<Beutiful Code Workshop>
<Error Handling>

Beautiful Code WiSe 2024/25 Richard Reh

Agenda

1. Workshopinhalt (45-60 min)
    - Relevanz
    - Prinzipien defensiver Programmierung
    - Techniken zur Vermeidung von Errors
    - Techniken zum Umgang mit Errors
    - Error Handling Tools

2. Übungsaufgaben (30 min)

3. Diskurs zum Thema (20 – 30 min)

Relevanz des Error Handlings

„Nearly zero provably error-free computer programs have ever been written”

- David A. W. Soergel

Die ungefähre Fehlerrate von industriellen Programmierfehlern liege bei ungefähr 15 – 50 Fehler pro 1000 Zeilen Code.

2022 wurde für die USA der Verlust durch Programmierfehler auf 2,41 Billionen Dollar geschätzt.

Gefahr eines Dominoeffekts, der die negativen Folgen von früh entstandenen Fehler immer weiter verschlimmert.

Relevanz des Error Handlings

Behandlung von Errors ist ein unvermeidbarer Bestandteil in jedem Programm.

Unzureichendes Error Handling führt häufig erst spät zu Problemen.

→ Meistens dann, wenn das Programm bereits in einer Produktionsumgebung mit realen Daten läuft.

Wichtig ist:

Frühzeitige Erkennung und Behandlung von Fehlerquellen.

Gute Fehlermeldungen zu erstellen, für eine präzise Problemsuche und leichtes Debugging.

Trotz Fehlerbehandlungs-Maßnahmen, gute Lesbarkeit und Verständlichkeit von Code.

Einfach gesagt: unzureichendes Error Handling führt zu ehrblichem Verlust von:

→ Zeit → Geld → Energie

Relevanz des Error Handlings

**“Clean code is readable, but it must also be robust.
These are not conflicting goals”.** - Robert C. Martin

Defensive Programmierung - Prinzipien

Ein Prorammierstil, bei dem...

→ Proaktives Vordenken erfordert wird, in der Erwartung, dass Error aufkommen können.

→ Programme so gestalten werden, dass sie abnormale Daten und unerwartete Zustände frühzeitig erkennen und entsprechend reagieren können.

Defensive Programmierung - Prinzipien

Alternativ:

"Programme implementieren, die mit verschiedenen Techniken wie Plausibilitätsprüfungen, Argumentvalidierungen, Typkontrollen und Überwachung von abweichenden Daten abnormale Parameter erkennen und in angemessener Weise auf solche Fehler reagieren können."

Defensive Programmierung - Prinzipien

Three rules of defensive programming:

1. Never assume anything.
2. All input must be validated against a set of all legal inputs.
3. Then determine the action you will take if the input is incorrect.

-Arnold S. Berger, PhD

Techniken zum Vermeiden von Errors

Simpler ,,null-check“

Einfache Abfrage, ob Parameter ,,null“ sind.

```
public void readFruit(){
if(name != null && calories != 0){
        System.out.println("Die Frucht "+name+ " hat "+calories+ " Kalorien");
    }else {
        throw new IllegalArgumentException("Ungültige Fruchtparameter");
    }
}
```

Jedoch weniger schön anwendbar, bei komplexeren Problemen.

Vermeidung von „null“ als Parameter

Alternative: Defaultwerte als Parameter statt null verwenden:

```
class DefaultFruit {
   private String name = "";
   private int calories = 0;

   public int getCalories() {
      return calories;
   }
   public void setCalories(int calories){
      this.calories = calories;
   }

   public String getName() {
      return name;
   }
   public void setName(String name){
      this.name = name;
   }

   public void readFruit(){
      System.out.println("Die Frucht "+name+ " hat "+calories+ "Kalorien");
   }
}
```

In der Main Methode:

```
DefaultFruit banana = new DefaultFruit();
```

Vermeidung von „null“ als Parameter

Alternative: Stille Objekte verwenden:

```
interface Animal {
    void makeSound();
}

class Dog implements Animal {
    public void makeSound() {
        System.out.println("Woof");
    }
}

class NullAnimal implements Animal {
    public void makeSound() {
        // Macht nichts --> Gegen NullPointerExceptions
        // Expliziter Kommentar zum Erklären für andere Entwickler, dass hierbei nichts passieren soll.
    }
}
```

Vermeidung von „null“ als Parameter

Alternative: nullObject-Designpattern

```
interface Discount {
    double apply(double price);
}

class PercentageDiscount implements Discount {
    private double percentage;

    public PercentageDiscount(double percentage) {
        this.percentage = percentage;
    }

    @Override
    public double apply(double price) {

        return price - (price * percentage);
    }
}

// Null Object für Discount
class NoDiscount implements Discount {
    @Override
    public double apply(double price) {
        return price; // Keine Veränderung des Preises
    }
}
```

```
class Order {
    private double price;
    private Discount discount;

    public Order(double price, Discount discount) {
        this.price = price;
        this.discount = discount != null ? discount : new NoDiscount(); // Null Object statt
null
    }

    public double getFinalPrice() {
        return discount.apply(price);
    }
}
```

In der Main Methode:

```
NoDiscount notDiscounted = new NoDiscount();
PercentageDiscount twentyPercent = new PercentageDiscount(0.20);

Order order = new Order(2.99,notDiscounted); // --> 2,99
Order orderWithDiscount = new Order(9.99, twentyPercent); // --> 7.99
```

Vermeidung von „null“ als Parameter

Zweck:

→ Programmabsturz durch ungültige Parameter vermeiden.

