


DEBUGGING

Fehler in Programmen identifizieren und beheben

Struktur und Inhalt des Kurses wurden 2012 von Markus Heckner entwickelt. Im Anschluss haben Alexander Bazo und Christian Wolff Änderungen am Material vorgenommen. Die aktuellen Folien wurden von Alexander Bazo erstellt und können unter der **MIT-Lizenz**  verwendet werden.

AKTUELLER SEMESTERFORTSCHRITT (WOCHE 12)

| Kursabschnitt | Themen | | |
|-------------------------|------------------------------------|---|------------------------------|
| Grundlagen | Einführung | Einfache Programme erstellen | Variablen, Klassen & Objekte |
| | Kontrollstrukturen & Methoden | Arrays & komplexe Schleife | |
| Klassenmodellierung | Grundlagen der Klassenmodellierung | Vererbung & Sichtbarkeit | |
| Interaktive Anwendungen | Event-basierten Programmierung | String- & Textverarbeitung | |
| Datenstrukturen | Listen, Maps & die Collections | Speicherverwaltung | Umgang mit Dateien |
| Software Engineering | Debugging | Planhaftes Vorgehen bei der Softwareentwicklung | |
| | Qualitätsaspekte von Quellcode | | |

PINGO-QUIZ



<https://pingo.coactum.de/089521> 


DAS PROGRAMM FÜR HEUTE

- Qualitätsaspekte in der Programmierung
- Verschiedene Formen von Programmfehlern
- Methoden und Techniken zum Beheben von Programmfehlern

Hinweis: Vergessen Sie nicht die Klausuranmeldung in Flexnow vom 1. bis zum 9. Februar.

AUSBLICK AUF DIE NÄCHSTEN WOCHEN

| Datum | Inhalte | Übung |
|----------------|---|---------------|
| 21. Januar | Komplexere Datenstrukturen und Preisverleihung zur Weihnachts-Challenge | Übungsblatt |
| 28. Januar | Speicherverwaltung und Dateien | Übungsblatt |
| 04. Februar | Ausblick und Klausurvorbereitung | Klausurfragen |

Hinweis: Die Klausur findet am 18. Februar statt. Neben den Inhalten der Vorlesung und Übung können Sie sich auch mit den Klausurfragen vergangener Semester vorbereiten. Diese finden Sie auf [der Seite der Fachschaft](#) .

QUALITÄTSASPEKTE: FORMALE UND FUNKTIONALE KORREKTHEIT VON SOFTWARE

WAS MACH "GUTE" SOFTWARE AUS?

Software muss funktionieren, d.h. sie erfüllt die Erwartungen der NutzerInnen und allen anderen relevanten Personenkreisen (z.B. den EntwicklerInnen, AuftragsgeberInnen, ...). Diese gewünschten Eigenschaften werden Anforderungen genannt. Die zentralen Anforderungen lassen sich aufteilen in:

- Funktionale Anforderungen: *Was soll die Software tuen?*
- Nicht-Funktionale Anforderungen: *Wie soll sich die Software dabei verhalten?*

Daneben gibt es aber noch weitere Aspekte, die zur Qualität von Software beitragen.

NUTZERINNEN-PERSPEKTIVE: FEHLERFREIHEIT

Nutzerinnen und Nutzer nehmen die *fertige* Software war, die nach der Entwicklung durch das *Compilieren* erstellt und ausgeliefert wird.

NutzerInnen möchten vor allem, dass **die Software** ...

- ... fehlerfrei die Nutzung der angebotenen Funktionen erlaubt.
- ... einfach und intuitiv bedienbar ist (*Usability* bzw. Gebrauchstauglichkeit).
- ... hinsichtlich der Nutzung eine positive Erfahrung darstellt (*Use Experience*).

Hinweis: Der Weg vom Quellcode zu den NutzerInnen ist in der Regel weit aus komplexer und geht über das reine Compilieren hinaus. Software muss bereitgestellt, beschafft und eingerichtet werden.

PROGRAMMIERINNEN-PERSPEKTIVE: CODEQUALITÄT (1/3)

Programmiererinnen und Programmierer nehmen die *entstehende* Anwendung vor allem als Sammlung von Quellcodedateien war. Diesbezüglich stellen sie andere Anforderungen an die Software und erwarten unter anderem, dass der **Quellcode** ...

- ... verständlich formuliert ist.
- ... gut lesbar ist.
- ... logisch strukturiert und formuliert ist.
- ... einfach um weitere Funktionen ergänzt werden kann.

PROGRAMMIERINNEN-PERSPEKTIVE: CODEQUALITÄT (2/3)

Qualitativ hochwertiger Quellcode zeichnet sich u.a. dadurch aus, dass die folgenden Aspekte bei der Gestaltung berücksichtigt wurden.

