Kapitel 9 - Exceptions und Logging

Inhalt - Überblick

1. Java Grundlagen: Entwicklungszyklus, Entwicklungsumgebung

2. Datentypen, Kodierung, Binärzahlen, Variablen, Arrays

3. Ausdrücke, Operatoren, Schleifen und Verzweigungen

4. Blöcke, Sichtbarkeit und Methoden (Teil 1)

5. Grundkonzepte der Objektorientierung

6. Objektorientierung: Sichtbarkeit, Vererbung, Methoden (Teil 2), Konstruktor

7. Packages, lokale Klassen, abstrakte Klassen und Methoden, Interfaces, enum

8. Arbeiten mit Objekten: Identität, Listen, Komparatoren, Kopien, Wrapper, Iterator

9. Fehlerbehandlung: Exceptions und Logging

10. Utilities: Math, Date, Calendar, System, Random

11. Rekursion, Sortieralgorithmen und Collections

12. Nebenläufigkeit: Arbeiten mit Threads

13. Benutzeroberflächen mit Swing

14. Streams: Auf Dateien und auf das Netzwerk zugreifen

Exceptions

- Strukturierte Behandlung von Fehlern, die während der Ausführung des Programms auftreten
- Beispiel:

```
7       public static void main(String[] args){
8
9           List<Integer> liste = new ArrayList<Integer>();
10          liste.add(1);
11          liste.add(2);
12
13          liste.get(5);
14
15      }
```

java.lang.IndexOutOfBoundsException: Index: 5, Size: 2

- exception: Die eigentliche Fehler-Ausnahme
- throwing: Eine Ausnahme auslösen
- catching: Eine Ausnahme behandeln

- Ein Laufzeitfehler oder eine vom Entwickler gewollte Bedingung löst eine Ausnahme aus
- Diese kann entweder in dem Programmteil, der sie ausgelöst hat, behandelt werden oder sie kann weitergegeben werden
- Wird die Ausnahme weitergegeben, so hat der Empfänger wiederum die Möglichkeit, diese zu behandeln (catch) oder sie erneut weiterzugeben (throws)
- Wird die Exception an keiner Stelle behandelt, so bricht das Programm ab

Eine Exception auslösen

```
public static int add(Integer x, Integer y) throws PG1MathException {

    if (x == null || y == null) {

        throw new PG1MathException("x oder y ist null.", x, y);
    }
    return x + y;
}
```

Eine Exception weiterreichen

```
public static int add(Integer x, Integer y) throws PG1MathException {

    if (x == null || y == null) {

        throw new PG1MathException("x oder y ist null.", x, y);
    }
    return x + y;
}


public static int add(Float x, Float y) throws PG1MathException {
    return add(x.intValue(), y.intValue());
}
```