→ Keine Anomalien, da die leeren Parameter keine Auwirkungen auf Daten haben.

→ Gute Alternative zu einfachen „null“-Parametern, um die Absicht von null Parametern zu nutzen und dabei keine negativen Auswirkungen zu riskieren.

,,Fail Fast“-Prinzip

Auch bekannt als: ,,instant fail“-, ,,fail fast“-, ,,fail earyl“- Prinzip.

Ansatz, bei dem Fehler bspw. in Methodenaufrufen frühzeitig aufgefangen werden.

Bei nicht erfüllten Voraussetzungen führt die Methode ihre Operation nicht mehr weiter aus (fail).

→Methode schlägt fehl und es wird eine Fehlermeldung ausgeworfen oder ein Fehler wird zurückgegeben.

Einfacheres Debugging → Vermeidung von Folgefehlern und späterem manuellem ,,backtracing“.

Simpler „null-check“

Simple null-check sind ebenfalls eine Art die Operation frühzeitig zu beenden.

```
public void readFruit(){
   //Einfacher null-check als Fail fast Bedingung
   if(name != null && calories != 0){
      System.out.println("Die Frucht "+name+ " hat "+calories+ " Kalorien");
   }else {
      throw new IllegalArgumentException("Ungültige Fruchtparameter");
   }
```

Jedoch weniger schön anwendbar, bei komplexeren Problemen.

ArgumentNullValidator

Einsparung von mehreren Zeilen zur Abfrage der Validität von Parametern.

→ Simplere, kürzere und lesbarere null-checks.

→ Reduziert das Risiko von einer ,,NullPointerException“

→ Eigentliche Abfrage wird ,,ausgelagert“.

→ Vermeidung von Redundanz derselben null-Abfragen, die jeweils mehrere Zeilen brauchen.

ArgumentNullValidator

ArgumentNullValidator Klasse:

```
public class ArgumentNullValidator {
    public static <T> T validateNotNull(T argument, String argumentName) {
        if (argument == null) {
            throw new IllegalArgumentException(argumentName + " darf nicht null sein.");
        }
        return argument;
    }
}
```

Beispiel ausführlicher in der IDE vorzeigen:

Optinals in Java

Ähnliches Ziel: Einsparung von mehreren Zeilen zur Abfrage der Validität von Parametern.

→ Reduziert das Risiko von einer ,,NullPointerException“
→ Bessere Lesbarkeit des Codes
→ Reduziert Redundanzen / Codedopplungen für gleiche null-checks.

Allerdings nur Java-spezifisch.

Optinals in Java

Anwendungsbeispiele von Optionals:

```java
// Beispiel 1: Wert vorhanden
Optional<String> optionalName = Optional.of("Alice");
optionalName.ifPresent(name -> System.out.println("Hallo, " + name + "!"));

// Beispiel 2: Kein Wert vorhanden
Optional<String> emptyOptional = Optional.empty();
System.out.println(emptyOptional.orElse("Kein Name angegeben"));

// Beispiel 3: Optional mit Standardwert
String name = optionalName.orElse("Unbekannt");
System.out.println("Name: " + name);
```

of(value) → Erstellt ein Optional mit einem Wert
empty() → Erstellt ein Optional mit einem Wert
orElse(defaultValue) → Gibt einen Standardwert zurücIk, falls leer
ifPresent → Führt eine Aktion aus, wenn ein Wert vorhanden ist

Ausgabe:

```
Hallo, Alice!
Kein Name angegeben
Name: Alice
```

Null-safety Beispiele aus anderen Sprachen

C#:

Nullable Reference Types:

```
string nonNullable = "Hello"; // Kann nicht null sein
string? nullable = null;      // Kann null sein
```

→ Compiler zwingt bei Zugriff auf Variablen auf null-Abfrage

Null Coalescing Operator (??) & Assignment (??=):

```
string? name = null;

// Null-Coalescing Operator
string displayName = name ?? "Default Name"; // "Default Name", wenn name null ist

// Null-Coalescing Assignment
name ??= "Assigned Name"; // Weist einen Wert zu, falls name null ist
Console.WriteLine(name); // "Assigned Name"
```

→ Erleichtern die Arbeit und erhöhen die Sicherheit mit potentiellen Null-Werten

Null-safety Beispiele aus anderen Sprachen

C#:

Null-Conditional Operator:

```
string? name = null;
int? length = name?.Length; // Gibt null zurück, wenn name null ist
Console.WriteLine(length ?? 0); // "0"
```

→ Compiler zwingt bei Zugriff auf Variablen auf null-Abfrage

JavaScript:

Null Coalescing Operator

```
function getOptionalValue() {
    return null; // Kein Wert vorhanden
}

const value = getOptionalValue() ?? "Default Value";
console.log("Value:", value); // Default Value
```

