

**Oliver Geisel**

Fakultät Informatik, Professur Softwaretechnologie

# Debugging mit IDEs

Einführung ins Debuggen von Code und das finden von Fehlern

2. Juni 2022

# Inhalt

1. Wie es nicht geht!
2. Grundlagen
3. Der Debugger und seine Bestandteile in einer IDE
4. Debuggen mit Tests
5. Das Beispiel Bibliothek

# Ziele

- **Kennen der Debugging-Grundlagen**
- **Debugging mit der DIE**
- **Kennen der Test-Grundlagen**
- **Test mit IDEs**
- **Bugs durch Tests finden**
- **Motivation für Java-Features im Selbststudium**

# Was wird benötigt

**Ab Java 8 für Einführungsbeispiele**

**Ab Java 17 für das große Beispiel**

**IDE der Wahl**

**Das Repo Github.com....**

**Maven**

# Finde den Fehler!

```
public static int summeVonBis(int a, int b) {  
    if (a > b) {  
        int temp = a;  
        a = b;  
        b = temp;  
    }  
    int summe = 0;  
    for (int run = a; run < b; run++) {  
        summe += run;  
    }  
    return summe;  
}
```

```
public static void main(String[] args) {  
    // Gib die Summe der Zahlen von a bis b (inklusive a,b) aus!  
    // es können auch negative Zahlen eingegeben werden  
    int a = 2;  
    int b = 7;  
    int c = summeVonBis(a, b);  
  
    System.out.print("Summe der Zahlen a bis b = " + c); // Hier ist es 20  
}
```

**Ergebnis ist 20**

**Muss aber 27 sein**

# System.out.print()

Eingrenzen eines Fehlers durch setzen von `System.out.print(variable)` an einer Stelle im Code

## Probleme:

- Nicht unterbrechbar
- Muss weitere Informationen mit liefern
- Bedingungen/ Filterung muss programmiert werden
- Keine Veränderung

```
Summe ist: 2  
run ist: 3  
Summe ist: 5  
run ist: 4  
Summe ist: 9  
run ist: 5  
Summe ist: 14  
run ist: 6  
Summe ist: 20  
Summe am Ende der Methode: 20  
Summe der Zahlen a bis b = 20  
Process finished with exit code 0
```

```
if (a > b) {  
    int temp = a;  
    a = b;  
    b = temp;  
}  
  
int summe = 0;  
for (int run = a; run < b; run++) {  
    System.out.println("run ist: " + run);  
    summe += run;  
    System.out.println("Summe ist: " + summe);  
}  
  
System.out.println("Summe am Ende der Methode: " + summe);  
return summe;
```

# Logging ist nicht viel besser - Aber eine Unterstützung

## Erzeugen eines Logger-Objektes

## Kategorisierung durch Log-Level

### Probleme:

- Nicht unterbrechbar
- Keine Veränderung

```
May 05, 2022 11:48:03 PM BreakpointLogging summeVonBis
INFO: run ist: 5
May 05, 2022 11:48:03 PM BreakpointLogging summeVonBis
INFO: Summe ist: 14
May 05, 2022 11:48:03 PM BreakpointLogging summeVonBis
INFO: run ist: 6
May 05, 2022 11:48:03 PM BreakpointLogging summeVonBis
INFO: Summe ist: 20
May 05, 2022 11:48:03 PM BreakpointLogging summeVonBis
INFO: Summe am Ende der Methode: 20
Summe der Zahlen a bis b = 20
Process finished with exit code 0
```

```
private static Logger log = Logger.getLogger( name: "Debug-Logger");
1 usage
public static int summeVonBis(int a, int b) {
    if (a > b) {
        int temp = a;
        a = b;
        b = temp;
    }
    int summe = 0;
    for (int run = a; run < b; run++) {
        log.info( msg: "run ist: " + run);
        summe += run;
        log.info( msg: "Summe ist: " + summe);
    }
    log.info( msg: "Summe am Ende der Methode: " + summe);
    return summe;
}
```

# Bugs

**Ein Bug ist ein (Laufzeit-)Fehler eines Programmes**

**Arten eines Bugs:**

- Syntax
- Semantisch
- Logisch

**Zeigen sich im „besten Fall“ als:**

- Ein falsches Ergebnis
- Ein Absturz

**Im schlimmsten Fall unentdeckt**





# Stack Trace - Den Ort des Fehlers eingrenzen

Informiert, dass eine nicht gefangene Exception auftrat -> Programm bricht ab

Enthält:

- Threadname
- Typ der Exception
- Fehlermeldung/Information
- „Methoden-Stack“

```
Exception in thread "main" java.lang.IllegalArgumentException Create breakpoint : java.util.NoSuchElementException: No value present
    at ExceptionBeispiele.gemeineException(ExceptionBeispiele.java:35)
    at ExceptionBeispiele.main(ExceptionBeispiele.java:71)
Caused by: java.util.NoSuchElementException Create breakpoint : No value present
    at java.base/java.util.Optional.get(Optional.java:143)
    at ExceptionBeispiele.tiefeException(ExceptionBeispiele.java:24)
    at java.base/java.util.Optional.orElseGet(Optional.java:364)
    at ExceptionBeispiele.arbeiten(ExceptionBeispiele.java:48)
    at ExceptionBeispiele.gemeineException(ExceptionBeispiele.java:32)
    ... 1 more
```

# Stack Trace – Beispiel 1

Wo ist die Exception passiert?