- Guter Quellcode folgt den formalen *Best Practices* der gewählten Programmiersprache und selbstgewählten Auflagen des Entwicklungsteams (z.B. bei der Formatierung oder der Schreibweise von Bezeichnern).
- Guter Quellcode ist verständlich formuliert, in dem z.B. Bezeichner für Variablen oder Methoden treffend die jeweilige Funktion des Elements beschreiben.
- Guter Quellcode ist klar strukturiert. d.h., dass Teilaufgaben in abgeschlossenen Komponenten implementiert werden und z.B. wiederholt verwendeter Code zentral definiert wird (Vgl. *Duplicate Code*).

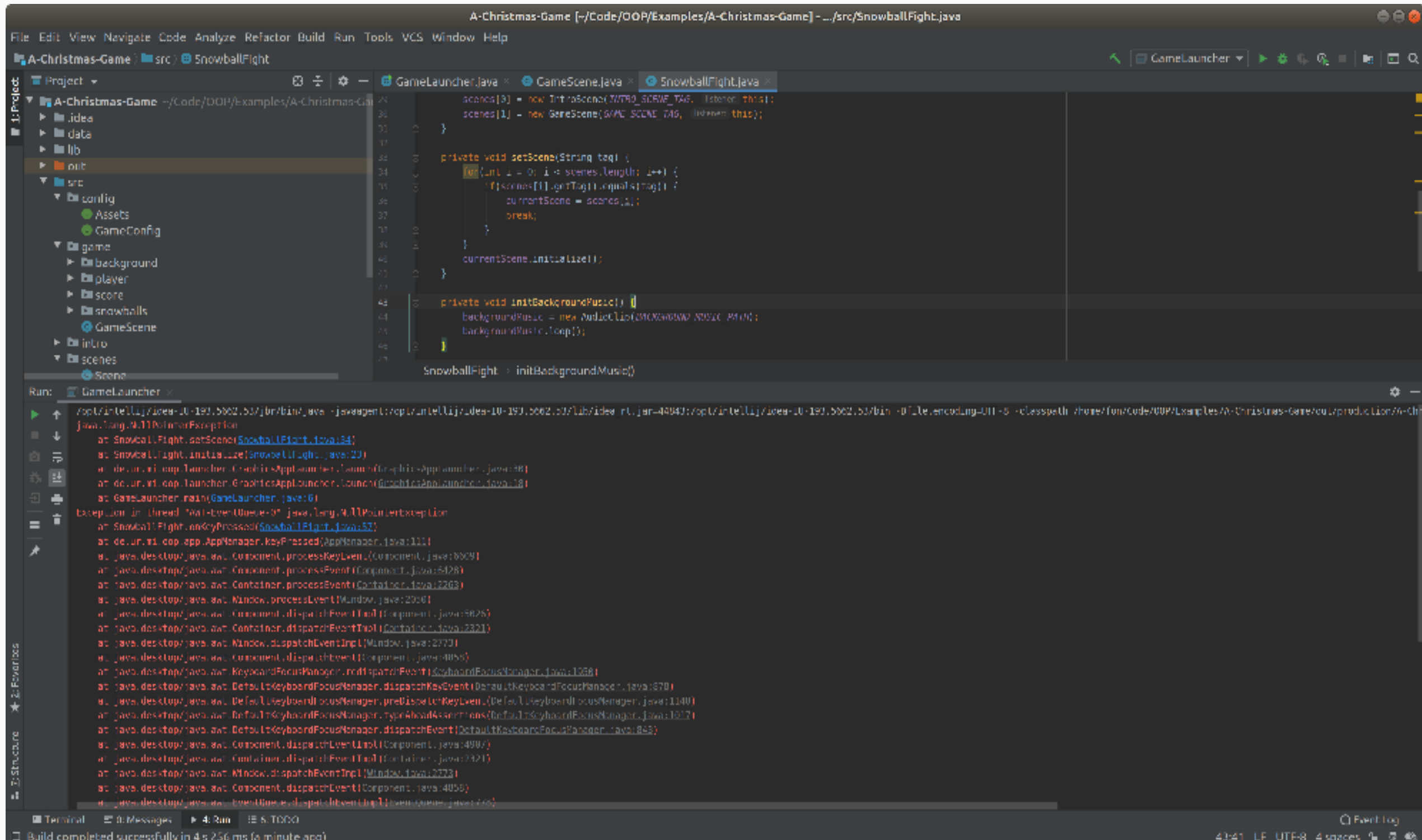
PROGRAMMIERINNEN-PERSPEKTIVE: CODEQUALITÄT (3/3)

- Guter Quellcode ist erweiterbar und folgt dazu einem (hierarchischen) Aufbau, der die Ergänzung weiterer Komponenten/Klassen ohne großer Änderungen am restlichen Code erlaubt.
- Guter Quellcode ist dokumentiert, d.h. Kommentare und andere Dokumente geben nützliche Informationen zur Verwendung und Bearbeitung.