→ Prinzip und Schreibweise ähnlich wie in C#

Null-safety Beispiele aus anderen Sprachen

JavaScript:

Promises:

```
function fetchData() {
    return new Promise((resolve, reject) => {
        // Simulierte Bedingung
        const data = null;
        if (data) {
            resolve(data);
        } else {
            reject(new Error("No data found"));
        }
    });
}

fetchData()
    .then(data => console.log("Data:", data))
    .catch(error => console.error("Error:", error.message));
```

→ Promise verspricht ein zukünftigen Rückgabewert und sichert bei Fehlschlag einen alternativen Programmablauf

Paradigmen und Strategien

Paradigma Error Codes:

Nummerische Rückgabecodes, die den Ausgang eines Programm(teils) mitteilen.

Bspw. C nutzt Codes wie: 0, -1 oder –ENOENT, da in C per se keine Exceptions definiert sind.

Vorteile:

→ Effizienz, kein zusätzlicher Overload

Nachteile:

→ Fehlender Kontext

→ Weniger Lesbarkeit

Paradigmen und Strategien

Paradigma Result Types:

Methoden geben ein Objekt vom ,,Result-Typ“ zurück, der den Erfolg oder Fehler der Methode einkapselt.

Der Erfolg oder Fehler des Result-Objektes kann dynamisch ermittelt werden und darauf basierend die nächste Operation ablaufen.

Vorteile:

→Alternative zu Exceptions, weniger Overhead

→ Typsicherheit, durch Signatur der Methoden

→ Explizites Error Handling erzwungen

Nachteile:

→ Potentielle Gefahr für Boilerplate-Code durch ständige Prüfungen nach dem Ergebnis des Results

Assertions

Strategie Assertions:

Tests von Entwicklern, die Annahmen über Bedingungen und States von Programmstellen während der Laufzeit prüfen.

→ Sicherstellen eines korrekten Programmablaufs.

→ Typischerweise in Objektorientierten Sprachen verwendet wie Java, Python, C++ etc.

→ NICHT zum Error Handling gedacht, nur zum Prüfen von Zuständen.

Vorteil:

→ Erlaubt es Fehler während der Entwicklung besser zu erkennen.

(Nachteil ?):

→ Liefert kein Feedback bei Fehlverhalten für Benutzer, nur für Entwickler relevant

Paradigmen und Strategien

Strategie Log-basierte Fehlerbehandlung:

Egal ob mit einfachen Logs oder Logger Frameworks, Fehler können so protokolliert und genauer analysiert werden.

Überwachung des Systemverhaltens während der Laufzeit.

Vorteile:

→Ermöglicht es Enwticklern spätere Fehlerdiagnosen durchzuführen.

Nachteile:

→ Kann bei zu vielen Logs die Sauberkeit / Lesbarkeit des Codes etwas einschränken.

Making impossible state impossible

Grundgedanke:

Datenstrukturen, Klassen und Funktionen so gestalten, dass logisch unmögliche bzw. unerwünschte Zustände verhindert werden.

Keine feste Technik, kann auf unterschiedliche Weise umgesetzt werden.

→ Dadurch werden Fehler schon zur Compilezeit statt zur Laufzeit verhindert.

Vorteil: Es werden insgesamt weniger ,,extra“-Sicherheitsabfragen zur Überprüfung der Validität von Objekten benötigt, ob ihr Zustand zu jedem Zugriffszeitpunkt überhaupt logisch gesehen korrekt ist.

→ Dadurch weniger Fehlerbehandlungen nötig und Erzielung einer besseren Code-Lesbarkeit.

Making impossible state impossible

Verwendung von Invarianten zur Vermeidung von unmöglichen Zuständen:

Invariante: Eine Bedingung, die an bestimmten Stellen im Code immer wahr sein muss, unabhängig vom Zustand des Programms.

→ Helfen dabei, Korrektheit und Konsistenz innerhalb von Programmen sicherzustellen.

→ Schützen vor (sinnhaft) ungültigen Zuständen im Programm.

Making impossible state impossible

Klasseninvariante

```
public class BankAccount {
   private double balance;

   public BankAccount(double initialBalance) {
      if (initialBalance < 0) {
         throw new IllegalArgumentException(„Initial balance cannot be negative.");
      }
      this.balance = initialBalance;
   }
...
```

→ balance >= 0 ist zum Zeitpunkt der Instanziierung immer erfüllt

Dateninvariante

```
public void push(T element) {
   if (stack.size() >= maxCapacity) {
      throw new IllegalStateException("Stack capacity exceeded.");
   }
   stack.push(element);
}
```

→ stack.size() ist zu jedem Zeitpunkt < maxCapacity

Schleifeninvariante

```
public static int calculateSum(int[] array) {
   int sum = 0;
   for (int i = 0; i < array.length; i++) {
      // Schleifeninvariante: sum enthält die Summe von array[0] bis array[i-1]
      sum += array[i];
   }
   return sum;
}
```

→ sum ist zu jedem Zeitpunkt sie Summe von array[0] bis array [i-1]

Making impossible state impossible

Speziell die Dateninvarianten helfen bei der Zuschnürrung von Types, um frühzeitig Fehler durch ungültige Zustände zu vermeiden.