In Zeile 15

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "String.toUpperCase()" because "ExceptionBeispiele.hallo" is null
    at ExceptionBeispiele.NullPointerBeispiel(ExceptionBeispiele.java:15)
    at ExceptionBeispiele.main(ExceptionBeispiele.java:62)
```

```
Process finished with exit code 1
```

# Stack Trace – Beispiel 2

```
Exception in thread "main" java.lang.RuntimeException Create breakpoint : Zur Laufzeit gab es einen Fehler!  
    at ExceptionBeispiele.einfacheException(ExceptionBeispiele.java:19)  
    at ExceptionBeispiele.main(ExceptionBeispiele.java:65)
```

# Stack Trace – Beispiel 3

```
Exception in thread "main" java.util.NoSuchElementException Create breakpoint : No value present  
    at java.base/java.util.Optional.get(Optional.java:143)  
    at ExceptionBeispiele.tiefeException(ExceptionBeispiele.java:24)  
    at ExceptionBeispiele.main(ExceptionBeispiele.java:68)
```

# Stack Trace – Beispiel 4

Wo ist eine Exception passiert?

In Zeile 35 und 143

```
Exception in thread "main" java.lang.IllegalArgumentException Create breakpoint : java.util.NoSuchElementException: No value present
    at ExceptionBeispiele.gemeineException(ExceptionBeispiele.java:35)
    at ExceptionBeispiele.main(ExceptionBeispiele.java:71)
Caused by: java.util.NoSuchElementException Create breakpoint : No value present
    at java.base/java.util.Optional.get(Optional.java:143)
    at ExceptionBeispiele.tiefeException(ExceptionBeispiele.java:24)
    at java.base/java.util.Optional.orElseGet(Optional.java:364)
    at ExceptionBeispiele.arbeiten(ExceptionBeispiele.java:48)
    at ExceptionBeispiele.gemeineException(ExceptionBeispiele.java:32)
    ... 1 more
```

# Der Debug-Modus

- Startet Java-Anwendung innerhalb einer Debug-Umgebung
  - Debugger kontrolliert/ überwacht die Anwendung
  - Liefert detaillierte Informationen in speziellen Fenstern
  - Steuerung des Ablaufes
- 
- Start mit extra Knopf





## Kontrollelemente

### Dateien

### Editor

### Frames

### Variablen

### Memory

Overhead		Memory
Class	Count	Diff
java.lang.module.ModuleDescript	370	0
java.util.HashMap	327	0
java.util.HashMap\$Node[]	324	0
java.util.HashSet	262	0
java.lang.Integer	262	0
java.util.ImmutableCollections\$Se	226	0
java.lang.module.ModuleDescript	168	0

# Der Breakpoint

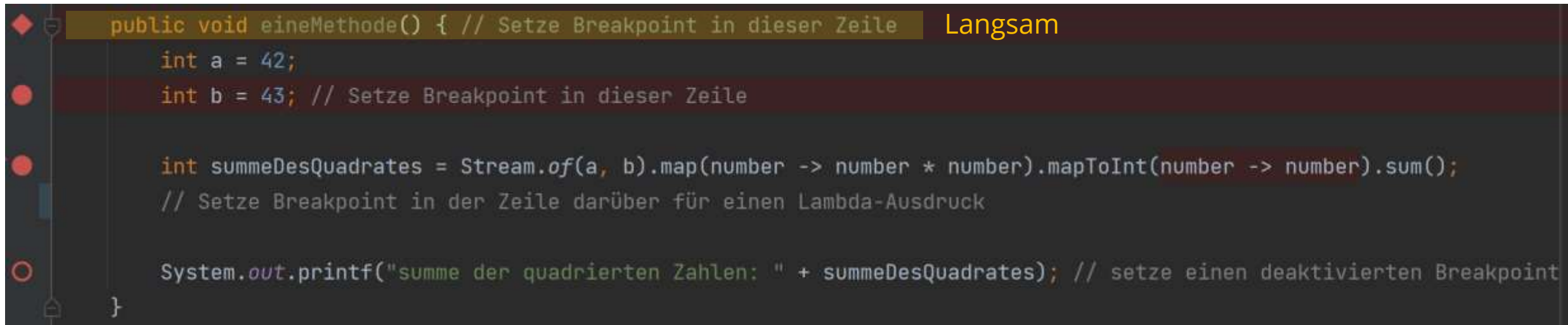
Eine Markierung in einer Zeile, die im Debug-Modus, die Ausführung anhält

Normalfall: an der linken Seite wird ein Punkt gesetzt

Spezialfall: Lambda, Methoden, Verschachtelung in Methoden- hängt von IDE ab

Sind deaktivierbar oder entfernbar

- 1.Methode
- 2.Zeile
- 3.Lambda
- 4.Deaktiviert



```
public void eineMethode() { // Setze Breakpoint in dieser Zeile
    int a = 42;
    int b = 43; // Setze Breakpoint in dieser Zeile

    int summeDesQuadrates = Stream.of(a, b).map(number -> number * number).mapToInt(number -> number).sum();
    // Setze Breakpoint in der Zeile darüber für einen Lambda-Ausdruck