Hinweis: In modernen Entwicklungsumgebungen, z.B. auch in IntelliJ IDEA, sind Hilfsfunktionen integriert, die Sie bei der Erstellung qualitativ hochwertigen Codes unterstützen. Dazu gehören u.a. automatische Formatierungshilfen, Fehlerfeedback, *Refactoring*-Werkzeuge oder Kommentarhilfen. Nutzern Sie diese Funktionen, **sobald Ihnen der Sinn hinter den jeweiligen Aktionen bewusst ist!**

PROGRAMMFEHLER IN JAVA

SYNTAKTISCH KORREKT, TROTZDEM VERBUGGT



The screenshot shows an IDE window for a project named "A-Christmas-Game". The file "SnowballFight.java" is open, showing a method `setScene` that iterates over an array of scenes. A syntax error is highlighted at line 57: `currentScene = scenes[i];`. The error message is `NullPointerException`. The stack trace below the code shows the exception being thrown from `SnowballFight.setScene` and propagating through various Java Swing components, including `AppManager`, `Component`, `Container`, `Window`, and `DefaultKeyboardFocusManager`.

```
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
A-Christmas-Game src / SnowballFight
Project
A-Christmas-Game
  .idea
  data
  lib
  out
  src
    config
      Assets
      GameConfig
    game
      background
      player
      score
      snowballs
      GameScene
    intro
    scenes
      Scene
GameLauncher.java
GameScene.java
SnowballFight.java
24 scenes[0] = new IntroScene(WTR9_SCENE_TAG, listener: null);
25 scenes[1] = new GameScene(GAME_SCENE_TAG, listener: this);
26 }
27
28 private void setScene(String tag) {
29     for(int i = 0; i < scenes.length; i++) {
30         if(scenes[i].getTag().equals(tag)) {
31             currentScene = scenes[i];
32             break;
33         }
34     }
35     currentScene.initialize();
36 }
37
38 private void initBackgroundMusic() {
39     backgroundMusic = new AudioClip(ASSETS_PATH + "music.mp3");
40     backgroundMusic.loop();
41 }
42
43 SnowballFight -> initBackgroundMusic()
Run: GameLauncher
/opt/intellij/idea-10-190.5002.50/bin/java -javaagent:/opt/intellij/idea-10-190.5002.50/lib/idea_rt.jar=40040:/opt/intellij/idea-10-190.5002.50/bin -Dfile.encoding=UTF-8 -classpath /home/fon/Code/OOP/Examples/A-Christmas-Game/out/production/A-Christmas-Game
java.lang.NullPointerException
    at SnowballFight.setScene(SnowballFight.java:57)
    at SnowballFight.initialize(SnowballFight.java:23)
    at de.ur.wi.dop.app.Launcher.ChrisAppLauncher.LauncherAppLauncher.java:80
    at de.ur.wi.dop.app.Launcher.GraphicsAppLauncher.LauncherAppLauncher.java:18
    at GameLauncher.main(GameLauncher.java:6)
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException
    at SnowballFight.onKeyPressed(SnowballFight.java:57)
    at de.ur.wi.dop.app.AppManager.onKeyPressed(AppManager.java:111)
    at java.desktop/java.awt.Component.processKeyEvent(Component.java:669)
    at java.desktop/java.awt.Component.processEvent(Component.java:628)
    at java.desktop/java.awt.Container.processEvent(Container.java:226)
    at java.desktop/java.awt.Window.dispatchEvent(Window.java:230)
    at java.desktop/java.awt.Component.dispatchEventImpl(Component.java:482)
    at java.desktop/java.awt.Container.dispatchEventImpl(Container.java:232)
    at java.desktop/java.awt.Window.dispatchEventImpl(Window.java:272)
    at java.desktop/java.awt.Component.dispatchEvent(Component.java:405)
    at java.desktop/java.awt.KeyboardFocusManager.dispatchEventImpl(KeyboardFocusManager.java:125)
    at java.desktop/java.awt.DefaultKeyboardFocusManager.dispatchEvent(DefaultKeyboardFocusManager.java:87)
    at java.desktop/java.awt.DefaultKeyboardFocusManager.preDispatchKeyEvent(DefaultKeyboardFocusManager.java:1140)
    at java.desktop/java.awt.DefaultKeyboardFocusManager.typeAheadAssertions(DefaultKeyboardFocusManager.java:1017)
    at java.desktop/java.awt.DefaultKeyboardFocusManager.dispatchEvent(DefaultKeyboardFocusManager.java:853)
    at java.desktop/java.awt.Component.dispatchEventImpl(Component.java:490)
    at java.desktop/java.awt.Container.dispatchEventImpl(Container.java:232)
    at java.desktop/java.awt.Window.dispatchEventImpl(Window.java:272)
    at java.desktop/java.awt.Component.dispatchEvent(Component.java:405)
    at java.desktop/java.awt.EventQueue.dispatchEventImpl(EventQueue.java:67)
Build completed successfully in 4 s 256 ms (a minute ago)
4:41 LF UTF-8 4 spaces
```