Eine Exception abfangen

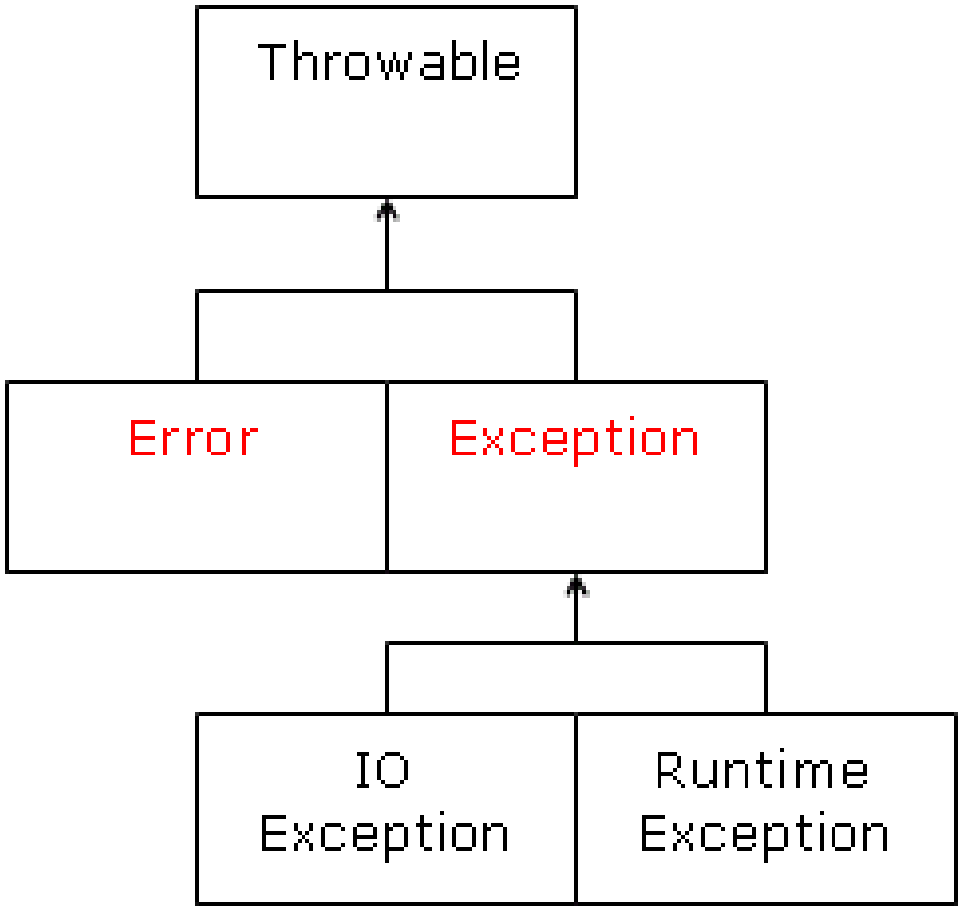
```
public static int add(Integer x, Integer y) throws PG1MathException {

    if (x == null || y == null) {

        throw new PG1MathException("x oder y ist null.", x, y);
    }
    return x + y;
}

public static int add(Float x, Float y)    {
    int result = 0;
    try {
        result = add(x.intValue(), y.intValue());
    } catch (PG1MathException e) {
        e.printStackTrace();
    }
    return result;
}
```

- Der try-Block enthält (mehrere) Anweisungen, bei deren Ausführung ein Fehler eines bestimmten Typs auftreten kann

```
try{
    // Mehrere Anweisungen für den Normalfall ...
    liste.get(5);
} catch (IndexOutOfBoundsException e) {
    // Anweisungen für den Fehlertyp IndexOutOfBoundsException ...
    e.printStackTrace();
} catch (Exception e) {
    // Anweisungen für alle anderen Fehlertypen ...
    System.out.println("Ein Fehler ist aufgetreten: " + e.getMessage());
} finally {
    // Abschließende Maßnahmen ...
    System.out.println("Das erledigen wir immer (mit und ohne Fehler)");
}
```

- catch macht keine Aktionen des Programms rückgängig
- Mit catch erreicht man eine Fortsetzung des Programms an einer bestimmten Stelle
- Je nach Typ des Fehlers kann eine andere Fehlerbehandlung erfolgen, d. h. es kann mehrere catch Blöcke geben
- finally wird immer durchlaufen (mit oder ohne Fehler)

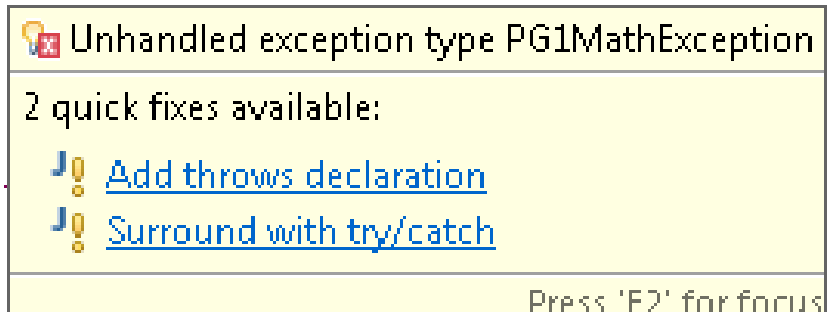
Fehlerklasse (Typen) in Java



- Throwable: Die Vaterklasse aller Exceptions
- Error: Ein schwerwiegender Fehler, der nicht abgefangen werden sollte („abnormal condition“)
- Exception: Die Hauptklasse für die wichtigsten checked exceptions
  - RuntimeExcpetion: Laufzeitfehler, die nicht abgefangen werden müssen
  - IOException: Fehler beim Zugriff auf Ein- / Ausgabegeräte (z. B. Dateizugriff)