    System.out.printf("summe der quadrierten Zahlen: " + summeDesQuadrates); // setze einen deaktivierten Breakpoint
}
```

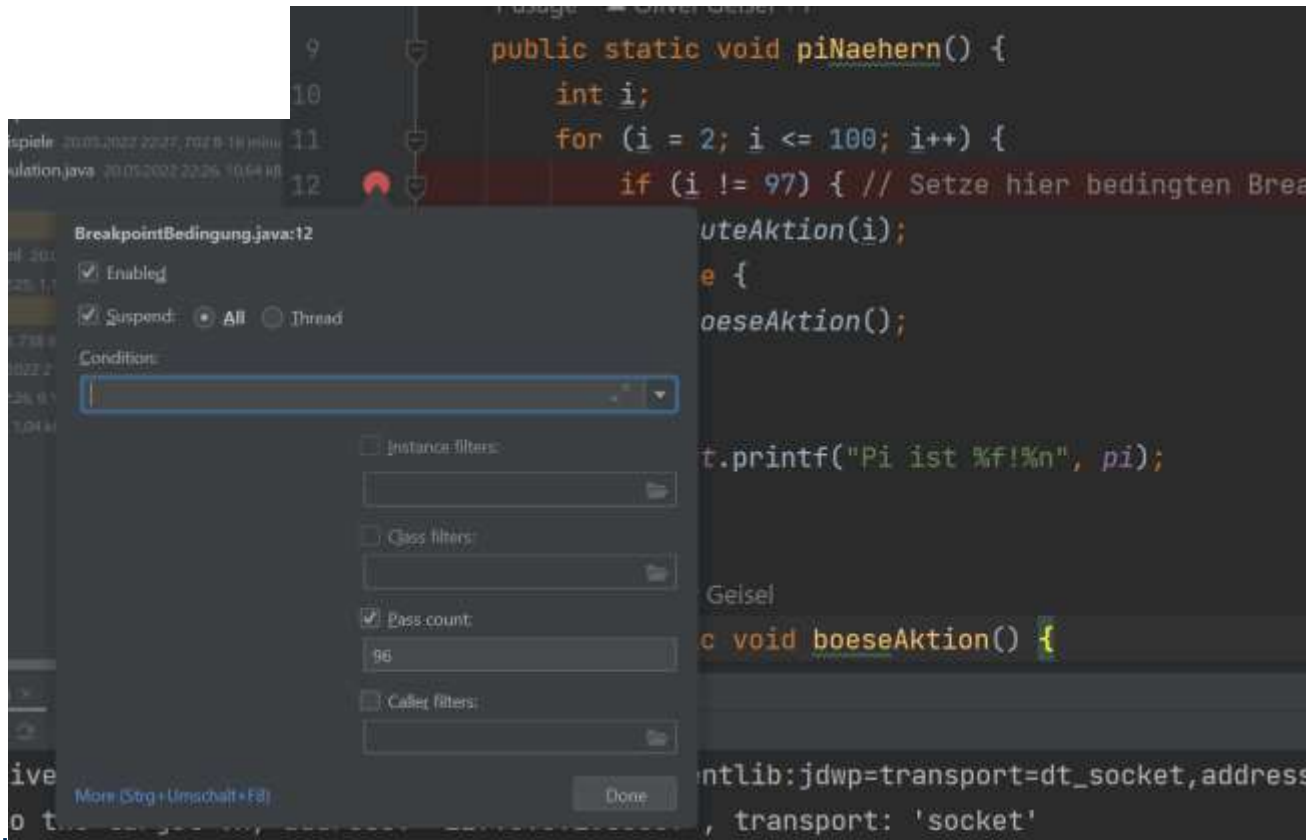


# Bedingte Breakpoints

Breakpoints können auch halten wenn:

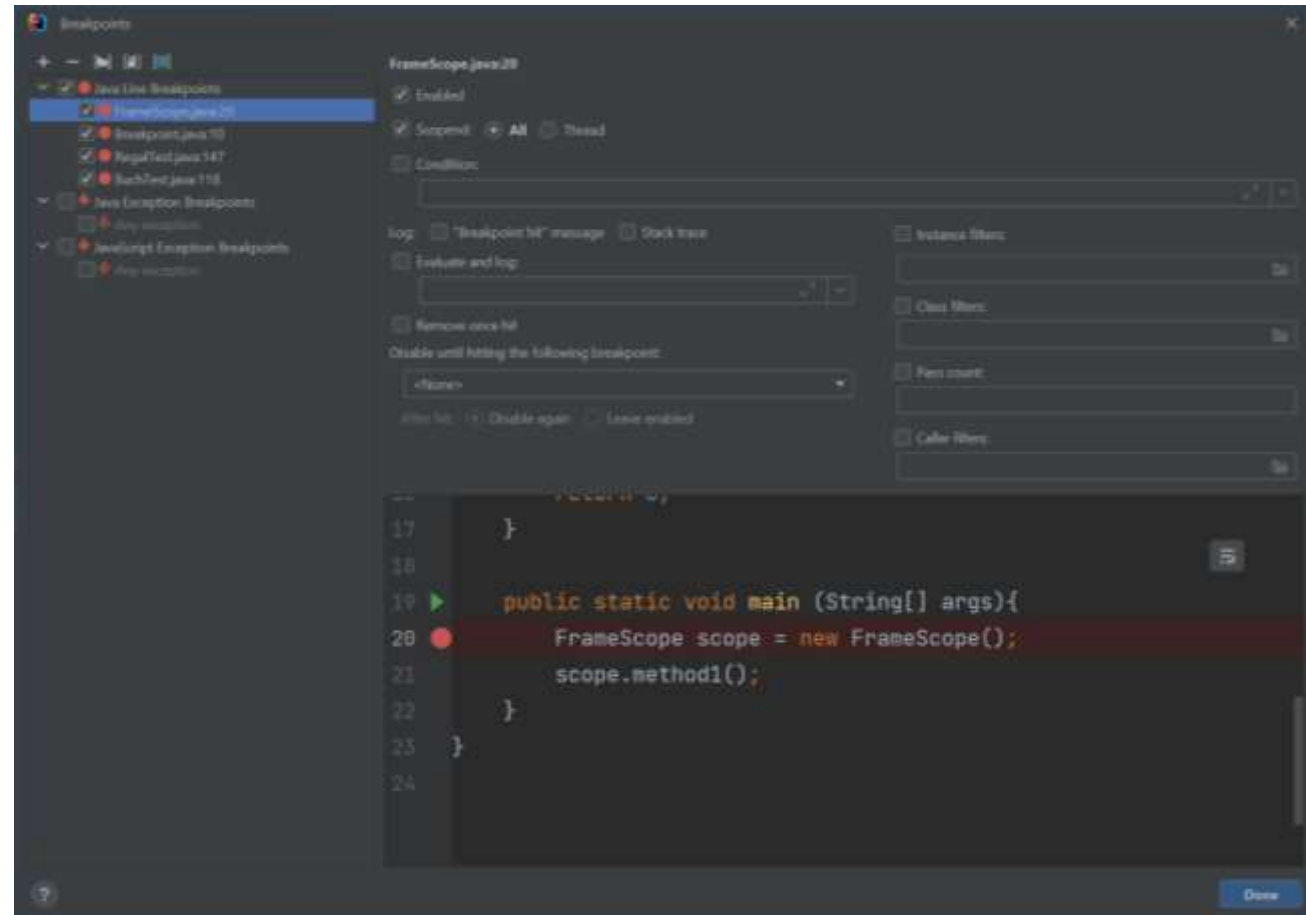
- Eine Expression true ist
- Eine Schleife n-Mal durchlaufen ist

```
Pi ist am Anfang: 3.14072
Achtung! Der Prozess ist sehr rechenintensiv!
Abbruch mit Ctrl + C!
Pi ist 3,000000!
Benötigte Zeit: 16,774375800 sec.
```



# Breakpoint-Fenster

- Zeigt alle gesetzten Breakpoints
- Erlaubt diese zu aktivieren/ deaktivieren
- oder auch wieder zu entfernen
- Je nach IDE können Bedingungen hinzugefügt werden
- Logging ist möglich



# Das Programmfenster

Breakpoint in Zeile 11

Blaue Zeile markiert, wo das Programm aktuell ist  
Zeile wurde noch nicht ausgeführt!