PROGRAMMKOMPLEXITÄT UND FEHLERHÄUFIGKEIT

Mit steigender Komplexität der entwickelten Programme steigt auch die *Chance* auf Programmierfehler:

- Mehr Code bietet mehr Raum für Fehler: in der Softwareindustrie wird häufig mit Fehlerraten von 1-5 *Bugs* pro 1000 *Lines of Code* gerechnet.
- Aufwendigere Programme, z.B. interaktive Anwendungen, bieten eine Vielzahl an unterschiedlichen Möglichkeiten für den Programmablauf an.
- Komplexere Software hat mehr mögliche Fehlerquellen (Eingabe, Ausgabe, Verarbeitung, Ablauf, NutzerInnen(!), ...).

Die hier genannten Gründe lassen sich beliebig erweitern. Die Kernaussage ist: **Bugs lassen sich niemals vollständig vermeiden. Wir brauchen Strategien, um damit umzugehen.**

BUGS TRETEN ZUR LAUFZEIT AUF

Durch genaues Lesen des Quellcodes können Bugs auch im Vorfeld identifiziert werden. In der Regel wird man aber erst durch auftretende Fehler oder ein bemerktes Fehlverhalten des Programms auf sie aufmerksam:

Exception, Fehler oder Absturz: *Das Programm verursacht an einer bestimmten Stelle einen Fehler, der die weitere Ausführung verhindert.*

Funktionales Fehlverhalten: *Das Programm verhält sich unerwartet.*

EXCEPTIONS IN JAVA

Bugs stellen unerwartet Fehler im Programmablauf dar. Java kommuniziert bestimmte Formen dieser Probleme über Exceptions (engl. *Ausnahme*)

- *Exceptions* werden ausgelöst, wenn der eigentlich vorgesehene Ausdruck oder Befehl zur Laufzeit nicht ausgeführt werden kann.
- Die betroffene Stelle *wirft (throws)* in diesem Fall eine *Exception*, die innerhalb der Anwendung als Objekt kommuniziert wird.
- Das Exception-Objekt und dessen Subklassen beinhalten Informationen zum Fehler und sorgen u.A. für die rote Ausgabe in der IntelliJ-Konsole.

Hinweis: Exceptions beschreiben erwartbare, mögliche Fehlersituationen. Einige davon lassen sich schon zur Laufzeit mit einer gewissen Wahrscheinlichkeit vorhersehen und erfordern eine entsprechende Vorbereitung des Codes. Dazu und zum Entwurf eigener *Exceptions* hören wir im weiteren Verlauf der Vorlesung noch mehr.

EINIGE EXCEPTIONS UND DEREN MÖGLICHE URSACHE

| Exception | Beschreibung | Mögliche Ursache* |
|---------------------------|--|---|
| IndexOutOfBoundsException | Zugriff auf eine Element außerhalb des Bereichs der validen Array-Indizes. | Fehler im iterativen Zugriff auf ein Array. |
| NullPointerException | Zugriff auf eine Variable, die kein Objekt beinhaltet. | Fehlende Initialisierung eines Objekts |
| Stackoverflow | Überlauf des Speicherbereichs, der temporärer Parameter und lokale Inhalte von Methoden verwaltet. | Endlosschleife mit Methodenaufrufen |
| ClassCast | Versuch, ein Objekt als Instanz einer inkompatiblen Klassen zu betrachten. | Inkompatible Klassen, z.B. falsche Elternklasse |
| FileNotFoundException | Versuch, auf eine nicht existierende Datei zuzugreifen. | Schreibfehler im Dateiname |

BEISPIELE FÜR FEHLVERHALTEN IN PROGRAMMEN

| Fehlverhalten | Mögliche Ursache |
|--|--|
| Elemente werden nicht gezeichnet | Die draw-Methode wurde nicht aufgerufen. |
| Die Position von einem graphischen Objekt stimmt nicht | Im Konstruktor wurden falsche Werte verwendet oder berechnet. |
| Das Ergebnis einer mathematischen Berechnung ist "falsch". | Durch die gewählten Datentypen z.B. <code>int</code> entstehen "Rundungsfehler". |

DEBUGGEN: SYSTEMATISCHE IDENTIFIKATION UND BEHEBUNG VON FEHLERN

DEBUGGEN

Debugging is the process of finding and resolving defects or problems within a computer program that prevent correct operation of computer software or a system. Debugging tactics can involve interactive debugging, control flow analysis, unit testing, integration testing, log file analysis, monitoring at the application or system level, memory dumps, and profiling.

Quelle: <https://en.wikipedia.org/wiki/Debugging>

WARUM ÜBERHAUPT DEBUGGEN?