- Checked exceptions (Vaterklasse Exception) müssen behandelt werden, sonst gibt es einen Compiler-Fehler

```
add(4, null);
```



- Runtime exceptions (Vaterklasse RuntimeException) können behandelt werden, ohne Behandlung gibt es keinen Compiler-Fehler

[Programm.java]

Wichtige Methoden von Throwable

Method Summary

Throwable	**fillInStackTrace**() Fills in the execution stack trace.
Throwable	**getCause**() Returns the cause of this throwable or `null` if the cause is nonexistent or unknown.
String	**getLocalizedMessage**() Creates a localized description of this throwable.
String	**getMessage**() Returns the detail message string of this throwable.
StackTraceElement[]	**getStackTrace**() Provides programmatic access to the stack trace information printed by `printStackTrace()`.
Throwable	**initCause**(Throwable cause) Initializes the *cause* of this throwable to the specified value.
void	**printStackTrace**() Prints this throwable and its backtrace to the standard error stream.

- Eine eigene Klasse von (checked) Exception oder RuntimeExpcetion ableiten, bspw. um zusätzliche Informationen im Fehlerfall abzulegen:

```
2  public class PG1MathException extends Exception{
3
4      private static final long serialVersionUID = -2704495885990850175L;
5
6      private Integer x;
7      private Integer y;
8
9      public PG1MathException(String message, Integer x, Integer y) {
10         super(message);
11         this.x = x;
12         this.y = y;
13     }
```

[PG1MathException.java]

Logging mit Log4J

http://logging.apache.org/log4j

- Ein Logger protokolliert Informationen über den Programmablauf
- Diese Informationen können
  - in eine Datei,
  - in die Console oder
  - in eine Datenbank, usw.

  geschrieben werden
- Die Informationen, welche protokolliert werden, sind in verschiedene Kategorien (Loglevel) eingeteilt

Übliche Logging-Kategorien (Loglevel)

- Trace: ausführliches Debugging inkl. Kommentare
- Debug: allgemeines Debugging (Methode x wurde mit Parameter y aufgerufen, etc.)
- Info: allgemeine Informationen (Programm gestartet, Verbindung aufgebaut, etc.)
- Warn: Auftreten einer unerwarteten Situation
- Error: Fehler, das Programm läuft aber weiter
- Fatal: Kritischer Fehler, Programmabbruch

Ordnung:

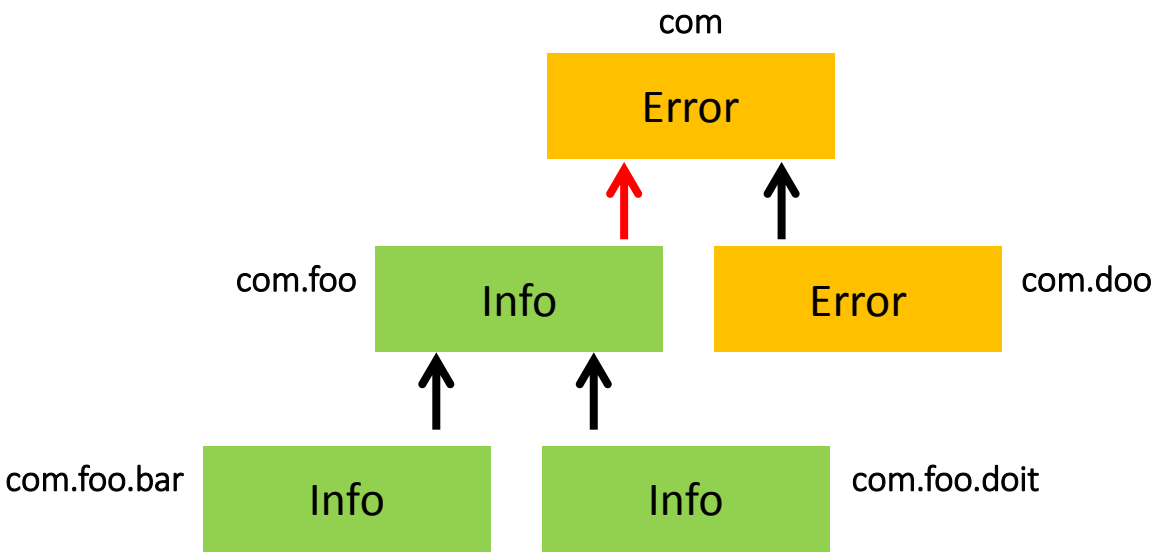
TRACE > DEBUG > INFO > WARN > ERROR > FATAL

