# Narrow down the search (1)

---

## ◆ **Reduce the search space: you're looking for the source of the bug**

### ▶ Divide and conquer

- ▶ Eliminate parts of the code not involved in the bug
- ▶ Insert trace and instrumentation to infirm or confirm an hypothesis
- ▶ Proceed by successive approximations

# Narrow down the search (1)

---

## ◆ **Reduce the search space: you're looking for the source of the bug**

### ▶ Divide and conquer

- ▶ Eliminate parts of the code not involved in the bug
- ▶ Insert trace and instrumentation to infirm or confirm an hypothesis
- ▶ Proceed by successive approximations

### ▶ Use assertions

- ▶ To validate key points of your program
- ▶ To compare what is expected with what you observe

# Narrow down the search (1)

---

## ◆ **Reduce the search space: you're looking for the source of the bug**

### ▶ Divide and conquer

- ▶ Eliminate parts of the code not involved in the bug
- ▶ Insert trace and instrumentation to infirm or confirm an hypothesis
- ▶ Proceed by successive approximations

### ▶ Use assertions

- ▶ To validate key points of your program
- ▶ To compare what is expected with what you observe

### ▶ Target fine-grained entities

- ▶ Use conditions to define your debugging space
  - ▶ Target variables and state involved in the bug
  - ▶ In object-oriented programs: debug objects (possibly only one)
- ▶ Change one thing at a time: you need to be able to conclude information

# Narrow down the search (2)

---

## Example of assertion in the Java Virtual Machine (Open JDK)

```
#ifdef ASSERT
void ResourceObj::set_allocation_type(address res, allocation_type type) {
    // Set allocation type in the resource object
    uintptr_t allocation = (uintptr_t)res;
    assert((allocation & allocation_mask) == 0, "address should be aligned to 4 bytes at least: " INTPTR_FORMAT, p2i(res));
    assert(type <= allocation_mask, "incorrect allocation type");
    ResourceObj* resobj = (ResourceObj *)res;
    resobj->_allocation_t[0] = ~(allocation + type);
    if (type != STACK_OR_EMBEDDED) {
        // Called from operator new(), set verification value.
        resobj->_allocation_t[1] = (uintptr_t)&(resobj->_allocation_t[1]) + type;
    }
}
```

# Fix the bug

---

◆ **When you find the bug, and fixed it**

# Fix the bug

---

## ◆ **When you find the bug, and fixed it**

- ▶ Explain the bug!
  - ▶ Answer the bug report
  - ▶ Organize a code review

# Fix the bug

---

## ◆ **When you find the bug, and fixed it**

### ▶ Explain the bug!

- ▶ Answer the bug report
- ▶ Organize a code review

### ▶ Write tests!

- ▶ Ensure that you can detect the bug if it reappears
- ▶ Also counts as an explanation!

# Fix the bug

---

## ◆ **When you find the bug, and fixed it**

### ▶ Explain the bug!

- ▶ Answer the bug report
- ▶ Organize a code review

### ▶ Write tests!

- ▶ Ensure that you can detect the bug if it reappears
- ▶ Also counts as an explanation!

## ◆ **You did not fix it...**

- ▶ Because you cannot? This happens. Look for help.
- ▶ Because it suddenly work? If you did nothing, it is still broken...



# Traces

1

2

3

4

5

# Execution traces

---

◆ **A trace is a recorded information about a program at a given time of its execution**

- ▶ Also called « logging »
- ▶ Extract any kind of information:
  - ▶ Program state
  - ▶ Control flow (*i.e.*, « we're here in the code »)
  - ▶ Input/outputs
  - ▶ Etc.
- ▶ Basic traces are printed on the standard output stream (*e.g.*, in the terminal) («printf technique»)

# Execution traces - a C example

---

```
while (counter >= 0) {
```

```
    counter --;
```

```
}
```

# Execution traces - a C example

---

```
while (counter >= 0) {  
    printf("Counter before: %d", counter);  
  
    counter --;  
  
    printf("Counter after: %d", counter);  
}
```

# Execution traces

---

## ◆ Downsides

- ▶ Pollutes code, even in production
- ▶ Pollutes the execution when activated: traces are always visible, by all users

## ◆ Possible solutions

- ▶ Configuration files to activate traces (*e.g.*, *DEBUG* mode)
- ▶ Pre-processor macros to eliminate trace instructions in the production binary

## Execution traces - Debug mode with a macro in C

---

```
#define DEBUG 1
```

```
while (counter >= 0) {
```

```
    printf("Counter before: %d", counter);
```

```
    counter --;
```

```
    printf("Counter after: %d", counter);
```

```
}
```

# Execution traces - Debug mode with a macro in C

---

```
#define DEBUG 1

while (counter >= 0) {
    #ifdef DEBUG
    printf("Counter before: %d", counter);
    #endif

    counter --;

    #ifdef DEBUG
    printf("Counter after: %d", counter);
    #endif
}
```

## Execution traces - Debug mode with a macro in C

```
#define DEBUG 1
```



```
gcc -D DEBUG my_prog.c
```

```
while (counter >=
```

```
#ifdef DEBUG
```

```
printf("Counter before: %d", counter);
```

```
#endif
```

```
counter --;
```

```
#ifdef DEBUG
```

```
printf("Counter after: %d", counter);
```

```
#endif
```

```
}
```

Compile-time generation  
of the macro



# Execution traces - a C example

---

```
Counter after: -9306
Counter before: -9306
Counter after: -9307
Counter before: -9307
Counter after: -9308
Counter before: -9308
Counter after: -9309
Counter before: -9309
Counter after: -9310
Counter before: -9310
Counter after: -9311
Counter before: -9311
Counter after: -9312
Counter before: -9312
Counter after: -9313
Counter before: -9313
Counter after: -9314
Counter before: -9314
Counter after: -9315
Counter before: -9315
Counter after: -9316
Counter before: -9316
Counter after: -9317
Counter before: -9317
Counter after: -9318
Counter before: -9318
Counter after: -9319
Counter before: -9319
Counter after: -9320
```