Als ProgrammierInnen müssen Sie lauffähige, benutzbare Software entwickeln (Bugs verhindern die intendierte Nutzung Ihrer Programme). Die Motivation dafür kann unterschiedlich sein:

- Kommerzieller Profit
- Gute Noten in der Ausbildung
- Validität, z.b. bei Software für wissenschaftliche Experimente
- Verantwortung, z.B. bei Sicherheitssystemen oder Medizinprodukten
- Intrinsisches Bedürfnisse, *gute* Software zu schreiben

Hinweis: Je später ein Fehler entdeckt wird, desto teurer (aufwendiger) ist es, ihn zu beheben. Mit jedem Bug wird es schwieriger, den restlichen Teil

FEHLER FINDEN

- Meist machen Sie (und alle anderen ProgrammiererInnen) kleine Fehler, die Sie häufig ohne größere Probleme selbst lösen können bzw. könnten.
- Schwieriger als die eigentliche Lösung des Fehlers ist das Finden der entsprechenden Stelle im Code.
- Wir werfen heute einen Blick auf unterschiedliche Strategien zum Finden von Fehlern an: **Analyse von Exceptions und dem StackTrace, Printlining** und den **in IntelliJ eingebauten Debugger**.

ALLGEMEINE STRATEGIEN (1/2)

Denken Sie daran, dass Bugs in der Regel sehr einfachen Ursachen haben: ein falsch platziertes Semikolon, die Verwendung der falschen Variable, ein vergessener Methodenaufruf, ...

```
1  if(j > k);  
2  // Durch das Semikolon am Ende der if-Bedingung gehört der nachfolgende Block nicht mehr  
3  // zur Programmflusssteuerung und wird IMMER ausgeführt.  
4  {  
5      System.out.println("j ist größer als k");  
6  }
```

```
1  Rectangle rect;  
2  
3  public void init() {  
4      // initRect();  
5  }  
6  private void initRect() {  
7      rect = new Rect(0,0,100,100,Colors.RED);  
8  }  
9  public void draw() {  
10     rect.draw(); // NullPointerException weil rect nicht initialisiert wurde!
```


ALLGEMEINE STRATEGIEN (2/2)

- Gehen Sie systematisch beim Identifizieren des Fehlers vor: Ein Bug wird immer durch dieselbe(n) Stelle(n) im Code verursacht.
- Beheben Sie unterschiedliche Bugs nacheinander.
- Ändern Sie stets nur eine Stelle Ihres Codes und testen Sie die Änderungen.
- Seien Sie kritisch gegenüber dem eigenen Code.

**Bugs zu finden ist oft schwierig, aber nie unmöglich: Keine Panik!
Keine Frust!**

DER IDEALFALL: DIE EXCEPTION MIT STACKTRACE

```
1 java.lang.NullPointerException
2     at SnowballFight.setScene(SnowballFight.java:34)
3     at SnowballFight.initialize(SnowballFight.java:23)
4     at de.ur.mi.oop.launcher.GraphicsAppLauncher.launch(GraphicsAppLauncher.java:30)
5     at de.ur.mi.oop.launcher.GraphicsAppLauncher.launch(GraphicsAppLauncher.java:18)
6     at GameLauncher.main(GameLauncher.java:6)
```

Der StackTrace zeigt die Hierarchie der Methodenaufrufe an, die zum Fehler geführt haben. Die Ausgabe ist von unten nach oben zu lesen. Wenn Sie ganz oben einen Verweis in Ihren eigenen Code finden, haben Sie Glück und können die entsprechende Stelle untersuchen/fixen.

PRINTLINING (1/2)

- Die meisten Programmiersprachen erlauben es Ihnen, Ausgaben auf der Standardausgabe des Systems zu erzeugen.
- Meist landen diese Ausgaben in einer Textkonsole, sind also bei graphischen Anwendungen nicht sichtbar.
- IntelliJ verfügt über eine eigene Konsole, in der die Ausgaben zur Laufzeit angezeigt werden.
- In Java kann zu jedem Zeitpunkt mit der Methode `System.out.println` eine Ausgabe erzeugt werden. Dazu wird der Methode ein String-Objekt als Parameter übergeben.

PRINTLINIG (2/2)

Printlining bezeichnet das systematische Einbauen von Konsolenausgaben in den Quellcode um den Programmablauf und Zwischenstände zu prüfen bzw. nachzuvollziehen. Mit Hilfe des *Printlining* können Sie u.a. die folgenden Fragen beantworten:

- In welcher Reihenfolge werden Methoden tatsächlich aufgerufen?
- Welchen Wert hat eine Variable zu einem bestimmten Zeitpunkt?
- Wird eine bestimmte Stelle des Quellcodes überhaupt ausgeführt?