Mit festgelegten Constraints für gültige Programmzustände sorgen

```
public static int getFirstElement(int[] array) {
    if (array == null || array.length == 0) {
        throw new IllegalArgumentException("Array must have at least one element.");
    }
    return array[0];
}
```

Beispielsweise wird hierbei vor der konkreten Operation der Methode sichergestellt, dass das übergebene Array immer mindestens 1 Element enthält.

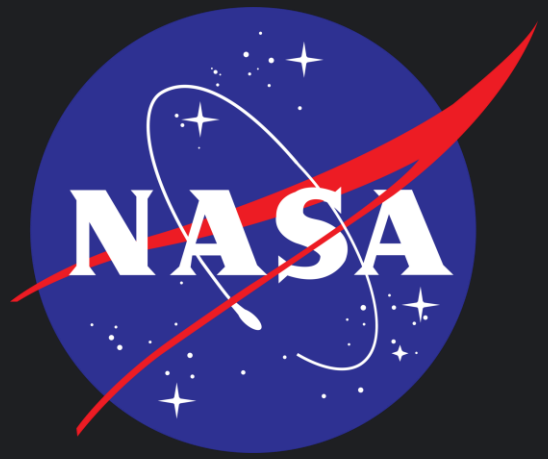
Making impossible state impossible

Code Beispiel

Nasa´s Tiger Style

Die **Nasa** hat ihren eigenen Programmierstil, der spezifisch beim Thema Error Handling einige interessante Ansätze lehrt.

ACHTUNG! Keine feste Muster-Richtlinie an die man sich unbedingt halten muss, lediglich ein „Vorschlag“ für einen robusten Programmierstil !



Nasa´s Tiger Style

1. Kontrollflüsse:

Kontrollflüsse simpel halten (für Klarheit)

→ Begrenzung aller Operationen

→ Keine Rekursion verwenden, nutze Schleifen mit festen Obergrenzen.

Keine dynamische Speicherallokationen (nur statische)

→ Keine Infinite Loops und unvorhersehbare Fehler

→ Dadurch einfacheres und vorhersehbareres Debugging möglich.

Nasa´s Tiger Style

2. Assertions als Sicherheitsnetz:

Assertions nutzen für:

→ Prüfung von Pre- /Postbedingungen (Argumente und Rückgabewerte prüfen)

→ Prüfung von Invarianten, durch explizite Annahmen über Zustände im Code sicherstellen.

→ Prüfung von Datenvalidität vor und nach Dateioperationen.

Assertions splitten:

Statt: assert(a && b)

Besser: assert(a)
assert(b)

Nasa´s Tiger Style

3. Funktionen

Aufteilung der Codestruktur:

→ Teile Codestruktur seperat in Flusskontrolle und konkrete Funktionalität auf.

→ "Push Ifs Up, Fors Down": Kontrollstrukturen in der Hauptfunktion; Helferfunktionen bleiben zustandslos

Funktionsstruktur:

→ Funktionen klein halten (< 70 Zeilen)

→ Variablen im kleinstmöglichen Scope deklarieren.

→ Beim aufteillen von zu großen Funktionen, alle ,,if“ und ,,switch“ Statements in der Parent-Funktion lassen die nicht verzweigte Logik in die Hilfsfunktion packen.

Nasa´s Tiger Style

Kurzgefasst:

Reduktion von Komplexität und Dynamik, mehr Sicherstellungen durch Assertions

→ Dafür mehr Vorhersehbarkeit von Fehlern, Klarheit beim Lesen und Einfachheit beim Debugging

Techniken zum Umgang mit Errors

Recoverable und Non-Recoverable Errors

Recoverable Errors:

→ Vorhersehbare Fehler, bei denen das Programm durch Fallback-Lösungen trotzdem weiter fortsetzen kann.

→ In Java genauer unter ,,Checked Exceptions“ bekannt.

Non Recoverable Errors:

→ Unvorhersehbare Fehler, meistens verursacht durch Programmierfehler / Bugs und fehlerhafte Vorbedingungen.

→ In Java genauer unter ,,Unchecked Exceptions“ bekannt.

Checked Exceptions

Von außen kommende, vorhersehbare Fehler:

→ Compiler zwingt zur Fehlerbehandlung

→ Fehlerbehandlung üblicherweise durch Techniken wie „try-catch-Blöcke“

→Beispiele: IOException, SQLException

Verwendung bei Operationen, die unsicher aber vorhersehbar sind.

Checked Exceptions

Hochpropagieren und auf höherer Ebene behandeln:

```
static void checkedExceptionWithThrows() throws FileNotFoundException {
    File file = new File("not_existing_file.txt");
    FileInputStream stream = new FileInputStream(file);
}
```

Direkt behandeln:

```
static void checkedExceptionWithTryCatch() {
    File file = new File("not_existing_file.txt");
    try {
        FileInputStream stream = new FileInputStream(file);
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
}
```

← Streng genommen ein Verstoß gegen das Single-Responsibility-Prinzip

→ In beiden Fällen, zwingt der Compiler zur Behandlung!