# Log files and core dumps

---

## ◆ **Log files**

- ▶ Traces are written in a file on disk rather than on the output stream
- ▶ Log files can be of large size and verbose: we require tools to efficiently exploit them

## ◆ **Core dumps**

- ▶ Copy of the state of a program at the moment of a bug
- ▶ Can be reloaded into a debugger to visualise that state

# Execution traces: pros and cons

---

## ◆ **Pros**

- ▶ Quick, cheap and intuitive to implement to start an investigation
- ▶ Allows developers to obtain a simple visualisation of their program's state and behaviour — in time and space
- ▶ Lots of trivial bugs can be observed and fixed very quickly using simple traces
- ▶ We can use traces to build advanced tools (e.g., execution analysis)

# Execution traces: pros and cons

---

## ◆ Cons

- ▶ Without tool support, we have to insert them manually into the source code
- ▶ Low flexibility to explore the observation scope of the program when looking for bugs
- ▶ High granularity and noise
  - ▶ a lot of traces can be generated
  - ▶ reducing them to the minimum information needed to observe/understand a bug is not trivial
- ▶ Intrusive: they pollute the program's source code

# Execution traces: conclusion

---

- ◆ Try them as a first, basic tool
- ◆ **If the bug is not quickly found:**
  - ▶ abandon the technique and don't lose time
  - ▶ use more advanced techniques...
- ◆ **On some systems, there are no other means for debugging...**
  - ▶ Embedded systems without communication capabilities
  - ▶ Systems without debugging primitives
  - ▶ Kernel code (e.g., low-level drivers)

# Debuggers

## basic usage

1

2

3

4

5

# Debuggers

---

## ◆ **What is a debugger?**

- ▶ It is a tool (an application) which is used to debug other applications

## ◆ **What does a (basic) debugger do?**

- ▶ Set breakpoints
- ▶ Step-by-step execution
- ▶ Program state observation
- ▶ Code execution

# Different kind of debuggers

---

## ◆ **Source-level debuggers**

- ▶ The executed machine-level code is mapped to the source code from which it was generated
- ▶ Gives a human-readable view of the executed code
- ▶ Gives the illusion to execute the source code

## ◆ **Machine-level debuggers**

- ▶ No source code, only machine instruction or assembly
- ▶ Developers need to examine registries, memory addresses, and disassemble machine code

## ◆ **Advanced debuggers**

- ▶ Specialised tools (e.g., analysis tools), research prototypes...



# Different kind of integration

---

## ◆ **Standalone debuggers**

- ▶ With or without GUI (e.g., from a terminal)
  - ▶ Sometimes, there are no other means of debugging...
- ▶ Hook up to a program compiled with debugging support

## ◆ **Integrated debuggers**

- ▶ Integrated to IDEs
- ▶ Usually provide helpers and additional tools to facilitate debugging (compilation in debug mode, visualisations, etc.)

# Different kind of interfaces

---

## ◆ **Programming languages usually provide debugging interfaces relying on debugging primitives**

- ▶ Interfaces are provided by the language (e.g., debugging functions) and the runtime (e.g., remote connection and actions)
- ▶ Primitives are provided by run time infrastructures (e.g. virtual machines)

## ◆ **Sometimes debuggers require specific interfaces**

- ▶ From the operating system (if specific)
- ▶ From the hardware... (an example just after!)











# A C debugger: GDB

---

- ◆ Usable in command-line and IDEs
- ◆ Requires a program compiled with **debugging support**
  - ▶ Additional information for use by a debugger
  - ▶ Contains symbols (e.g., functions names) and information that may be lost due to compilation/optimization
- ◆ **Example:**

```
gcc -g my_program.c -o my_program
```

# Run GDB

---

```
gdb my_program  
> (gdb)
```

- ▶ Give your compiled program as argument to the `gdb` command
- ▶ Gdb starts: your program **is not** started
- ▶ Gdb awaits for your commands!

# Set a breakpoint

---

>(gdb) break my\_program.c:25

```
18 void sort(int* array, int array_size)
19 {
20     int i;
21     int minimum;
22
23     for(i = 0; i < array_size - 1; ++i)
24     {
25         minimum = i;
26         int j;
27         for(j = i + 1; j < array_size; ++j)
28             if(array[j] < array[minimum])
29                 minimum = j;
30
31         swap_ints(&array[i], &array[minimum]);
32     }
33 }
```

▶ Will break at line 25 of  
file my\_program.c

# Set a conditional breakpoint

---

```
>(gdb) break my_program.c:25 if i > 2
```

```
18 void sort(int* array, int array_size)
19 {
20     int i;
21     int minimum;
22
23     for(i = 0; i < array_size - 1; ++i)
24     {
25         minimum = i;
26         int j;
27         for(j = i + 1; j < array_size; ++j)
28             if(array[j] < array[minimum])
29                 minimum = j;
30
31         swap_ints(&array[i], &array[minimum]);
32     }
33 }
```

► Will break at line 25 of file `my_program.c` when the local variable `i` is higher than 2




# Run the program

---

> (gdb) run

```
18 void sort(int* array, int array_size)
19 {
20     int i;
21     int minimum;
22
23     for(i = 0; i < array_size - 1; ++i)
24     {
25         minimum = i;
26         int j;
27         for(j = i + 1; j < array_size; ++j)
28             if(array[j] < array[minimum])
29                 minimum = j;
30
31         swap_ints(&array[i], &array[minimum]);
32     }
33 }
```



- ▶ The program starts
- ▶ The program is interrupted when:
  - ▶ The execution reaches our breakpoint
  - ▶ The conditions of the breakpoint are satisfied

# Exploring the program's state (1)

---

- Show the stack frame (where the program is stopped)

> (gdb) frame

```
(gdb) frame
#0  sort (array=0x7fffffffdd50, array_size=5) at anagrams.c:25
25      minimum = i;
```

- Show the current code being executed

> (gdb) list

```
(gdb) list
24      {
25          minimum = i;
26          int j;
27          for(j = i + 1; j < array_size; ++j)
28              if(array[j] < array[minimum])
29                  minimum = j;
30
31          swap_ints(&array[i], &array[minimum]);
32      }
33  }
```