```
4 public static int summeVonBis(int a, int b) { a: 2 b: 7
5     if (a > b) {
6         int temp = a;
7         a = b;
8         b = temp;
9     }
10    int summe = 0; summe: 0
11    for (int run = a; run < b; run++) { a: 2 b: 7
12        summe += run;
13    }
14    return summe;
```

Extra Informationen  
zur Laufzeit

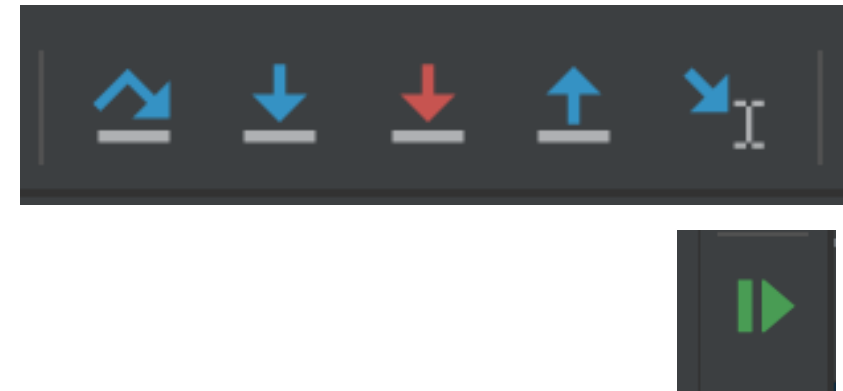


Springt zur  
aktuellen Zeile

# Schritt für Schritt zum Fehler

Es gibt verschiedene Varianten um das Programm weiter auszuführen

1. **Step Over** – führt nächsten Befehl aus, geht nicht in die Methode hinein
2. **Step Into** – geht in die nächste Methode hinein
3. **Step Out** – geht eine „Ebene“ höher
4. **Resume** – Programm läuft weiter
  - Bis zum nächsten Breakpoint
  - Bis zur Cursor-Position



# Variablen-Fenster

- Enthält alle Variablen im aktuellem Frame
- Man kann auch weitere Variablen überwachen; werden immer aufgelistet
- Alle Variablen können angesehen und geändert werden
- Manche Watches werden von der IDE hinzugefügt

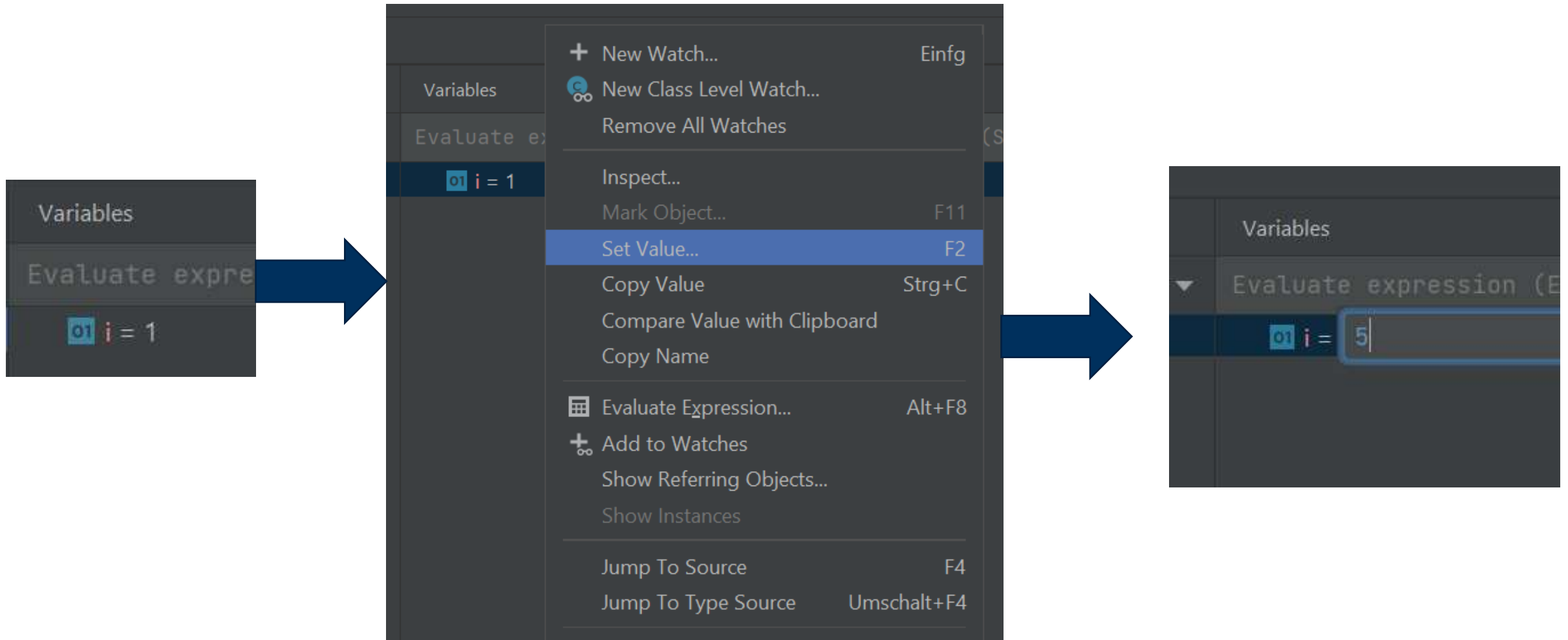
```
public static void level2() {  
    System.out.println("Start Level 2!\nViel Erfolg!");  
    final Level2Objekt zeus = new Level2Objekt( name: "Zeus", wert: 42, preis: 3.14); zeus: Lev  
    final Level2Objekt jupiter = new Level2Objekt( name: "Jupiter", wert: 42, preis: 3.14); jup  
    if (zeus.equals(jupiter)) { zeus: Level2Objekt@536 jupiter: Level2Objekt@538  
        // ...  
    }  
}
```

Variables

Evaluate expression (Eingabe) or add watch

- ▼ **zeus** = {Level2Objekt@536}
  - > **name** = "Zeus"
  - wert** = 42
  - preis** = 3.14
- ▼ **jupiter** = {Level2Objekt@538}
  - > **name** = "Jupiter"
