# Inhaltsverzeichnis



# 1. Exceptions & Exception-Handling

Beim Ausführen von Anweisungen in Java Programmen kann es (immer) zu **Fehlern** im Code kommen. Eine gute Ausnahmebehandlung kann Fehler behandeln und das Programm ordnungsgemäß umleiten, um Benutzern - trotz Fehler - ein positives Erlebnis zu bieten.

Normalerweise wird Code in einer idealisierten Umgebung geschrieben: Das Dateisystem enthält immer unsere Dateien, das Netzwerk ist fehlerfrei und die JVM verfügt immer über genügend Speicher. Manchmal wird dies auch als **"happy path"** bezeichnet.

In der Produktion können jedoch Dateisysteme beschädigt werden, Netzwerke zusammenbrechen und JVMs nicht mehr über genügend Speicher verfügen. Das Funktionieren des Codes hängt von vielen Aspekten ab, die während der Programmierung nicht beachtet werden oder bekannt sind.

Die Software muss mit diesen Bedingungen umgehen können, da sie den Ablauf der Anwendung negativ beeinflussen und Fehlersituationen bilden:

```
public static List<Player> getPlayers() throws IOException {

    Path path = Paths.get("players.dat");
    List<String> players = Files.readAllLines(path);
    ...
}
```

Dieser Code ist so (mangelhaft) geschrieben, die möglicherweise auftretende Exception nicht zu behandeln, sondern sie stattdessen einfach zu **"werfen"**. In einer idealisierten, kontrollierten Umgebung wird der Code in aller Regel aber einwandfrei funktionieren.

Aber was könnte in der Produktion passieren, wenn die Datei `players.dat` fehlt?

Ohne die **Behandlung** dieser Ausnahme kann es sein, dass ein ansonsten fehlerfreies Programm überhaupt nicht mehr ausgeführt wird! Es muss sichergestellt werden, dass der Code einen Plan für den Fall hat, dass etwas schiefgeht.

# 1.1. Exceptions Hierarchie

Der Java Standard bietet eine API zum **"Fangen"** und zur Behandlung von Fehlern:

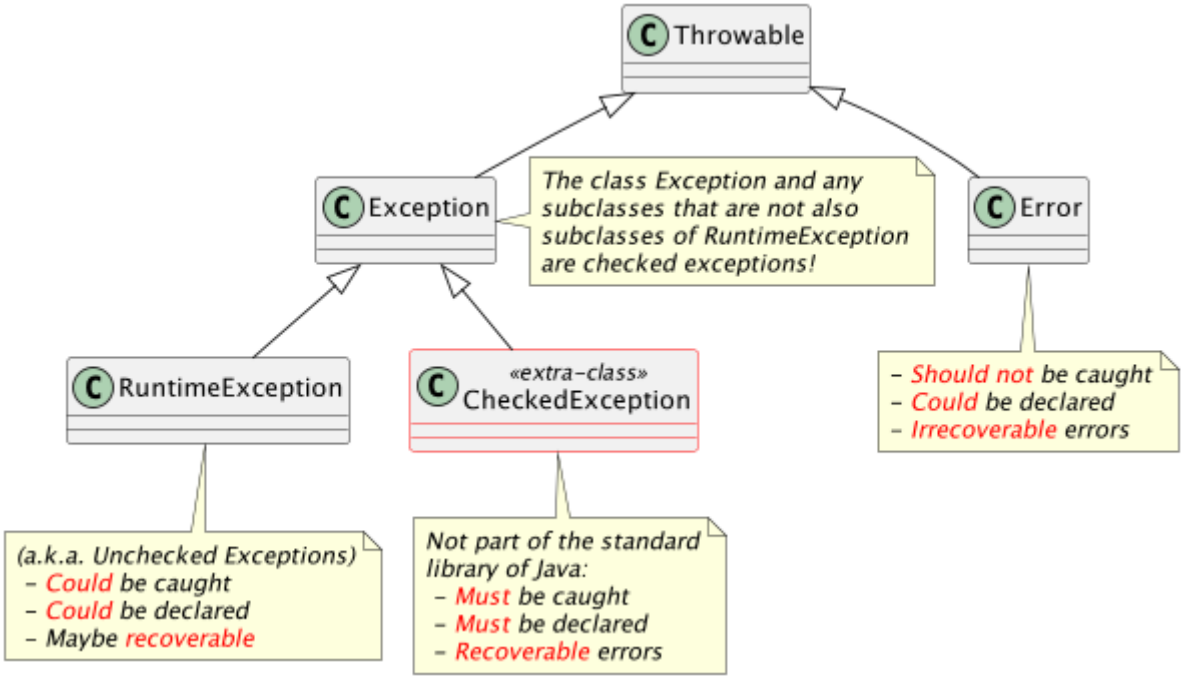


*Figure 1. Exceptions Hierarchie*

**Throwable**

Die `Throwable` Klasse ist die Superklasse von allen `Errors` oder `Exceptions` innerhalb von Java. Nur Objekte, die Instanzen dieser Klasse oder einer seiner Subklassen sind, werden von der JVM selbst geworfen, durch `throw new` manuell geworfen oder das entsprechende Schlüsselwort `throws` deklariert werden. Gleichermaßen können nur diese oder ihre Subklassen als Argumenttyp im `catch` Abschnitt genutzt werden.

Die wichtigsten Codefragmente:

```
public void process() throws ValidationException { ①
   // code that may throw an exception
}

public void process() {
  // code that may throw an exception in
  // a specific situation
  if (!isValid) {
    throw new ValidationException(); ②
  }
}

public void execute() {
  try {
      process(); ③
  } catch (ValidationException ve) { ④
    // ...
```

```
    }
  }
```

① Die Methode zeigt an, dass sie diese `Exception` werfen kann

② Hier wird eine spezielle `Exception` tatsächlich geworfen

③ Hier wird die Methode, die eine Exception wirft, genutzt/aufgerufen …

④ … und behandelt in Form des sogenannten `exception handler`

### Error

Die `Error` Subklasse zeigt ein "ernstes" Problem an, das eine Applikation nicht "fangen" oder "behandeln" sollte. Die meisten solcher Fehler bilden außergewöhnliche Fehlerbedingungen oder -zustände ab, die (in aller Regel) nicht *zur Laufzeit* gehoben werden können.

### Exception

Die Klasse `Exception` und dessen Subklassen bilden Situationen im Code ab, die bekannt sind, eintreten könnten und daher "gefangen" und behandelt werden sollten. Tritt eine solche geplante Fehlersituation auf, so sollte der Fehler so behandelt werden, dass die Applikation nicht abgebrochen werden muss. Eine häufige Reaktion auf diese Art von Fehlern münden häufig in Meldungen an die Benutzer einer Anwendung.

Ein **Beispiel**-Klassenmodell:

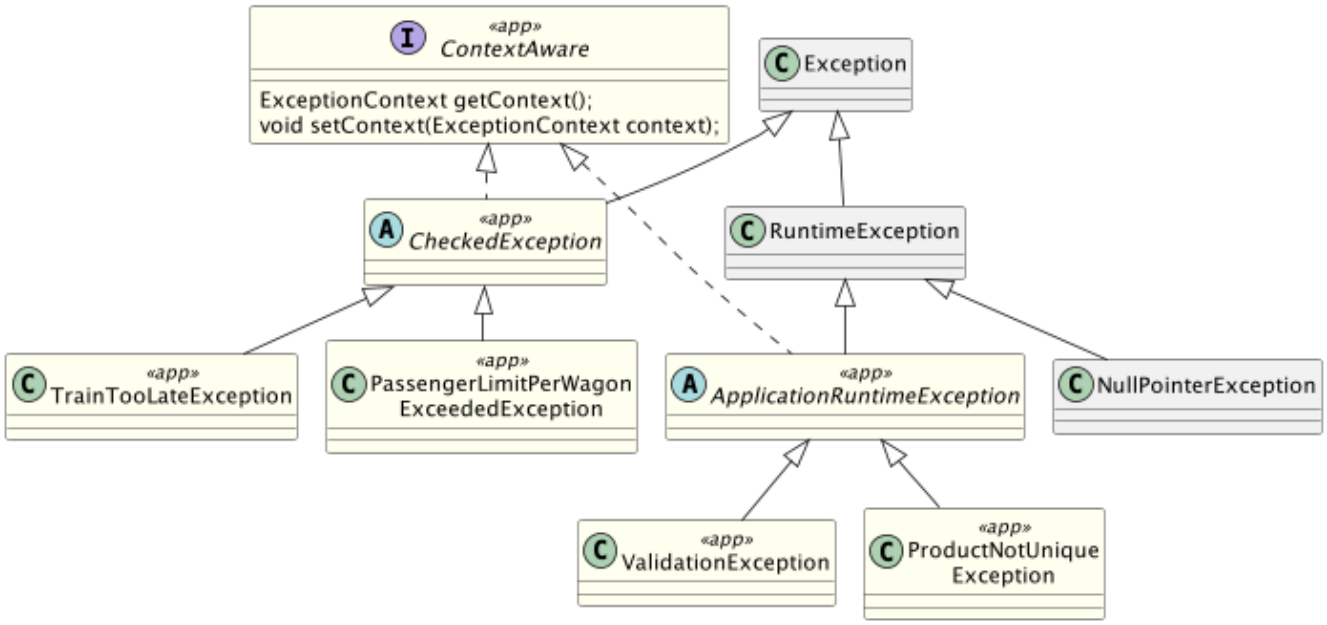


*Figure 2. Exceptions Beispiel*

- *gelbe Klassen sind eigenen Implementierungen, also anwendungsspezifische Erweiterungen der allgemeinen Java Exceptions API*
- *graue Klassen werden von Java zur Verfügung gestellt und sind Teil der API*

## 1.2. Exception Handling

Grundsätzlich stellt Java einen sogenannten `exception handler` bereit, und zwar einfach mithilfe des `catch` Abschnittes. Hier sollte der abgefangene Fehler "behandelt" werden,

aber **NICHT** so (sogenanntes **"Anti-Pattern"**):

```
} catch (FileNotFoundException e) {
    // DON'T DO THIS!
    e.printStackTrace();
}
```

Besser ist eine **echte Verarbeitung** des Fehlers. Hier sind sehr verschiedene Prozesse möglich, anhängig vom eingetretenen Fehler. In vielen Fällen sind bei Fehlern sowohl ...

- *technische* als auch
- *fachliche*

Dinge zu tun. Dazu kann z.B. ein eigener, applikationsspezifischer `ExceptionHandler` eingesetzt werden, der die Behandlung an eine andere Komponente *delegiert*:

```
try {
    risky();
} catch (Exception ex) {
    // do something important for this exception situation
    this.getLogger().error(ex);
    this.rollback();
    this.clearCache();
    this.sendAlarmTo(vip);
}
```

oder

```
catch (IOException ioe) {
    // we want to handle this exception
    // our own way, using an application-
    // specific exception handler!
    handler.handle(ioe);
}
```

## 1.3. Mehrere Exceptions

In manchen Fällen gibt es Methoden oder Codeabschnitte, die gleich mehrere Fehler verursachen können. Sind dies `checked` Exceptions, so müssen sie alle mittels `catch` erfasst und behandelt werden. Dazu gibt es 3 Optionen:

1. Fangen der **allgemeinsten Exception** als derjenigen, von denen alle anderen vorkommenden Exceptions abgeleitet sind

```
try {
    risky();
} catch (Exception ex) {
    // ...
}
```

→ *Anti-Pattern: Generische Exception Handler*

2. **Mehrere** `catch` Abschnitte

```
try {
    risky();
} catch (FileNotFoundException ex) {
    // ...
} catch (EOFException ex) {
    // ...
}
```

3. Ein **Multi-Catch** Block

```
try {
    risky();
} catch (FileNotFoundException | EOFException ex) {
    // ...
}
```