- Inspect variables

> (gdb) print i

```
(gdb) print i
$3 = 3
```

# Exploring the program's state (2)

► Ask for the disassembly  
> (gdb) disassemble

- We see disassembled machine code
- It shows where the program is stopped

```
(gdb) disassemble
Dump of assembler code for function sort:
0x0000555555554827 <+0>:      push    %rbp
0x0000555555554828 <+1>:      mov     %rsp,%rbp
0x000055555555482b <+4>:      sub     $0x20,%rsp
0x000055555555482f <+8>:      mov     %rdi,-0x18(%rbp)
0x0000555555554833 <+12>:     mov     %esi,-0x1c(%rbp)
0x0000555555554836 <+15>:     movl    $0x0,-0xc(%rbp)
0x000055555555483d <+22>:     jmpq    0x5555555548cc <sort+165>
=> 0x0000555555554842 <+27>:     mov     -0xc(%rbp),%eax
0x0000555555554845 <+30>:     mov     %eax,-0x8(%rbp)
0x0000555555554848 <+33>:     mov     -0xc(%rbp),%eax
0x000055555555484b <+36>:     add     $0x1,%eax
0x000055555555484e <+39>:     mov     %eax,-0x4(%rbp)
0x0000555555554851 <+42>:     jmp     0x55555555488d <sort+102>
0x0000555555554853 <+44>:     mov     -0x4(%rbp),%eax
0x0000555555554856 <+47>:     cltq
0x0000555555554858 <+49>:     lea     0x0(,%rax,4),%rdx
0x0000555555554860 <+57>:     mov     -0x18(%rbp),%rax
0x0000555555554864 <+61>:     add     %rdx,%rax
0x0000555555554867 <+64>:     mov     (%rax),%edx
0x0000555555554869 <+66>:     mov     -0x8(%rbp),%eax
```

```
(gdb) info breakpoint
Num   Type      Disp Enb Address                               What
1     breakpoint keep y  <PENDING>    my_program.c:25 if i > 2
2     breakpoint keep y  <PENDING>    my_program.c:25 if i > 1
3     breakpoint keep y  <PENDING>    my_program.c:25 if i < 1
4     breakpoint keep y  0x0000555555554842 in sort at anagrams.c:25
stop only if i > 2
breakpoint already hit 1 time
```

► Ask for debugging info  
> (gdb) info breakpoints

# Step-by-step execution (1)

---

> (gdb) next

```
18 void sort(int* array, int array_size)
19 {
20     int i;
21     int minimum;
22
23     for(i = 0; i < array_size - 1; ++i)
24     {
25         minimum = i;
26         int j;
27         for(j = i + 1; j < array_size; ++j)
28             if(array[j] < array[minimum])
29                 minimum = j;
30
31         swap_ints(&array[i], &array[minimum]);
32     }
33 }
```



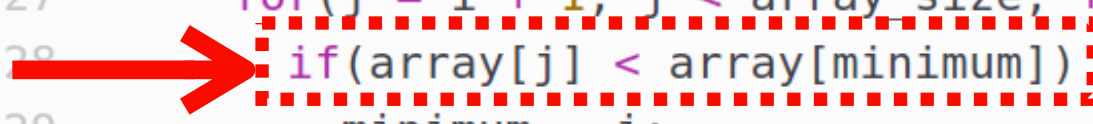
► The program executes the current line and step to the next line

# Step-by-step execution (2)

---

> (gdb) next

```
18 void sort(int* array, int array_size)
19 {
20     int i;
21     int minimum;
22
23     for(i = 0; i < array_size - 1; ++i)
24     {
25         minimum = i;
26         int j;
27         for(j = i + 1; j < array_size; ++j)
28             if(array[j] < array[minimum])
29                 minimum = j;
30
31         swap_ints(&array[i], &array[minimum]);
32     }
33 }
```



- ▶ The program executes the current line and step to the next line
- ▶ Each time the program's state is updated

# Execute to next breakpoint

---

- ▶ Continue execution until next breakpoint or end of execution  
`>(gdb) continue`
- ▶ If another breakpoint is hit, the execution will halt again

```
(gdb) info breakpoints
Num      Type             Disp Enb Address                  What
3        breakpoint       keep y   <PENDING>             my_program.c:25 if i < 1
4        breakpoint       keep y   0x0000555555554842     in sort at anagrams.c:25
                                stop only if i > 2
                                breakpoint already hit 2 times
```

Removes  
breakpoint 3

- ▶ Remove a breakpoint  
`>(gdb) delete breakpoint 3`
- ▶ Remove all breakpoints  
`>(gdb) delete breakpoints`

# An overview of standard breakpoints

1

2

3

4

5

# Standard breakpoints (1)

---

- ◆ Method Breakpoint
- ◆ Conditional Breakpoint
- ◆ Hit Point
- ◆ Watchpoint
- ◆ Tracepoint
- ◆ Exception Breakpoint
- ◆ Common breakpoint options