  - wert** = 42
  - preis** = 3.14

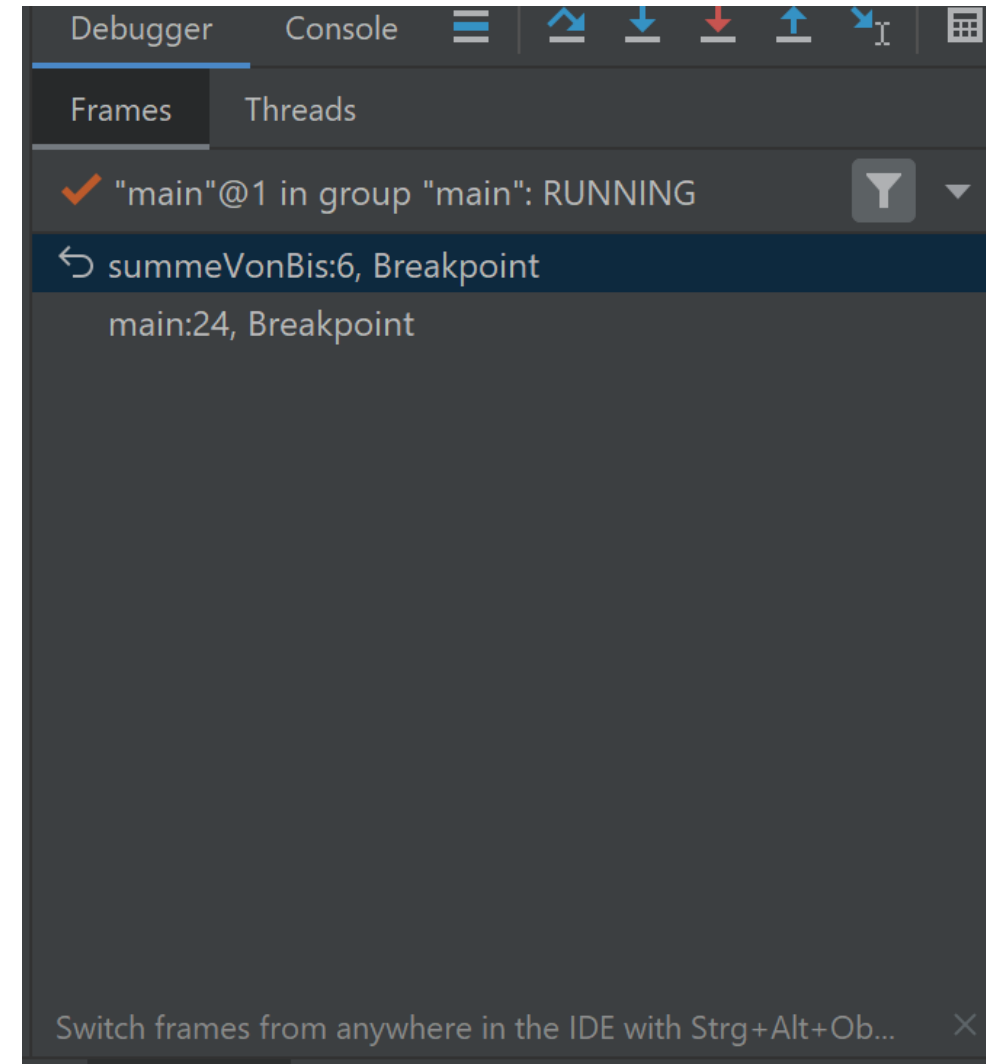
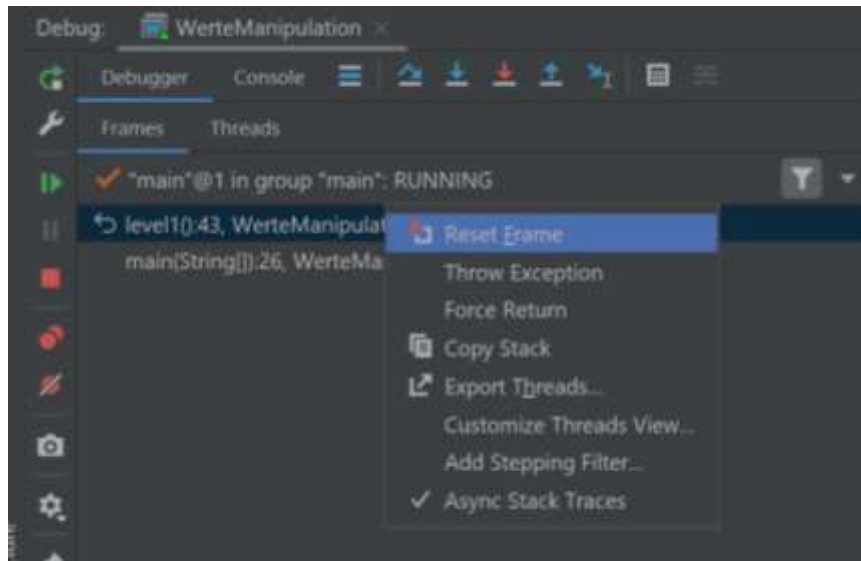
# Variablen-Fenster – Verändern von Werten



# Frame Stack-Fenster

In Java-VM ist jede Methode in einem Frame – Siehe [3] 2.6  
Dazu gehören die entsprechenden Variablen  
Frames können durchsucht werden

Ein Frame kann auch verworfen werden



# Extra Fenster - Memory

- Zeigt *alle* Objekte im Speicher!
- Sind nach Klasse sortiert.
- Zeigt exakte Anzahl der Objekte
- Diff = Änderung seit letztem Update

Memory		Overhead
<input type="text"/>		
Class	Count	Diff ▼
byte[]	2700	0
java.lang.String	2615	0
java.util.HashMap\$Node	1198	0
java.util.concurrent.ConcurrentHashMap\$Node	1014	0
java.lang.Object[]	990	0
java.lang.Class	750	0
java.lang.module.ModuleDescriptor\$Exports	370	0
java.util.HashMap	327	0
java.util.HashMap\$Node[]	324	0
java.util.HashSet	262	0
java.lang.Integer	262	0
java.util.ImmutableCollections\$Set12	226	0
java.lang.module.ModuleDescriptor\$Requires	168	0
java.util.ImmutableCollections\$SetN	127	0
java.lang.invoke.MethodType	95	0
java.lang.invoke.MethodType\$ConcurrentWeakInternSet\$WeakEntry	95	0
java.lang.invoke.LambdaForm\$Name	93	0
java.lang.invoke.MemberName	79	0
java.lang.Class[]	79	0
java.lang.Object	77	0
java.lang.invoke.ResolvedMethodName	72	0



# Weitere Hilfen - IntelliJ

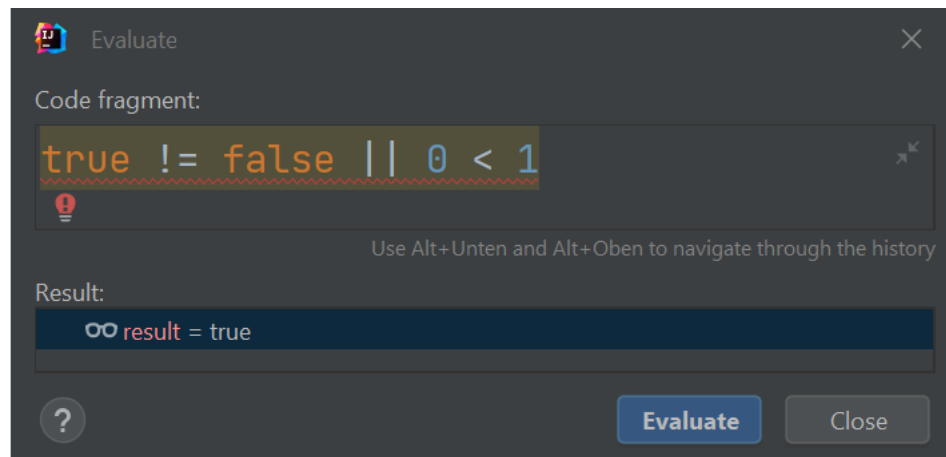
## Expression Evaluator

(Alt + F8)

Kann jeden Ausdruck auswerten

Achtung kann Schaden anrichten!

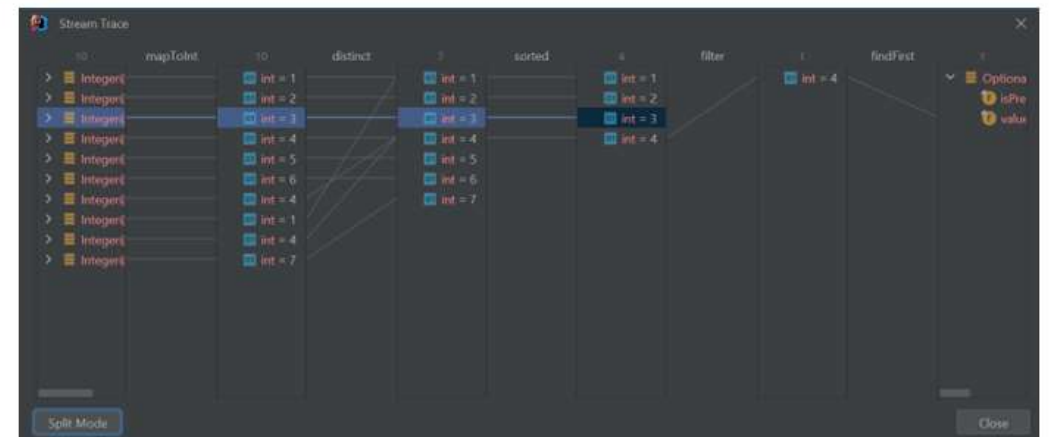
```
if (a != 0
    && 5 % 2 == 1
    || true ^ false
    && (true != false || 0 < 1)) {
```



## Chain Stream

Benötigt ausführende Aktion