→ *Anti Pattern "Throw-Rethrow"*

```
try {
    risky();
} catch (FileNotFoundException ex) {
    throw new IAmSureThisIsAMuchBetterException(ex);
}
```

## 1.4. Der `finally` Block

Der `finally` Block, der grundsätzlich zum Konstrukt `try-catch-finally` gehört, wird *immer* ausgeführt, wenn der `try` Block beendet wird. Dies stellt sicher, dass der `finally` Block auch dann ausgeführt wird, wenn eine unerwartete Exception aufgetreten ist. Darüber hinaus ist der `finally` Block auch über das reine Exception Handling hinaus nützlich, er erlaubt dem Entwickler insbesondere ein *clean up* durchzuführen, d.h. allokierte Ressourcen wie z.B. geöffnete Dateien

oder Speicherbereiche wieder freizugeben:

```
String data = "data-to-save-to-file";
try {

    // can go wrong
    fileWriter.write(data);

} catch (Exception ex) {
    // ... handle 'expectable' exception
} finally {
    // clean up 'ressources'
    if (fileWriter != null) {
        f.close();
    }
}
```

Vor allem bei der Verarbeitung von Dateien bei sogenannten `IO` (kurz für Input-Output) Operationen - siehe Beispiel-Code - ist das *CleanUp* sehr wichtig und eine gute Praxis!

## 1.5. `throws` und `throw`

Das `throws` **Schlüsselwort** zeigt im Rahmen einer Methodensignatur an, dass diese Methode eine Ausnahme werfen könnte.

Das Schlüsselwort `throw` dagegen wirft eine tatsächliche, konkrete Exception.

```
public void vote(Vote vote, Voter voter) throws TooYoungToVoteException {
  if (voter.getAge() < 18) {
    throw new TooYoungToVoteException(
      "Voter's must be at least 18 years old!");
  }
  // continue normally ...
}
```

## 1.6. Demonstrationen

Die Unit-Tests zur **Demonstration** finden sich hier:

```
src/test/java/de/dhbw/exceptions/ExceptionsDemoTests.java
```

Der zugehörige, in den Tests genutzte **Quellcode** findet sich hier:

```
src/main/java/de/dhbw/exceptions/demo/*.java
```

## 1.7. Exercises

Nutze folgendes Package für deine **Unit-Tests**:

```
src/test/java/de/dhbw/exceptions/ExceptionsExerciseTests.java
```

Die im Test benutzten **Implementierungen** gehören in das Package:

```
src/main/java/de/dhbw/exceptions/exercises/*.java
```

**Übung 1:**

1. **Implementiere** das Interface `Executable` mit einer Methode `void execute()`, dazu eine Klasse Task, die das Interface implementiert
2. **Implementiere** dazu auch eine neue *konkrete, eigene* Exception, die von `CheckedException` abgeleitet werden soll und von `execute()` geworfen werden kann.
3. **Wirf** diese neue Exception einfach mittels `throws` in der konkreten Methode `execute()` deiner konkreten Klasse.
4. Schreibe nun einen Unit-Test, der die Methode `execute()` aufruft und die geworfene Exception **fängt** und **behandelt**.

## 1.8. Tipps, Patterns & Best Practices

**Empfehlung: Flache Exception Hierarchien**

sind eine gute Praxis. Es erleichtert vor allem Entwicklern den Zugang zur Nutzung zum Exception Handling, da es sehr "gerne" vernachlässigt wird.

**Anti Pattern: Generische Exception Handler**

Das ist ein "Anti-Pattern", weil die wahre Fehlerursache hierdurch sehr schnell verschleiert wird!

**Anti Pattern: Throw-Rethrow Exceptions**

Das *"throw-rethrow"* Muster ist auch ein Anti-Pattern. Auch dieses erschwert stark das Erkennen der Fehlerursache und erzeugt einfach viel Code (*"Boilerplate-Code"*). Es verstösst auch gegen das `KISS` Prinzip (*Keep-It-Simple-And-Stupid*).