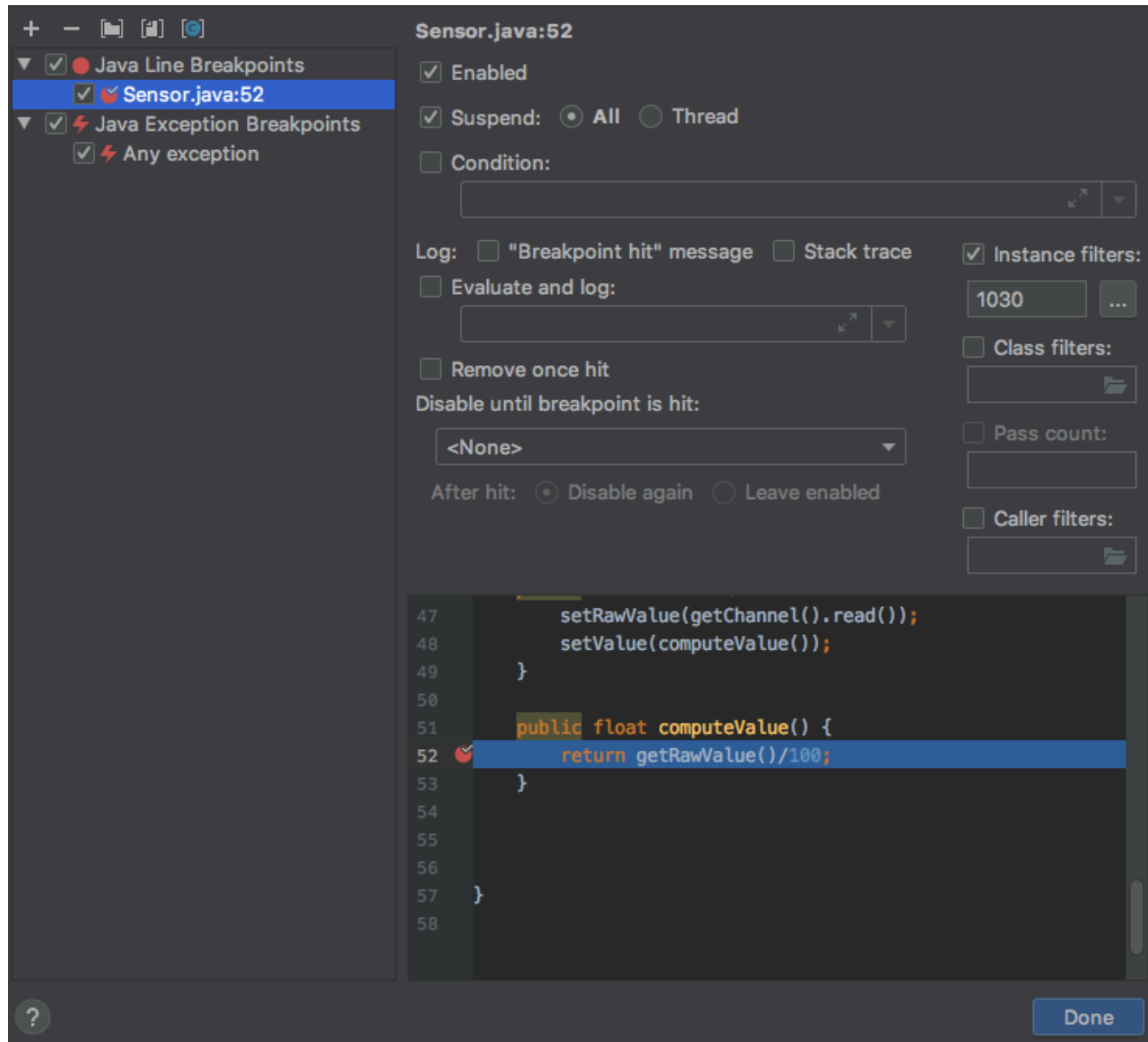


# Standard breakpoints (2)

Examples based in the IntelliJ CE debugger

Using breakpoints is a typical means of:

- ▶ Exploring a running program while tracking a bug
- ▶ Narrowing down a bug investigation to a very specific scope



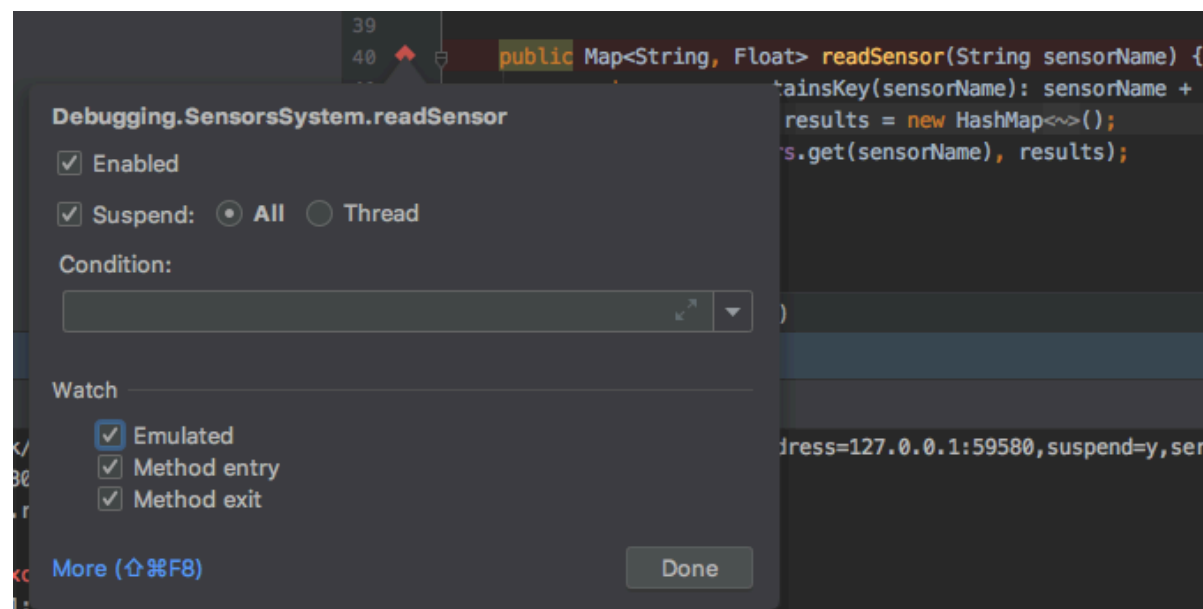
# Method Breakpoint

- Breaks execution when entering or returning from a method (or both)

**Enter**

```
40 public Map<String, Float> readSensor(String sensorName) {  
41     Map<String, Float> results = new HashMap<>();  
42     if(!sensors.containsKey(sensorName)) return results;  
43     readSensorIn(sensors.get(sensorName), results);  
44     return results;  
45 }
```

**Exit**



**Configuration**

# Conditional Breakpoint

---

- ◆ Breaks execution when a user-defined condition is satisfied
- ◆ Conditions are expressed on the state of the program
  - ▶ Return value of a method
  - ▶ Value of a variable