```
var test : Stream<Integer> = Stream.of(...values: 1, 2, 3, 4, 5, 6, 4, 1, 4, 7);
test.mapToInt(it -> it)
    .distinct()
    .sorted()
    .filter(it -> it > 3)
    .findFirst();
```



# Tests

- Prüft zur Laufzeit, ob ein bestimmter **Fehler** auftrat
- Geben die Abweichung vom **Sollzustand** an.
- Geben nicht die **Ursache** an
- Jeder Test sollte eine **Fehler-Message** besitzen



## Einordnung von Tests:

### Nach „Stufen“

- Unit-Test
- Integration-Test
- System-Test
- Abnahme-Test

(Siehe V-Modell)

### Nach „Akteur“

- Manuel
- Automatisch

### Nach Art

- Funktional
- Nicht funktional
- Änderungsbezogen
- White-Box  
(nach [5])

### Nach „Wissen“

- White-Box
- Gray-Box
- Black-Box

# Tests können mehr als public testen

Gängige Praxis:

Klassen in: `src.project.section.MeineKlasse.java`

Tests in: `test.project.section.MeineKlasseTest.java`

Beide Klassen sind im package `project.section`

Dadurch hat `MeineKlasseTest.java` auf alle *public*, *package* und *protected* Methoden/Attribute Zugriff