Unchecked Exceptions

Unvorhersehbar, entstehen meistens durch Programmierfehler:

→ Compiler zwingt nicht zur expliziten Behandlung

→ Behandlung möglich, aber Fehler nicht zu erwarten

→ Sollten idealerweise durch defensive Programmierung vermieden werden

→ Beispiele: NullPointerException, IllegalArgumentException, ArrayIndexOutOfBoundsException

Unchecked Exceptions

Unerwartete Fehler können hervorgerufen werden:

```
public int calculateDivision(int numerator, int denominator) {
   // Es gibt kein explizites throw. Division durch null löst automatisch ArithmeticException aus
   // Unchecked Exception
   return numerator / denominator;
}
```

→ Bleiben Fehler wie die Division durch 0 unbehandelt, werden diese Fehler automatisch der Aufruf-Reihenfolge nach hochpropagiert bis zur main() Methode. Spätestens dann stürzt das Programm endgültig ab.

Checked und Unchecked Exceptions

→ Beide Behandlungsarten sollten gezielt verwendet werden, um sauberen Code mit klaren Kontrollstrukturen zu implementieren.

→ Durch das Hochpropagieren von Errors können Fehlerbehandlungen und die reine Anwendungslogik in eigene Segmente getrennt werden und somit insgesamt zu klarer lesbarem Code beitragen.

Codebeispiel in der IDE

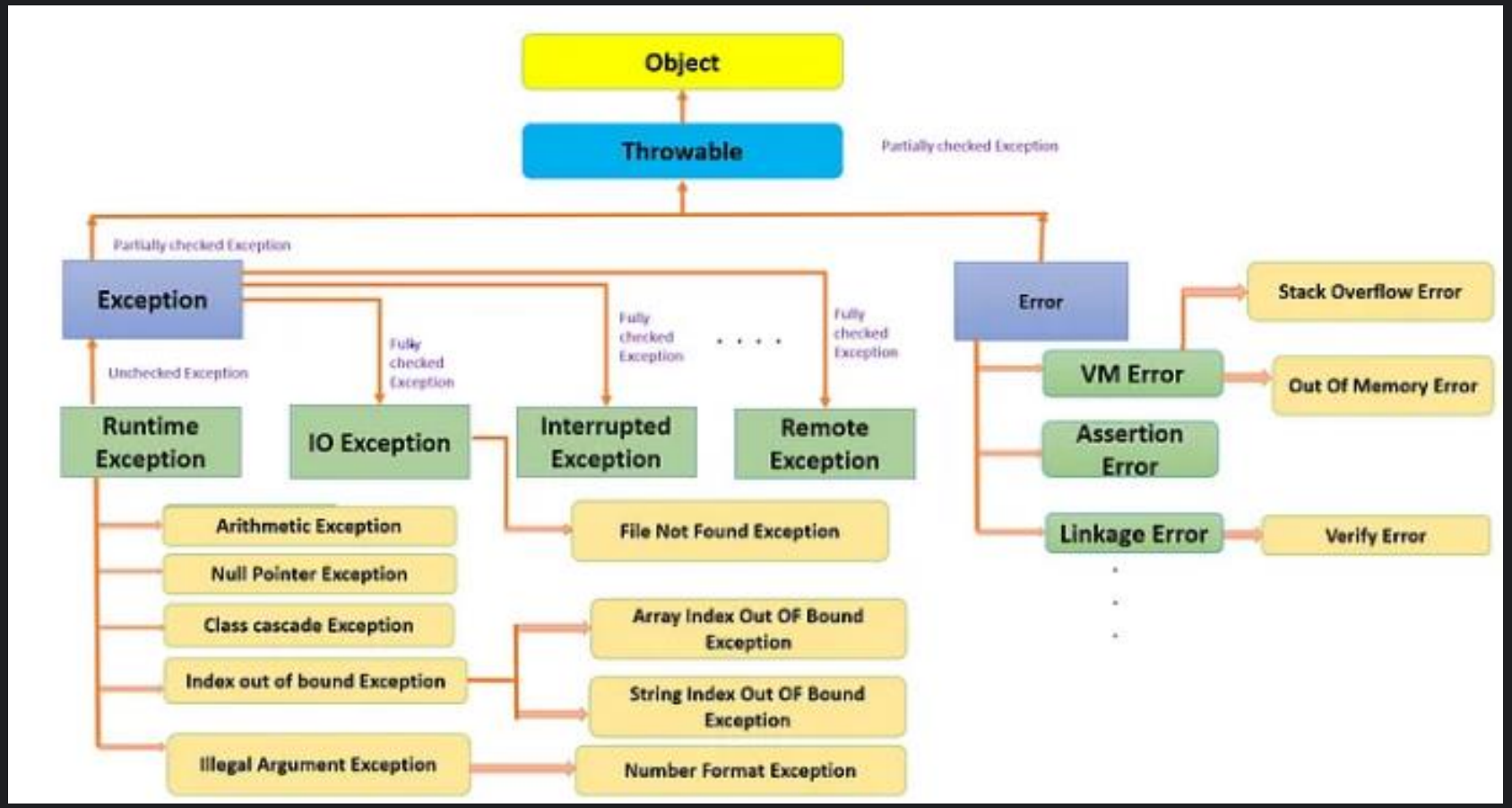
Catch specific Exception

Fangen von konkreten Ausnahmen:

Java implementiert mehrere Klassen für Exceptions in einer komplexen Typhierarchie.
Diese Hierarchie umfasst:
Allgemeine Klassen wie: ,,Exception“
und spezialisierte Subklassen wie:
FileNotFoundException, ArrayIndexOutOfBoundsException etc.