# Hit Point

- ❖ Breaks execution when the breakpoint has been reached a given number of times  $n$
- ❖ Implies that the execution flow « went »  $n$  times through the breakpoint

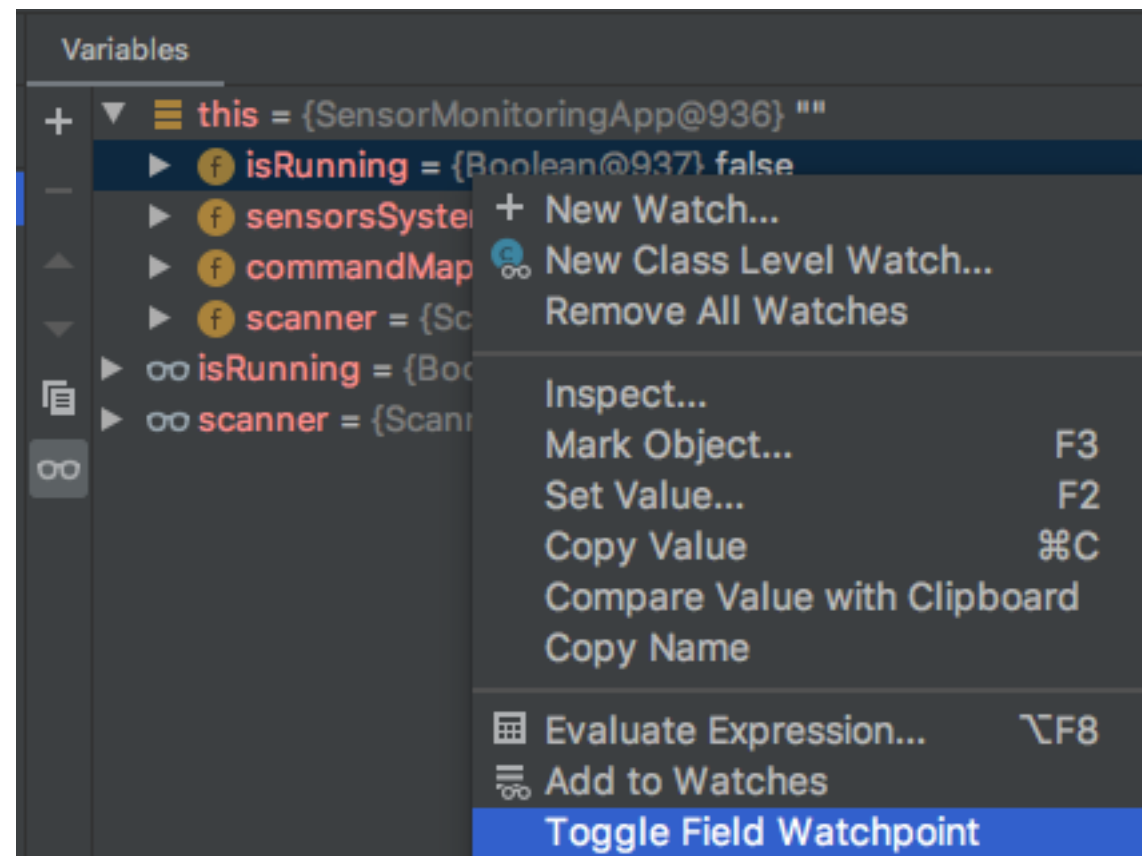
**Break  
after 5  
hits**

```
28 public void run() {
29     isRunning = true;
30     while (isRunning) {
31         String inputCommand = scanner.nextLine(); inputCommand: "sound" scanner: "java.util.Scanner[delimiters
32         if (inputCommand == "quit") {
33             isRunning = false; isRunning: true
34             System.out.println("Quitting");
35             break;
36         }
37
38         System.out.println("=====");
39         getCommandFromInput(inputCommand.toLowerCase()).execute(sensorsSystem).forEach((sensorName, sensorValue)
40             -> System.out.println(sensorName + ": " + sensorValue));
41         System.out.println("=====");
42     }
43 }
44 }
```

# Watchpoint

- ◆ Watch the value of a variable and breaks execution if the value changes
- ◆ Can monitor read and write access or both

- ▶ Set a first breakpoint and execute your program
- ▶ When it breaks, select a field and right-click «Toggle Field Watchpoint»