```
package de.oliver.structure;
...
public abstract class Arbeitsplatz<T extends Person> implements Verschmutzbar {

    protected T nutzer;
    protected double verschmutzung;
```

```
package de.oliver.structure;
...
import static org.junit.jupiter.api.Assertions.*;

class ArbeitsplatzTest {

    @Test
    void isDreckigTrue(){
        assertFalse(arbeitsplatz.isDreckig(),"Am Anfang darf es nicht dreckig sein");
        var t = arbeitsplatz.verschmutzung;
        arbeitsplatz.verschmutzen();
```

# Tests müssen nicht private testen

- Private steuert das innere Verhalten des Objektes. Es ist für alle anderen unwichtig.
- Private Methoden sollen helfen, das äußere Verhalten zu erreichen.
- Private Methoden werden durch nicht private Methoden aufgerufen
- Wenn private explizit getestet werden muss, dann ist es schlechtes Design!
- Reflection erlaubt das Testen der privaten Methoden, aber es sollte sparsam eingesetzt werden.

# JUNIT 5.8.0

`assertX(Sollwert, Istwert, „Nachricht bei Fehler“)`

Liste der Annotationen [4]/[#writing-tests-annotations](#)

`@ParameterizedTest` – „Template“ für viele Testfälle

`@DisplayName` – Name im Testergebnis

`AssertAll` – alle enthaltenen `AssertX` werden ausgeführt und zusammen ausgegeben.

# Tests in einer IDE

- IDEs haben integrierte Test-Umgebungen
- Liefern Übersicht der Ergebnisse
- Können Test automatisch starten

❗ <default package>	47 sec 74 ms
> ✓ ArbeitsplatzTest	88 ms
> ✓ AngestellterTest	52 ms
> ✗ BreakpointTest	24 ms
▼ ❗ RegalTest	46 sec 695 ms
✗ isVollFalseBeimErzeugen()	1 ms
❗ alleBuecherOkay()	46 sec 628 ms
✗ leereVollesRegal()	29 ms
❗ enthaltBuchOkay()	2 ms
✓ nullBuecherAmAnfang()	31 ms
✓ iteratorTest()	1 ms
✓ toStringTest()	
✓ enthaeltNullTest()	1 ms
✓ isVollTrue()	1 ms
✗ isVollFalseBeimEntnehmenEinesBuches	1 ms
> ❗ TestErgebnisse	6 ms
> ✓ ISBNTest	148 ms
> ✗ BuchTest	61 ms

```
org.opentest4j.AssertionFailedError: Die Ausleihe eines Buches darf nicht gelingen, wenn es bei
<3 internal lines>
    at de.oliver.core.BuchTest.ausleihenFehlslag(BuchTest.java:27) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <27 internal lines>
```

# Arten von Testergebnissen

Test ist erfolgreich.  
Keine Fehler.

Test ist fehlgeschlagen.  
Ergebnis stimmt nicht.

Test ist fehlgeschlagen.  
Explizit durch den Entwickler!

Der Test wurde unerwartet unterbrochen.  
Eine Exception trat auf.

Test wird als abgebrochen markiert

Test wurde ignoriert.

```
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assumptions.assumeFalse;

public class TestErgebnisse {

    @Test
    public void erfolgreicherTest(){
        assertTrue( condition: true, message: "Der Test erfordert true");
    }

    @Test
    public void fehlerhafterTest(){
        assertTrue( condition: false, message: "Der Test erfordert true");
    }

    @Test
    public void fehlgeschlagenerTest(){
        fail("Dieser Test schlägt immer Fehl!");
    }

    @Test
    public void TestMitUnerwarteterException(){
        throw new IllegalArgumentException("Unerwartete Exception");
    }

    @Test
    public void abgebrochenerTest(){
        assumeFalse( assumption: true, message: "Die Annahme ist nicht Falsch!");
    }

    @Test
    @Disabled
    public void übersprungenerTest(){
        // Dieser Test wird nicht ausgeführt
    }
}
```



# Arten von Testergebnissen

Run: TestErgebnisse x

Tests failed: 3, passed: 1, ignored: 2

Testergebnis	Dauer	Details
fehlerhafterTest()	33 ms	org.opentest4j.AssertionFail Expected :true Actual :false <a href="#">Click to see difference</a>
erfolgreicherTest()	1 ms	
abgebrochenerTest()	6 ms	
übersprungenerTest()		
fehlgeschlagenerTest()	1 ms	
TestMitUnerwarteterException()	2 ms	<3 internal lines> at TestErgebnisse.fehlerhafterTest() at java.base/java.util.A at java.base/java.util.A

```
import static org.junit.jupiter.api.Assertions.*;
import static org.junit.jupiter.api.Assumptions.assumeFalse;

public class TestErgebnisse {

    @Test
    public void erfolgreicherTest(){
        assertTrue( condition: true, message: "Der Test erfordert true");
    }

    @Test
    public void fehlerhafterTest(){
        assertTrue( condition: false, message: "Der Test erfordert true");
    }

    @Test
    public void fehlgeschlagenerTest(){
        fail("Dieser Test schlägt immer Fehl!");
    }

    @Test
    public void TestMitUnerwarteterException(){
        throw new IllegalArgumentException("Unerwartete Exception");
    }

    @Test
    public void abgebrochenerTest(){
        assumeFalse( assumption: true, message: "Die Annahme ist nicht Falsch!");
    }