Catch specific Exception

Fangen von konkreten Ausnahmen:



Catch specific Exception

Es ist wichtig, dass statt eines allgemeineren Errors, spezifische Errors gecatcht werden :

→Sonst werden **alle** möglichen Fehlerarten mit aufgefangen, auch die, die an einer konkreten Stelle im Code nicht behandelt werden sollen.

→ Es ist besser, das Programm bei einem unerwartetem Fehler abstürzen zu lassen, damit Entwickler schnell auf einen Fehler aufmerksam werden und ihn gezielt beheben können.

→ Mehrere Exception Types spezifisch abfangen, auch wenn dafür mehr Codezeilen nötig sind.

→ Manche Fehler können sonst verborgen bleiben (Risiko einer Heunadelsuche)

Catch specific Exception

Fangen von konkreten Ausnahmen:

Statt allgemeine Exceptions zu fangen:

```
try {
    //...
} catch (Exception e) {
    //...
}
```

Besser nach spezifischen Exceptionklassen fangen:

```
try {
    int result = example.calculateDivision(10, 0);
    System.out.println("Ergebnis: " + result);
} catch (ArithmeticException e) {
    System.err.println("Fehler: Division durch null ist nicht erlaubt.");
    System.out.println("Bitte geben Sie einen anderen Wert ein.");
}
```

Damit werden auch keine relevanten Exceptions ,,geschluckt“ und durch allgemeinere Exceptions ,,verdeckt“.

Feedback durch Logs

Logging hilft dabei, aussagekräftige und informative Nachrichten für Entwickler und Benutzer an bestimmten Programmstellen zu setzen.

→Bessere Nachvollziehbarkeit von Zuständen im Progamm für Entwickler.

→ Zur Fehlerüberwachung nützlich.

→ Können wichtige Ereignisse, Fehlermeldungen und Diagnosedaten enthalten.

Feedback und Logs

Exceptions wenn möglich nicht beim Exceptionnamen belassen.

→ Sowohl Entwicklern als auch Nutzern mehr Kontext über den Fehler liefern.

```
catch (ArithmeticException e) {
    System.err.println("Fehler: Division durch null ist nicht erlaubt.");
    System.out.println("Bitte geben Sie einen anderen Wert ein.");
}
```

Feedback durch Logs

Statt nur die Exceptionklasse, auch den Stacktrace mitprinten !

Erleichtert das Debugging und vermeidet Heunadelsuchen.

Beispiel:

Code

```
catch (ArithmeticException e) {
    System.err.println("Fehler: Division durch null ist nicht erlaubt.");
    e.printStackTrace();
    System.out.println("Bitte geben Sie einen anderen Wert ein.");
}
```

Konsole

```
Fehler: Division durch null ist nicht erlaubt.
java.lang.ArithmeticException Create breakpoint : / by zero
    at DivisionExample.calculateDivision(Main.java:25)
    at Main.main(Main.java:49)
Bitte geben Sie einen anderen Wert ein.

Process finished with exit code 0
```

Error Handling Tools

Logger Frameworks

Logger Frameworks ermöglichen es, noch detailliertere Logs zu erstellen.

Können auch als ,,Audit-Tools“ angesehen werden.

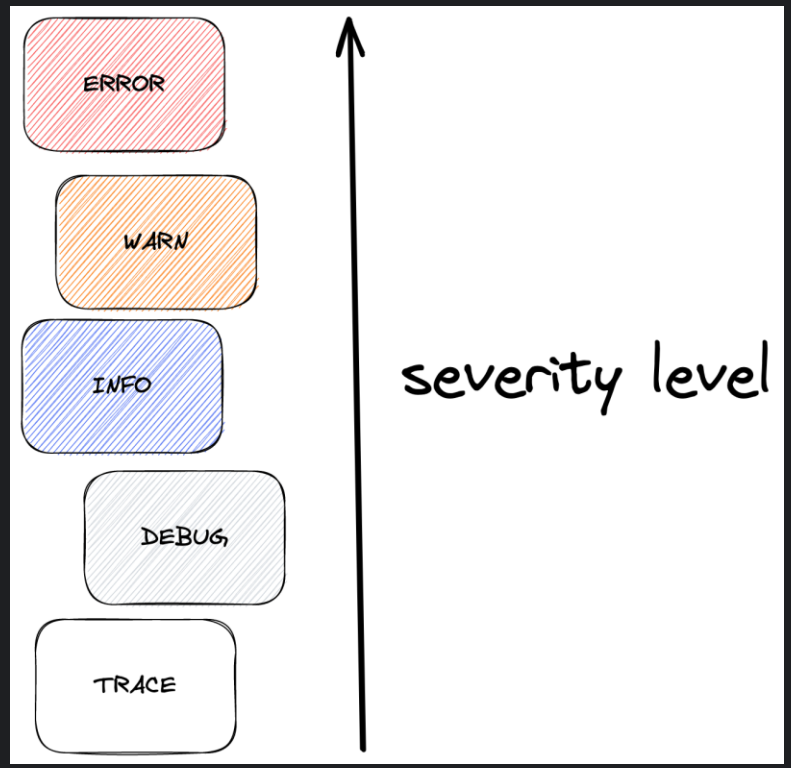
Für große, produktionsreife Anwendungen optimal, da hierbei eine hohe Anzahl detaillierterer Logs oft benötigt wird.