# Tracepoint (1)

---

## Conditional breakpoint trick:

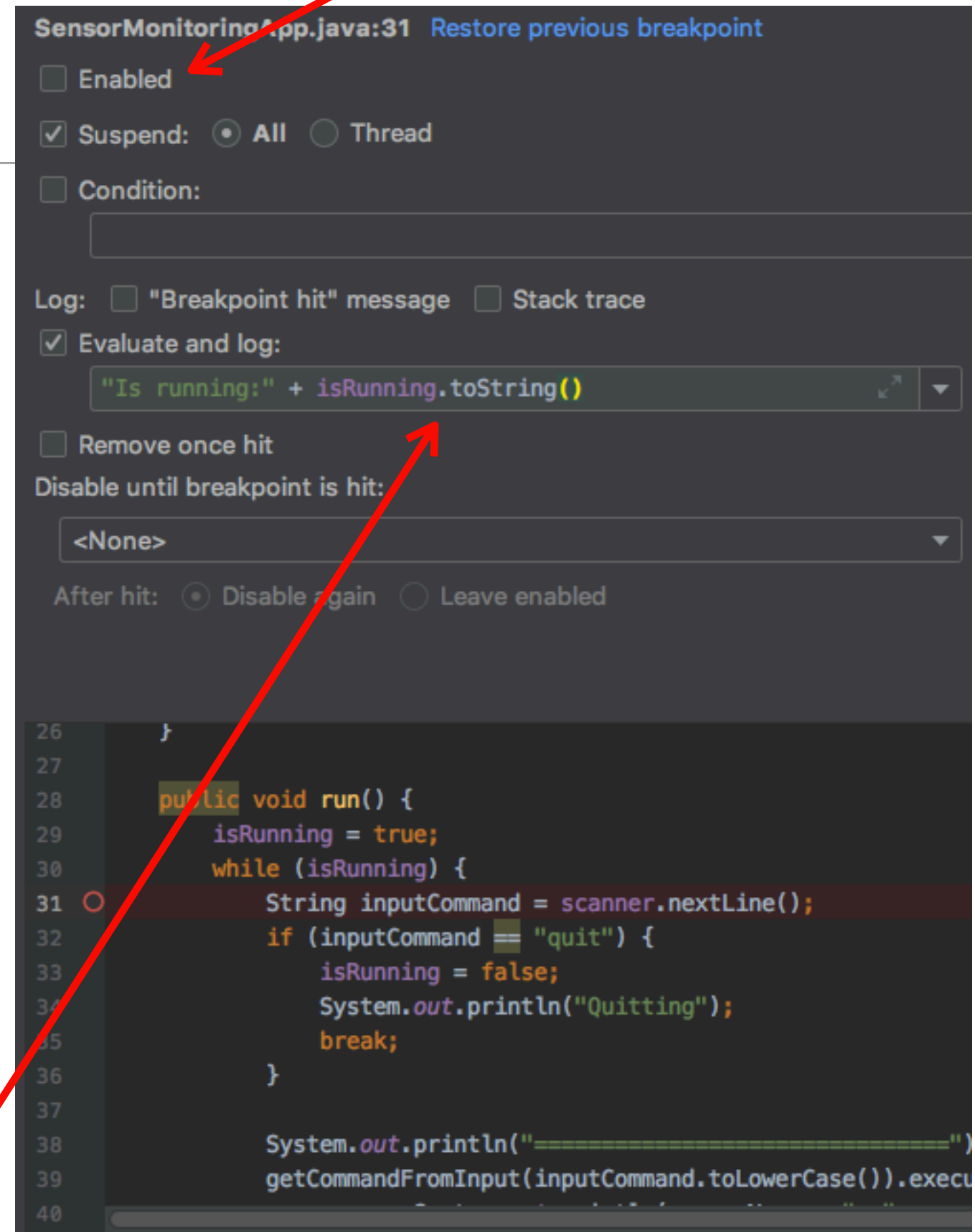
- ▶ Set a conditional breakpoint with a condition that is never satisfied: the breakpoint will never break the execution!
- ▶ Execute user-defined code when reaching the breakpoint! (traces)

**Will not break execution**

## Tracepoint (2)

### Conditional breakpoint trick:

- ▶ Set a conditional breakpoint with a condition that is never satisfied: the breakpoint will never break the execution!
- ▶ Execute user-defined code when reaching the breakpoint! (traces)



**Access to current execution scope state**

**Will not break execution**

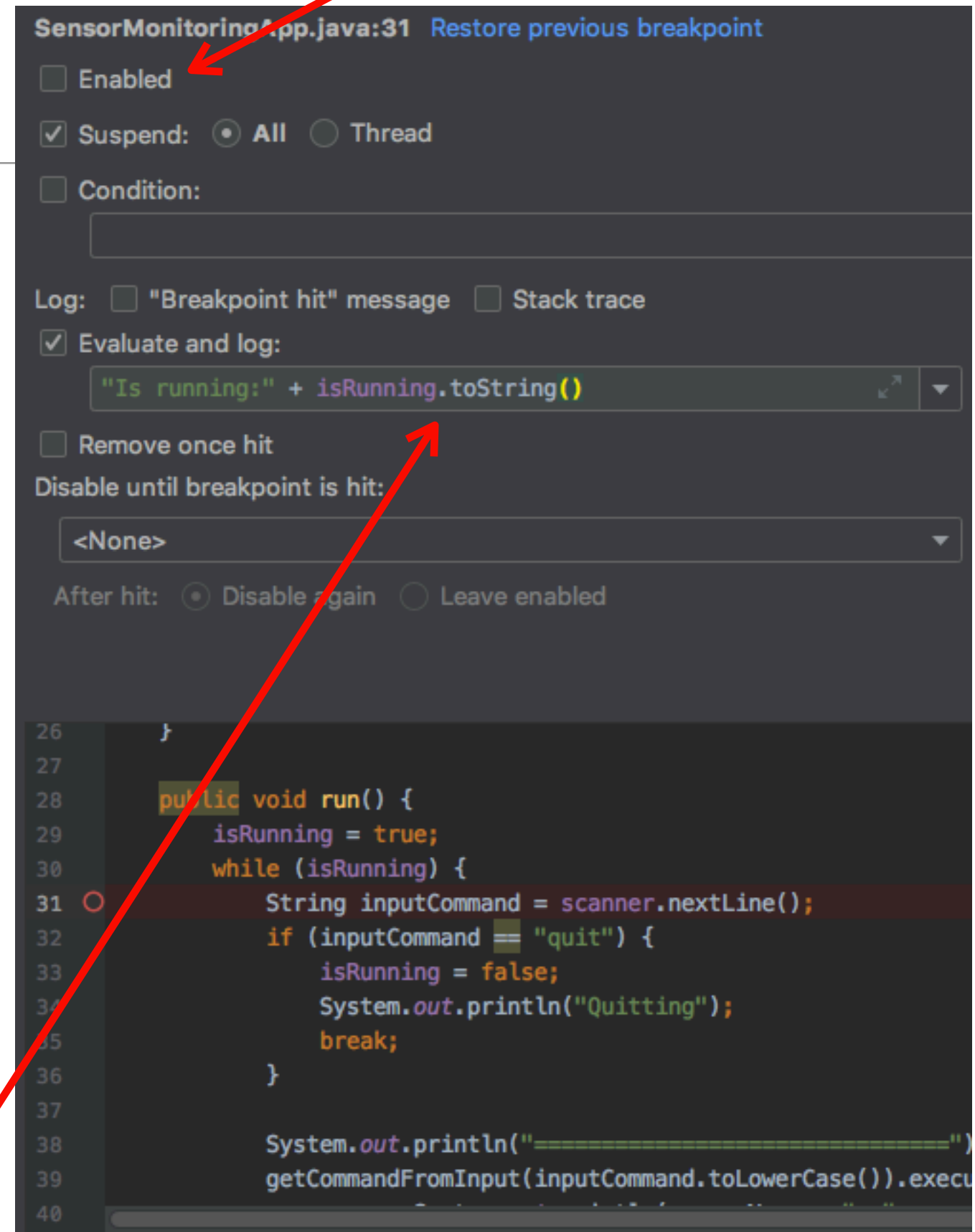
## Tracepoint (3)