    @Test
    @Disabled
    public void übersprungenerTest(){
        // Dieser Test wird nicht ausgeführt
    }
}
```



# Die Bibliothek - Anforderungen

- **Maven**
- **Java 17**
- **Eine IDE wie Eclipse, IntelliJ, VS Code usw.**
  
- **Download des Repos von GitHub**
- **Entweder lokaler Import oder direkt vom Server in die IDE**

# Die Bibliothek - Situation

- Aus den ersten Übungen ist die Bibliothek bereits bekannt.
- Diese ist inzwischen zu einer großen Anwendung angewachsen.
- Es gibt Personal- und Besucher-Verwaltung sowie Leseräume.
- Zwei SHKs haben diese Anwendung gebaut.
- Der eine hat die Tests, der andere den Quellcode geschrieben.
- Leider sind im Quellcode Fehler. Dies wurde durch die Tests herausgefunden.
- Die SHKs sind nicht mehr angestellt und die Fehler müssen schnell behoben werden.

# Die Bibliothek - Aufgabe

**Finden der Bugs im Projekt anhand der Tests.**

**Die Tests sind korrekt.**

**Hinweis! Alle Stellen, die Features ab Java 8 haben, sind korrekt.**

**Die neuen Features sollen Motivation sein, sich selbst mit neuen Features in Java zu beschäftigen**

**In *aufgabe.md* sind weitere Informationen zu finden**

**2 Phasen:**

- 1. Mit Tests die Fehler eingrenzen**
- 2. Durch falsche Ausgaben/Exceptions Stellen suchen und beheben.**

# Elemente, die neu sind – Java 8+

- **Var**
- **Optional**
- **Stream**
- **Lambda**
- **Records**
- **Pattern matching for switch**
- **Pattern matching for instanceof**
- **Enhanced Switch**
- **Switch-Expression**
- **Textblocks**

# Die Bibliothek – Test analysieren

Run: All in Debug-Tutorial-bibliothek x

Tests failed: 42, passed: 144, ignored: 2 of 188 tests – 2 sec 788 ms

Test	Duration
<default package>	2 sec 788 ms
AngestelltenComputerTest	8 ms
AngestelltenVerwaltungTest	13 ms
AngestellterTest	23 ms
ArbeitsplatzTest	116 ms
BestandsVerwaltungTest	199 ms
BesucherComputerTest	4 ms
BesucherTest	10 ms
BibliothekTest	7 ms
BuchTest	42 ms
DozentTest	121 ms
ISBNTest	121 ms
KundenregisterTest	210 ms
LeseraumTest	81 ms
RegalTest	75 ms
StudierenderTest	2 ms
WerkstattTest	1 sec 877 ms

Expected :false  
Actual :true  
[Click to see difference](#)

org.opentest4j.AssertionFailedError  
at de.oliver.structure.RegalTest  
at java.base/java.util.ArrayList  
at java.base/java.util.ArrayList

>	BibliothekTest	7 ms
>	BuchTest	42 ms
>	KundenregisterTest	210 ms
>	strafeFuerDozent(int, double)	138 ms
>	bezahlenOkay()	27 ms
>	getAusgelieheneBuecherOkay()	1 ms
>	getStrafe1TagVorAbgabe()	1 ms
>	getStrafe7TageVorAbgabe()	2 ms
>	getStrafeAmAbgabeTag()	1 ms
>	getStrafeNach1Tag()	10 ms
>	getStrafeNach2Jahren()	2 ms
>	getStrafeNach7Tagen()	1 ms
>	getStrafeNach8Tagen()	2 ms
>	getStrafeNach14Tagen()	1 ms
>	getStrafeNach15Tagen()	1 ms
>	getStrafeNach39Wochen()	2 ms
>	getStrafeNach40Wochen()	2 ms
>	getStrafeNach41Wochen()	1 ms
>	getStrafeNach43Tagen()	1 ms
>	gibBuchZurueckFehlerNichtDemBesucherZugr	1 ms
>	gibBuchZurueckOkay()	1 ms
>	leiheBuchAusOkay()	1 ms

```
java.lang.NullPointerException Create breakpoint : Cannot invoke "java.util.List.add(Object)" because "this.ausgelieheneBuecher" is null
    at de.oliver.person.visitor.Kundenregister$BesucherStatus.registriereAusgeliehenesBuch(Kundenregister.java:155)
    at de.oliver.person.visitor.Kundenregister.leiheBuchAus(Kundenregister.java:62)
    at de.oliver.person.visitor.KundenregisterTest.getAusgelieheneBuecherOkay(KundenregisterTest.java:276) <31 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <9 internal lines>
    at java.base/java.util.ArrayList.forEach(ArrayList.java:1511) <27 internal lines>
```



# Lösung

Im „Lösung“-Branch sind die Lösungen zu finden. Diese sind mit Todo´s markiert.

**Mindestens 18 Bugs**

**Refactoring:**

- **Singleton**
- **Abstrakte Klassen einfügen**
- **State-Pattern**
- **Observer**

# Fürs Selbststudium

**Findet schlechte Ideen und verbessert Sie**  
**Nutzt SonarQube/SonarLint für Codequalität**  
**Erweitert die Bibliothek**  
**Nutzt TDD (Test-Driven-Development)**  
**Baut für Bücher das Interface Zerstörbar**  
**Baut bei ISBN die Prüfzifferkontrolle**  
**Nutzt Logging**  
**Baut Pattern ein**  
**Multithreading von piNaehern()**



# Bonus – TDD nutzen

Nach Robert C Martin gibt es drei Regeln des TDD:

1. „You are not allowed to write any production code unless it make failing unittest pass.“  
-> Du Brauchst einen fehlschlagenden Test um Code zu schreiben!
2. „You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failuers“  
-> Du darfst nicht mehr als einen fehlschlagenden Test schreiben!
3. „You are not allowed to write any more production code than is sufficent to pass the one failing unit test.“  
-> Schreib nur Code, der den einen Test erfüllt!

Siehe:

<https://www.youtube.com/watch?v=qkblc5WRn-U>

<https://youtu.be/58jGpV2Cg50?t=1300>

# Literatur

## Java:

- [1] <https://docs.oracle.com/en/java/javase/18/docs/api/index.html> - JavaDoc 18
- [2] <https://docs.oracle.com/javase/specs/jls/se18/jls18.pdf> - Java Language Specification
- [3] <https://docs.oracle.com/javase/specs/jvms/se18/jvms18.pdf> - Java VM Specification
- [4] <https://junit.org/junit5/docs/current/user-guide/> - Junit 5 User Guide

## Testen:

- [5] <https://www.german-testing-board.info/lehrplaene/istqbr-certified-tester-schema/entwicklungstester/> - Lehrplan des ISTQB
- [6] <https://ieeexplore.ieee.org/document/5399061> - Klassifikation von Software Anomalien

## Debugging:

- [7] <https://www.jetbrains.com/help/idea/debugging-code.html> - IntelliJ Debugger
- [8] [https://www.eclipse.org/community/eclipse\\_newsletter/2017/june/article1.php](https://www.eclipse.org/community/eclipse_newsletter/2017/june/article1.php) - Eclipse Debugger Einführung
- [9] <https://help.eclipse.org/latest/nav/80> - Debugging in Eclipse