PRINTLINING: PROGRAMMABLAUF KONTROLLIEREN

```
1 public void doStuff() {  
2     println("in: doStuff()");  
3     doMoreStuff();  
4     println("finished: doStuff()");  
5 }  
6  
7 private void doMoreStuff() {  
8     println("in: doMoreStuff");  
9 }
```

Hinweis: Nutzen Sie sinnvolle Ausgaben, z.B. den Methodennamen. Erleichtern Sie durch zusätzliche Informationen die Interpretation der ausgegebene Inhalte. Die Ausgabe der Strings Test oder dasdas dsadasd ist nicht sinnvoll!

PRINTLINING: WERTE KONTROLLIEREN

```
1 @Override
2 public String toString() {
3     return "Circle [color=" + getColor() + "]";
4 }
5
6 Circle circle = new Circle(this, 50, 50, 50, BLACK);
7 println(circle);
8
9 // Ausgabe:
10 // Circle [getColor()=de.ur.mi.graphics.Color[r=0,g=0,b=0]]
```

EINE EINFACHE LOG-KLASSE (1/2)

Printlining oder *Logging* funktioniert am besten, wenn die folgenden Punkte beachtet werden:

- Ausgaben sollten nur dann erzeugt werden, wenn sie auch benötigt werden, z.B. nicht im fertigen Produkt!
- Neben dem Inhalt sollten immer auch zusätzliche Informationen wie z.B. Zeitpunkt oder Kontext ausgegeben werden.

Hinweis: Auch in kleineren Projekten bietet sich die Entwicklung oder Nutzung einer separaten Logger-Klasse an, die die *Printlining*-Aufgabe übernimmt.

EINE EINFACHE LOG-KLASSE (2/2)

```
1 public class Logger {  
2     private static boolean isEnabled = false;  
3  
4     public static void enable() {  
5         isEnabled = true;  
6     }  
7     public static void disable() {  
8         isEnabled = false;  
9     }  
10    public static void log(String msg, Object context) {  
11        if(!isEnabled) { return; }  
12        Date now = new Date();  
13        StringBuilder builder = new StringBuilder();  
14        Date now = new Date();  
15        builder.append(":\t");  
16        builder.append(msg);  
17        builder.append(" from: ");  
18        builder.append(context.getClass());  
19        System.out.println(builder.toString());  
20    }  
21 }
```


VERWENDUNG DER LOG-KLASSE

Einschalten und Loggen

```
1 public class App extends GraphicsApp {  
2  
3     public void initialize() {  
4         Logger.enable();  
5         Logger.log("in: initialize", this);  
6     }  
7  
8 }
```

Ausgabe

```
1 Tue Jan 14 13:37:00 CET 2020:    in: initialize from: class App
```

SCHRITTWEISER ABLAUF VON PROGRAMMEN (1/2)

Im Zweifel finden Sie einen Fehler nur dann, wenn Sie den Programmablauf Schritt für Schritt durchgehen.

Solange wir nicht nebenläufig programmieren, werden alle Anweisungen in unseren Programmen sequenziell ausgeführt. D.h. es gibt eine klare Abfolge an Befehlen und irgendwo in dieser Kette versteckt sich der Fehler.

SCHRITTWEISER ABLAUF VON PROGRAMMEN (1/2)

```
1 public void doStuff() {  
2     doMoreStuff();  
3 }  
4  
5 public void doMoreStuff() {  
6     doEvenMoreStuff();  
7     doEvenMoreStuff();  
8 }  
9  
10 public void doEvenMoreStuff() {  
11     //  
12 }
```

Beim Methodenaufwurf gilt: Eine Anweisung wird komplett abgearbeitet, bevor mit der nächsten Anweisung (oder dem nächsten Methodenaufwurf) begonnen wird. **Diese schrittweise Verarbeitung geschieht in der Regel sehr schnell, kann durch den Debugger aber in Einzelschritten nachvollzogen werden.**

DEBUGGER

Ein Debugger (von engl. de- (Präfix; dt. ent-, aus-) im Sinne von entfernen und engl. bug im Sinne von Programmfehler) ist ein Werkzeug zum Diagnostizieren und Auffinden von Fehlern in Computersystemen, dabei vor allem in Programmen, aber auch in der für die Ausführung benötigten Hardware. Debugging bezeichnet die Tätigkeit, solche Fehler zu diagnostizieren und aufzufinden, sei es unter Verwendung eines Debuggers oder anderer Methoden.