### Conditional breakpoint trick:

- ▶ Set a conditional breakpoint with a condition that is never satisfied: the breakpoint will never break the execution!
- ▶ Execute user-defined code when reaching the breakpoint! (traces)

### **Beware of side effects!**

- ▶ Executing code may change the state of the program and provokes inconsistent behaviour or bugs/crashes



**Access to current execution scope state**



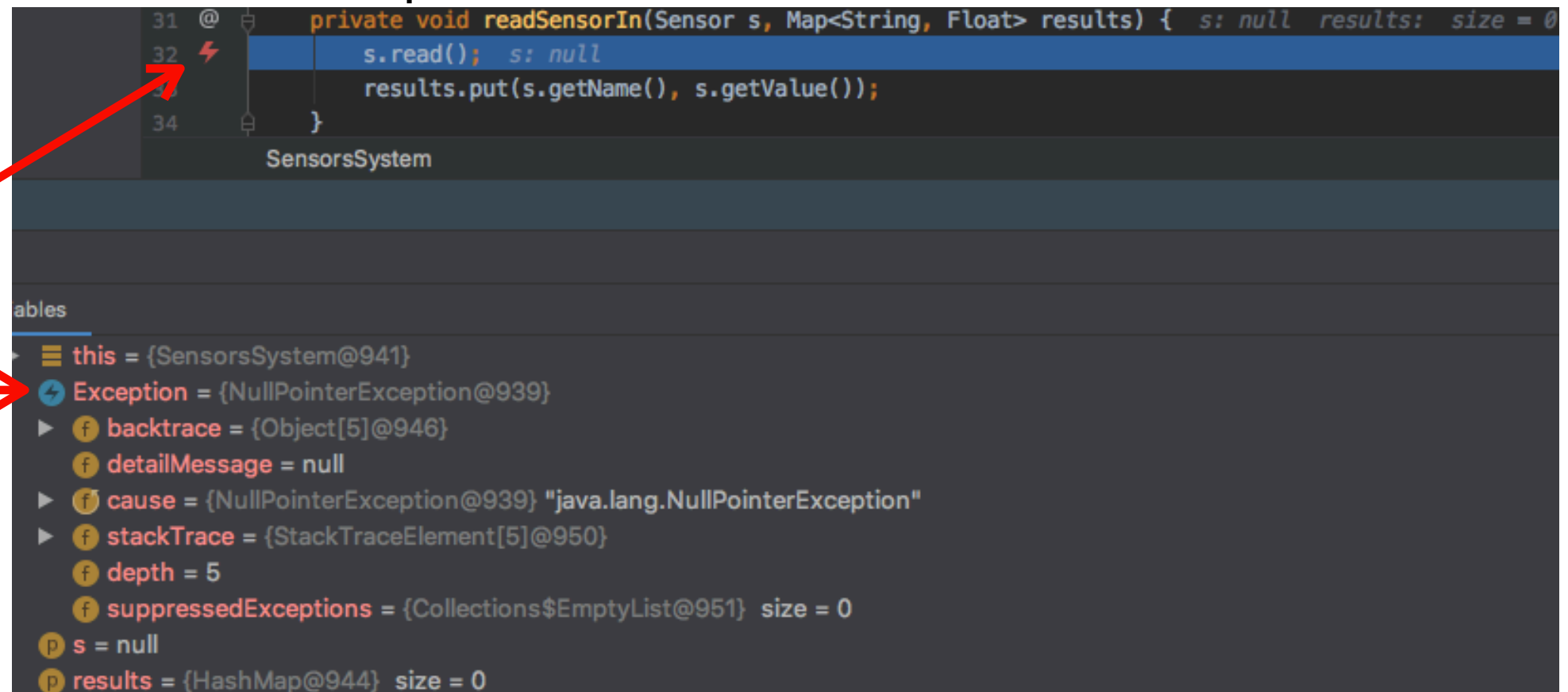
# Exception Breakpoint (1)

---

- ◆ Breaks the execution when...
  - ▶ A particular exception is caught, for example `NullPointerException`
  - ▶ When a particular exception is uncaught
  - ▶ Or when any kind of exception is (un)caught

# Exception Breakpoint (2)

- ❖ Breaks the execution when...
  - ▶ A particular exception is caught, for example `NullPointerException`
  - ▶ When a particular exception is uncaught
  - ▶ Or when any kind of exception is (un)caught
- ❖ Allows you to inspect your program's state at the moment of the exception



The screenshot shows a Java code editor with a file named `SensorsSystem`. The code is as follows:

```
31 @private void readSensorIn(Sensor s, Map<String, Float> results) { s: null results: size = 0
32 ⚡ s.read(); s: null
33 results.put(s.getName(), s.getValue());
34 }
```

An exception breakpoint is set on line 32, indicated by a red lightning bolt icon. Below the code editor, the 'Variables' pane shows the state of the program at the time of the exception:

- `this` = {SensorsSystem@941}
- `Exception` = {NullPointerException@939}
- `backtrace` = {Object[5]@946}
- `detailMessage` = null
- `cause` = {NullPointerException@939} "java.lang.NullPointerException"
- `stackTrace` = {StackTraceElement[5]@950}
- `depth` = 5
- `suppressedExceptions` = {Collections\$EmptyList@951} size = 0
- `s` = null
- `results` = {HashMap@944} size = 0

**Uncaught exception**

# Common breakpoints options (1)

---

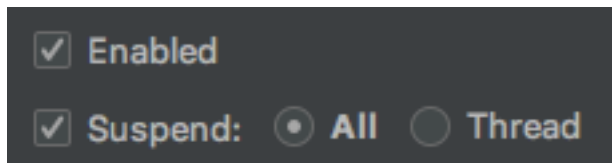
- ◆ Suspend options: current thread of virtual machine (all threads)