Log4j ist beispielsweise ein solches Logger Framework für Java,
welches nützliche Features für detaillierte Logs bereitstellt.

Logger Frameworks

Das ,,**schöne**“ an den Logs von Log4j: die Log-Level

Log4j bietet Gewichtungen für verschiedene Log-Arten → Bessere Unterteilung



Logger Frameworks

TRACE (600): Für feine Infos, z.B zum Nachverfolgen des Codes, wie etwa wenn eine Methode aufgerufen wird.

DEBUG (500): Logs die während der Entwicklung verwendet werden, wie etwa zur Prüfung von Zuständen im Programm und ob Operationen korrekt ablaufen. Werden in der Produktionsumgebung oft nicht aktiviert.

INFO (400): Loggt normale Laufzeitinformationen wie etwa API-Call oder Datenbank-Zugriff Events

WARN (300): Hinweise auf potentielle Probleme, die aktuell keine Fehler sind, später jedoch eventuell kritisch werden könnten.

ERROR(200): Unerwartete Fehler die aufgetreten sind und die Anwendung daran hindern, korrekt ausgeführt zu werden. Erfordern Behebung, führen aber nicht unbedingt zum Absturz

FATAL(100): Kritische Fehler, führen zum Systemabsturz, müssen umgehend behoben werden.

Logger Frameworks

Logstufen in der Praxis:

```
logger.trace("Entering method processOrder().");
logger.debug("Received order with ID 12345.");
logger.info("Order shipped successfully.");
logger.warn("Potential security vulnerability detected in user input: '...'");
logger.error("Failed to process order. Error: {. . .}");
logger.fatal("System crashed. Shutting down...");
```

Logger Frameworks

Performance Vorteil

→ Ermöglicht es eine hohe Anzahl an Logs zu tätigen, ohne dass die Performance des Programms dabei leidet. Kann I/O Operationen auf einem separatem Thread laufen lassen.

Anpassbare Speicherorte dank „Appender“

→Vielzahl an Appenders ermöglichen es, Logs an verschiedene Ziele zu senden wie etwa Dateien, Konsolen, Datenbanken, Sockets etc. Können bspw. Auch im JSON, XML oder CSV-Format ausgegeben werden

Log Rotation und Archivierung

→ Logs können in kleinere, handhabbare Dateien aufgeteilt werden. Aufteilung kann nach Kriterien wie Dateigröße oder Zeitintervall geschehen. Alte Logs können dabei archiviert oder gar gelöscht werden.

Logger Frameworks

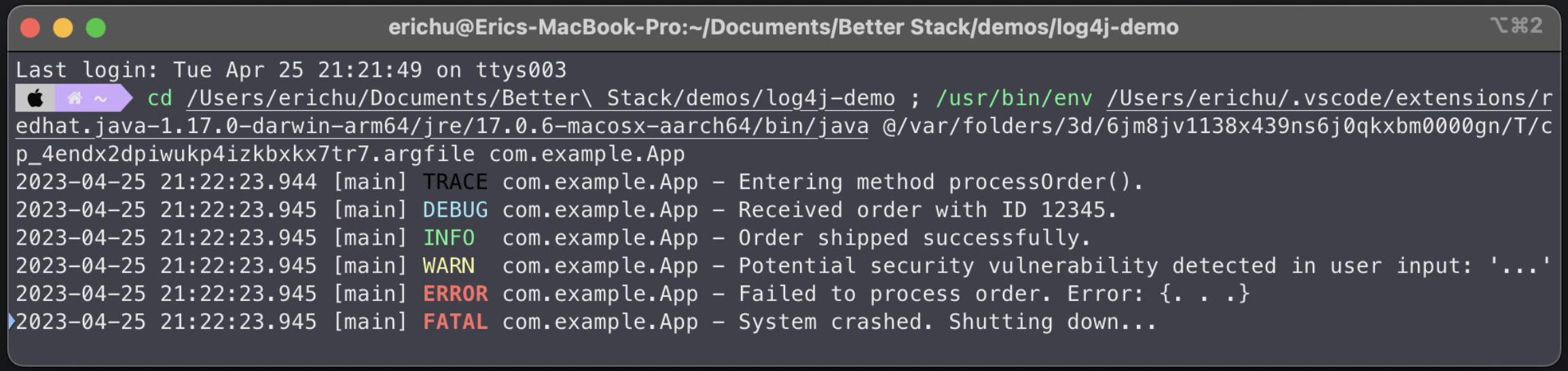
Filterung und Formatierungen von Nachrichten (z.B nach Log-Level)