Quelle: <https://de.wikipedia.org/wiki/Debugger>

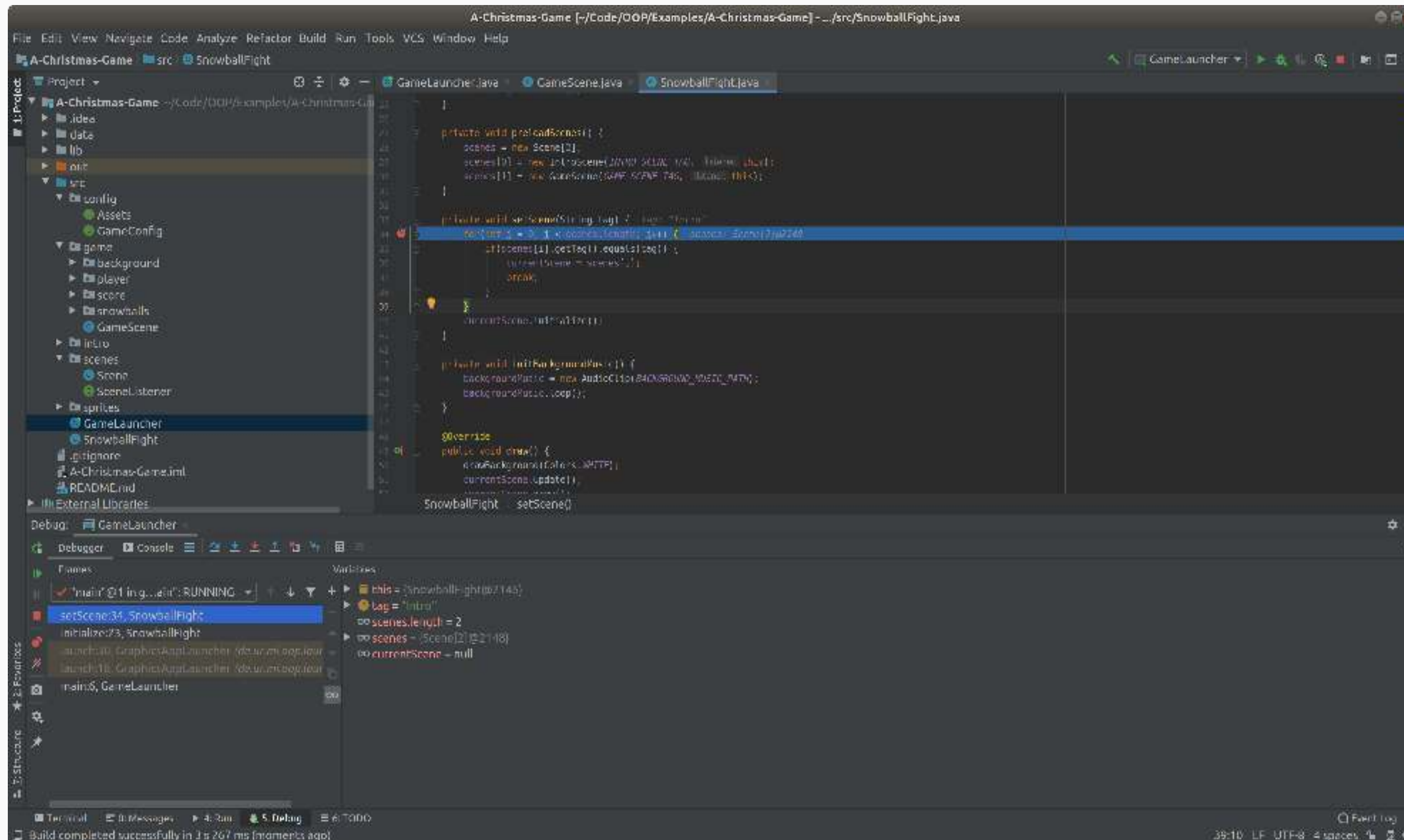
DER DEBUGGER IN INTELIJ

- IntelliJ beinhaltet einen Debugger, der die schrittweise (kontrollierte) Ausführung eines Programms erlaubt.
- Sie können den Programmablauf damit selbst steuern, also von Anweisung zu Anweisung springen.
- Für jeden Schritt können Sie mit diesem Tool den aktuellen Zustand der Anwendung (Variablen und deren Werte) einsehen.