☒ Enabled  
☒ Suspend: ☒ All ☐ Thread

# Common breakpoints options (2)

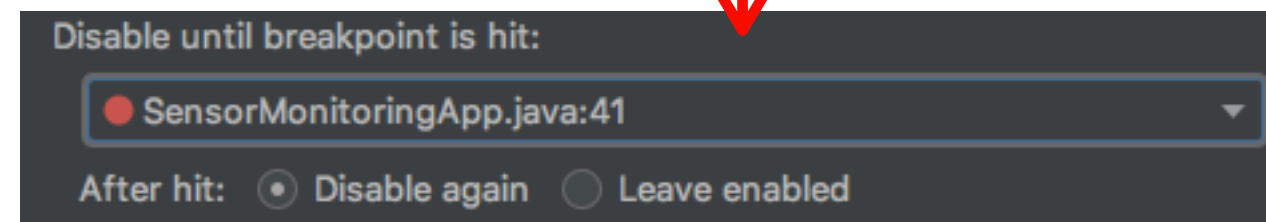
---

- ◆ Suspend options: current thread of virtual machine (all threads)



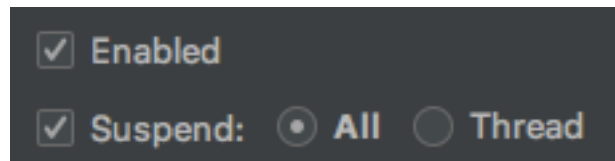
- ◆ Chained breakpoint

- ▶ Enable a breakpoint only after a previous breakpoint has been triggered
- ▶ The breakpoint can either be left enabled or disabled after hit



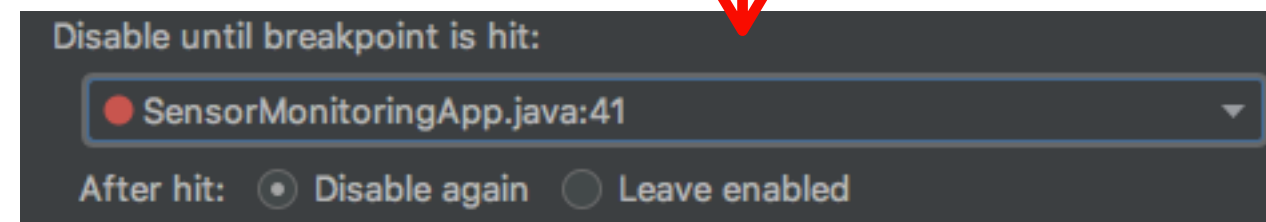
# Common breakpoints options (3)

- ◆ Suspend options: current thread of virtual machine (all threads)



- ◆ Chained breakpoint

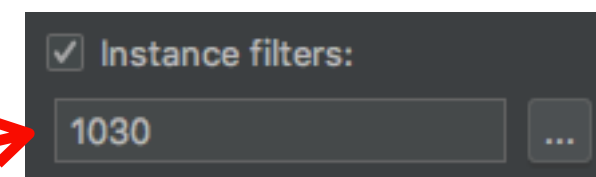
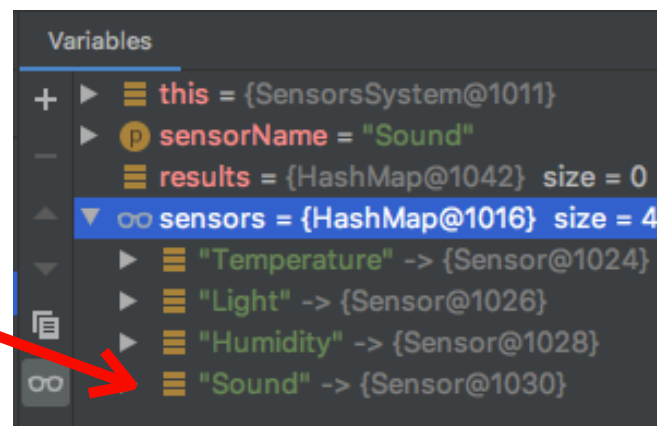
- ▶ Enable a breakpoint only after a previous breakpoint has been triggered
- ▶ The breakpoint can either be left enabled or disabled after hit



- ◆ Object-centric breakpoint

- ▶ Filter the instances for which the breakpoint is enabled
- ▶ Specify the target instances by giving their ID

**Find object's ID**



**Configure breakpoint**

# Common breakpoints options (4)

---

- Breakpoints can be dynamically (de)activated
- Breakpoints can be composed
  - Example : Tracepoint + condition + object-centric + hitpoint
- They can be removed when they are hit
- Usually they can be configured with a lot of filters:
  - **Callers:** only break if the current method was called (nor not called) from a specific method
  - **Classes:** which classes in which a breakpoint should be hit

# References

---

- 1.<https://www.gnu.org/software/gdb/documentation/>
- 2.[http://kirste.userpage.fu-berlin.de/chemnet/use/info/gdb/gdb\\_8.html](http://kirste.userpage.fu-berlin.de/chemnet/use/info/gdb/gdb_8.html)
- 3.[http://cseweb.ucsd.edu/classes/fa09/cse141/tutorial\\_gcc\\_gdb.html](http://cseweb.ucsd.edu/classes/fa09/cse141/tutorial_gcc_gdb.html)
- 4.**Debugging with Gdb: The Gnu Source-level Debugger twelve Edition**, for Gdb Version, January 2018
- 5.**Debugging: The 9 indispensable rules for finding even the most elusive software and hardware problems**, David J. Agans, 2002
- 6.**Why Programs Fail**, Andreas Zeller, 2009
- 7.**Effective Debugging**, Diomidis Spinellis, 2016

# TP

---

PDF à récupérer à cette adresse :

▶ [https://kloum.io/costiou/teaching/materials/practicals/debugging\\_TP\\_1.pdf](https://kloum.io/costiou/teaching/materials/practicals/debugging_TP_1.pdf)