→ Die Ausgabe der Logs kann nach spezifischen Anforderungen formatiert und gefiltert werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">

    <Appenders>
        <Console name="console" target="SYSTEM_OUT">
         → <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>
    </Appenders>

    <Loggers>
        <Root level="info">
            <AppenderRef ref="console" />
        </Root>
    </Loggers>

</Configuration>
```

Logger Frameworks

Log4j-Ausgabe Beispiel in der Konsole:



Supervisor-Prozesse

Linux Systemd:

Systemmanager für Linux-Systeme

Starten, Beenden und Überwachen von Systemdiensten

Automatisches Neustarten von Prozessen im Fehlerfall

Log-Verläufe können gespeichert werden

```
[Unit]
Description=Beispiel Service
After=network.target

[Service]
ExecStart=/path/to/your/program       # Startbefehl für den Prozess
Restart=always                        # Automatischer Neustart bei Fehler
RestartSec=5                          # Wartezeit vor dem Neustart (z. B. 5 Sekunden)
Environment=ENV_VAR=value             # Optionale Umgebungsvariablen
WorkingDirectory=/path/to/working/dir # Arbeitsverzeichnis des Prozesses
```

Supervisor-Prozesse

PM2 für Node.js:

Ähnlich wie Linux Systemd, ein Tool zur Verwaltung und Überwachung von Prozessen, jedoch für Node.js Projekte

```
[tknew:~/Unitech/pm2] master(+84/-121)+* ± pm2 list
PM2  Process listing
```

App Name	id	mode	PID	status	Restarted	Uptime	memory	err logs
bashscript.sh	6	fork	8278	online	0	10s	1.379 MB	/home/tknew/.pm2/logs/bashscript.sh-err.log
checker	5	cluster	0	stopped	0	2m	0 B	/home/tknew/.pm2/logs/checker-err.log
interface-api	3	cluster	7526	online	0	3m	15.445 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	2	cluster	7517	online	0	3m	15.453 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	1	cluster	7512	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log
interface-api	0	cluster	7507	online	0	3m	15.449 MB	/home/tknew/.pm2/logs/interface-api-err.log

Supervisor-Prozesse

Docker-Container:

Docker erlaubt es ebenfalls Container bzw. Prozesse automatisch neuzustarten

„restart=always““ – Richtlinie aktiviert den automatischen Neustart

```
docker run --name my-container \
 --restart=always \
 -d nginx
```

```yaml
services:
  web:
    image: nginx
    container_name: my-container
    restart: always
```

Core-Dumps

Speicherabbilder (Core Dumps):

→ Snapshot eines Programmspeichers zum Zeitpunkt des Absturzes.

→ Enthält Informationen über die letzten Zustände der Variablen und den Call-Stack.

→ Hilft beim Debugging, entstandene Fehler und die Bedingungen die zum Fehler geführt haben, genauer zu analysieren.

Wann geeignet anzuwenden ?

→ Schwerwiegende Fehler, low-level-analyse in der Speicherverwaltung, komplexe seltene Fehler

Auslese-Tools für verschiedene Core-Dumps:

Windows Minidumps mit:	**Windbg**
Linux Core Dumps mit:	**gdb**

Quellen

- Clean Code: A Handbook of Agile Software Craftsmanship – Robert C. Martin
- Besser Coden: Best Practices Für Clean Code - Uwe Post Rheinwerk Verlag
- Debugging Embedded and Real-Time Systems: The Art, Science, Technology, and Tools of Real-Time System Debugging – Arnold S. Berger
- IEC 61508
- Clean Code in C#: Refactor your legacy C# code base and improve application performance by applying best practices – Jason Alls

Quellen

- Good Habits for Great Coding: Improving Programming Skills with Examples in Python – Michael Stueben
- Java by Comparison: Becomve a Java Craftsman in 70 Examples – Simon Harrer, Jörg Lenhard, Linus Dietz.
- https://moboudra.com/post/making-impossible-states-impossible-in-typescript
- https://logging.apache.org/log4j/2.x/manual/getting-started.html UND log4j-users-guide.pdf

Quellen

- https://betterstack.com/community/guides/logging/how-to-start-logging-with-log4j/#improving-log4j-s-performance
- https://github.com/tigerbeetle/tigerbeetle/blob/main/docs/TIGER_STYLE.md
- https://learn.microsoft.com/de-de/troubleshoot/windows-client/performance/read-small-memory-dump-file
- https://www.baeldung.com/java-assert– Michael Pratt 2024
- The Concept of Class Invariant in Object-oriented Programming – Bertrand Meyer, Alisa Arkadova, Alexander Kogtenkov 2024

Quellen

- https://www.geeksforgeeks.org/c-sharp-nullable-types/
- https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/operators/member-access-operators
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing

Unit Testing ist ein Thema das am Thema Error Handling anknüpft. Es sollte sicherstellen, dass Programmbestandteile komplett fehlerfrei laufen.