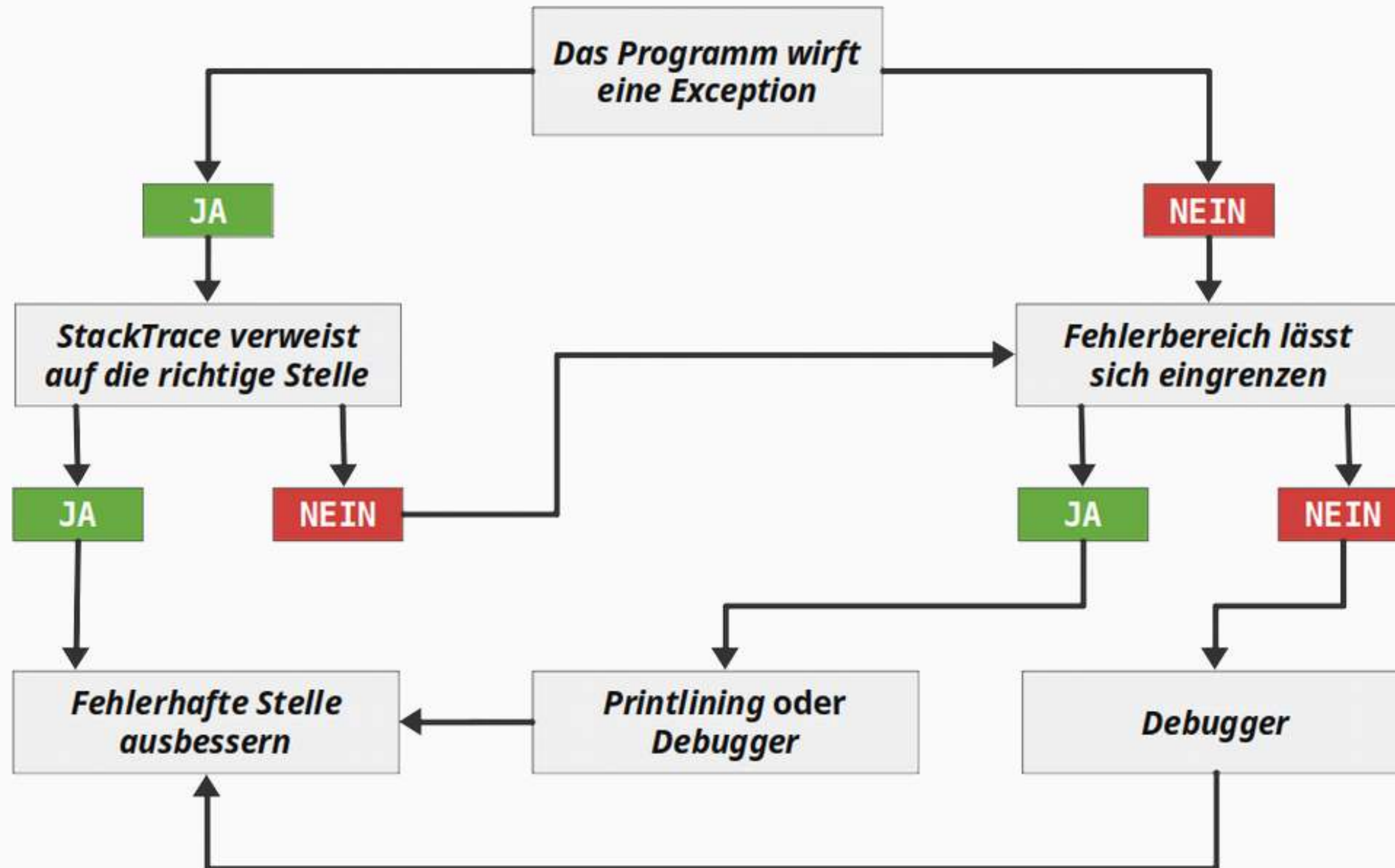
VERWENDUNG DES DEBUGGERS (1/2)

1. Die Anwendung muss explizit über den Debugger gestartet werden (Kontextmenü oder Symbol rechts oben).
2. Um in die eigentliche Debugger-Ansicht zu wechseln, muss die Anwendung pausiert werden.
3. Alternativ können im Code *Breakpoints* gesetzt werden, an denen der Debugger die Ausführung automatisch pausiert.
4. Im pausierten Zustand kann der Debugger über die entsprechenden UI-Elemente gesteuert werden.

VERWENDUNG DES DEBUGGERS (2/2)



ÜBERSICHT: FEHLER IDENTIFIZIEREN



ZUSAMMENFASSUNG

- Softwarequalität zeigt sich in unterschiedlichen Bereichen, die funktionale Vollständigkeit ist nur ein Teil von guten Programmen.
- Hohe Codequalität ist ein wichtiger Faktor der Softwarequalität und wirkt sich in der Regel auch positiv auf die Fehlerrate aus.
- *Bugs* lassen sich nicht vollständig vermeiden: Wir benötigen Strategien zum Identifizieren und Beheben dieser Probleme.
- *Printlining* und der *Debugger* sind zwei Strategien zum systematischen Beheben von Programmfehlern.
- Moderne Entwicklungsumgebungen unterstützen sich beim Beheben von Fehlern und beim Steigern der Codequalität.

JETZT: LIVE-TEST DES DEBUGGERS

VIELEN DANK FÜR IHRE AUFMERKSAMKEIT.

Fragen oder Probleme? In allgemeinen Angelegenheiten und bei Fragen zur Vorlesung wenden Sie sich bitte an Alexander Bazo (alexander.bazo@ur.de 🔗). Bei organisatorischen Fragen zu den Studienleistungen und Übungsgruppen schreiben Sie bitte Florin Schwappach (florin.schwappach@ur.de). Bei inhaltlichen Fragen zu den Übungen, Beispielen und Studienleistungen schreiben Sie uns unter mi.oop@mailman.uni-regensburg.de 🔗.

QUELLEN

Eric S. Roberts, *The art and science of Java: an introduction to computer science, New international Edition*, 1. Ausgabe, Pearson, Harlow, UK, 2014.