Log4J Logger anlegen

- Logger in Log4J besitzen einen eindeutigen Namen
- Die Logger sind in einer Baum-Hierarchie geordnet
- Mit Hilfe des Logger-Namens wird die Position des Loggers in der Hierarchie gesetzt (vgl. Packages):
  - Der Logger „com.foo“ ist Vater des Loggers „com.foo.Bar“
  - Der Logger „com.doit“ ist Kind des Loggers „com“
- An der Wurzel der Baumhierarchie befindet sich der Root-Logger

- Beispiel: Das Loglevel auf Error setzen:
- Damit werden nur Nachrichten der Kategoriae ERROR oder niedriger Kategorien (hier: FATAL) ausgegeben
- Beispiel: Das Loglevel auf Info setzen:
- Damit werden nur Nachrichten der Kategorie INFO oder niedriger Kategorien (hier: WARN, ERROR, FATAL) ausgegeben

- Das Loglevel eines Loggers kann direkt gesetzt werden (s. Folie zuvor) oder es wird vom übergeordneten Logger in der Hierarchie vererbt



- Im Programmcode werden an bestimmten Stellen Logging-Ausgaben erzeugt
- Abhängig von der Stelle im Programm muss man sich für die geeignete Logging-Kategorie (Level) entscheiden
- Beispiel für eine Error-Nachricht:

```
try{
    int result = 16 / 0;
} catch (Exception e) {
    logger.error("Bei der Berechnung ist ein Fehler aufgetreten", e);
}
```

- Beispiel für eine Info-Nachricht:

```
public static void main(String[] args) {

    Logger logger = Logger.getRootLogger();
    logger.info("Das Programm wurde gestartet");
```

- Beispiel für eine warn-Nachricht:

```
Scanner sc = new Scanner(System.in);
System.out.println("Geben Sie einen Benutzernamen ein: ");
String input = sc.next();

if (input.length() < 3){
    logger.warn("Der eingegebene Benutzername " + input +
                " hat weniger als 3 Zeichen");
}
```

- Die Log-Nachrichten können an verschiedene Log-Ziele (auch an mehrere gleichzeitig) ausgegeben werden
- Ein Ausgabe-Ziel von Log4J wird als **Appender** bezeichnet
- Es gibt Appender für die Console, Dateien, Windows Ereignisprotokolle, etc.
- Dabei wird die Logger-Hierarchie berücksichtigt
  - Jeder Log-Appender des Vater-Loggers erhält auch alle Logging-Nachrichten seiner Kind-Logger

- Beispiel für eine Log-Nachricht in einem bestimmten Ausgabeformat:

```
176 [main] INFO  org.foo.Bar - Located nearest gas station.
```

- Das Ausgabeformat lässt sich mit Hilfe der Klasse PatternLayout anpassen
- Dieses conversion pattern erzeugt beispielsweise das oben angezeigte Ausgabeformat

%r [%t] %-5p %c - %m%n

- Siehe dazu http://logging.apache.org/log4j/1.2/apidocs/org/apache/log4j/PatternLayout.html

- Eine einfache Start-Konfiguration, die auf die Console schreibt:

```
BasicConfigurator.configure();
```

- In der Praxis wird Log4J meist per Property-Datei konfiguriert:

```
PropertyConfigurator.configure("log4j.properties");
```

- Vorteil: Man kann die Logging-Einstellungen ändern ohne das Programm neu zu kompilieren
- Beispiel für eine Property-Datei: siehe *log4j.properties*

- Wird das Loglevel bspw. auf ERROR gesetzt, so werden insbesondere keine DEBUG-Nachrichten ausgegeben
- Das Programm durchläuft aber trotzdem immer die Anweisungen zur Debug-Nachrichtenausgabe
- Dies ist kein Effizienzproblem, sofern keine Parameter in die Nachricht eingesetzt werden
- Mit Parametern sollte die folgende if-Prüfung verwendet werden:

```
if (logger.isDebugEnabled()){
    logger.debug("Die Summe der Variablen i und j ist: " + i+j );
}
```