

## Teil

# Objektorientierte Programmierung

## Unterrichtseinheit 35

# Exceptions

**Dr. Dietrich Boles**

- Motivation Fehlerbehandlung
- Fehlerarten
- Fehlerquellen
- Traditionelle Fehlerbehandlung
- Fehlerbehandlung in Java im Überblick
- Fehlerklassen
- Fehlertypdeklaration
- "Werfen" von Fehlerobjekten
- "Abfangen" von Fehlerobjekten
- Anmerkungen
- Exceptions im Hamster-Modell
- Beispiel ADT-Klasse Stack
- Beispiel Klasse IO
- Zusammenfassung

```
public class Mathematik {  
    public static int fak(int n) {  
        if (n > 0) return n * fak(n-1);  
        else return 1; // falsch, wenn n < 0 (nicht def.)  
    }  
    public static double div(double wert, double durch) {  
        if (durch != 0.0) return wert/durch;  
        else return 0.0; // eigentlich falsch (unendlich)  
    }  
}
```

```
class Berechnungen {  
    public static void main(String[] args) {  
        double eingabe = IO.readDouble();  
        IO.println(Mathematik.fak(eingabe));  
        IO.println(Mathematik.div(eingabe, eingabe-2.0));  
    }  
}
```

- Hintergrund: Definition von zur Verfügung zu stellenden Klassen, in denen bestimmte Fehler auftreten können.
- "Schwere" Fehler           ⇒       Programmabbruch
- "leichte" Fehler           ⇒       Korrektur
- "schwer" / "leicht" unter Umständen situationsabhängig
  - Beispiel: Division durch 0
    - Programmabbruch
    - Aufforderung zur "Neueingabe"
- weitere Frage: Wer soll auf Fehler reagieren?
  - Programmierer einer Funktion
  - Aufrufender einer Funktion

- Inkonsistente Variablenwerte
- ungültige Parameterwerte
- falsche Adressen / Dateinamen
- nicht erfüllte Vorbedingungen (lesen erst nach Öffnen einer Datei)
- ungültiger Array-Index
- Zugriff auf Objekt über Objektvariable mit Wert `null`
- arithmetische Fehler
- kein Speicherplatz mehr vorhanden
- Endlosrekursion
- ...

## Ausgabe einer Fehlermeldung:

```
public class Mathematik {
    public static int fak(int n) {
        if (n > 0) return n * fak(n-1);
        else if (n == 0) return 1;
        System.err.println("ungueltiger Parameter");
        return -1;
    }
}
class Berechnungen {
    public static void main(String[] args) {
        int eingabe = IO.readInt();
        IO.println(Mathematik.fak(eingabe));
    }
}
```

**Problem: Aufrufendes Programm bekommt den Fehler nicht mit!**

## Benutzung einer error-Variablen:

```
public class Mathematik {
    public static final int NOERROR = 0;
    public static final int INVALID = 1;
    public static int error;
    public static int fak(int n) {
        if (n > 0) { error = NOERROR; return n*fak(n-1); }
        if (n == 0) { error = NOERROR; return 1; }
        error = INVALID; return -1;
    } }
...
int eingabe = IO.readInt();
int result = Mathematik.fak(eingabe);
if (Mathematik.error == Mathematik.NOERROR)
    IO.println(result);
else if (Mathematik.error == Mathematik.INVALID)
    System.err.println("ungueltiger Parameter");
...

```

**Problem: Ausgesprochen umständlich!**

- Bei der Klassen-/Methodendefinition:
  - überlegen, was für Fehler auftreten können
  - Definition und Implementierung geeigneter "Fehlerklassen"
  - Erweiterung der Methodensignatur um Angabe möglicher Fehler
  - erzeugen und liefern ("werfen") von "Fehlerobjekten" im Falle eines Fehlereintritts
  
- Bei der Klassen-/Methodennutzung:
  - beim Aufruf einer Methode ermitteln, welche Fehler prinzipiell auftreten können
  - zunächst "versuchen", die Methode auszuführen
  - falls kein Fehler auftritt, normal weitermachen
  - falls ein Fehler auftritt, das Fehlerobjekt abfangen und geeignete Fehlerbehandlung einleiten

- überlegen, was für Fehler auftreten können

```
public static int fak(int n) {  
    if (n > 0)  
        return n * fak(n-1);  
    else if (n == 0)  
        return 1;  
    else  
        ????????  
}
```

Fehlerquelle: ungültiger aktueller Parameterwert ( $n < 0$ )

- Definition und Implementierung geeigneter "Fehlerklassen"

```
public class InvalidParameter
    extends Exception
{
    int actParamValue;
    public InvalidParameter(int value) {
        super("Ungültiger Parameter");
        this.actParamValue = value;
    }
    public int getParamValue() {
        return this.actParamValue;
    }
}
```

- Erweiterung der Methodensignatur um Angabe möglicher Fehler
- erzeugen und liefern ("werfen") von "Fehlerobjekten" im Falle eines Fehlereintritts

```
public static int fak(int n)
    throws InvalidParameter
{
    if (n > 0)
        return n * fak(n-1);
    else if (n == 0)
        return 1;
    else {
        InvalidParameter errorObj =
            new InvalidParameter(n);
        throw errorObj;
    }
}
```

- beim Aufruf einer Methode ermitteln, welche Fehler prinzipiell auftreten können

```
static void doSomething() {  
    int eingabe = IO.readInt();  
    int result = fak(eingabe);  
    for (int i=0; i<result; i++)  
        IO.println(i);  
}
```

möglicher Fehler: ungültiger aktueller Parameterwert für `fak`-Aufruf  
(`eingabe < 0`)

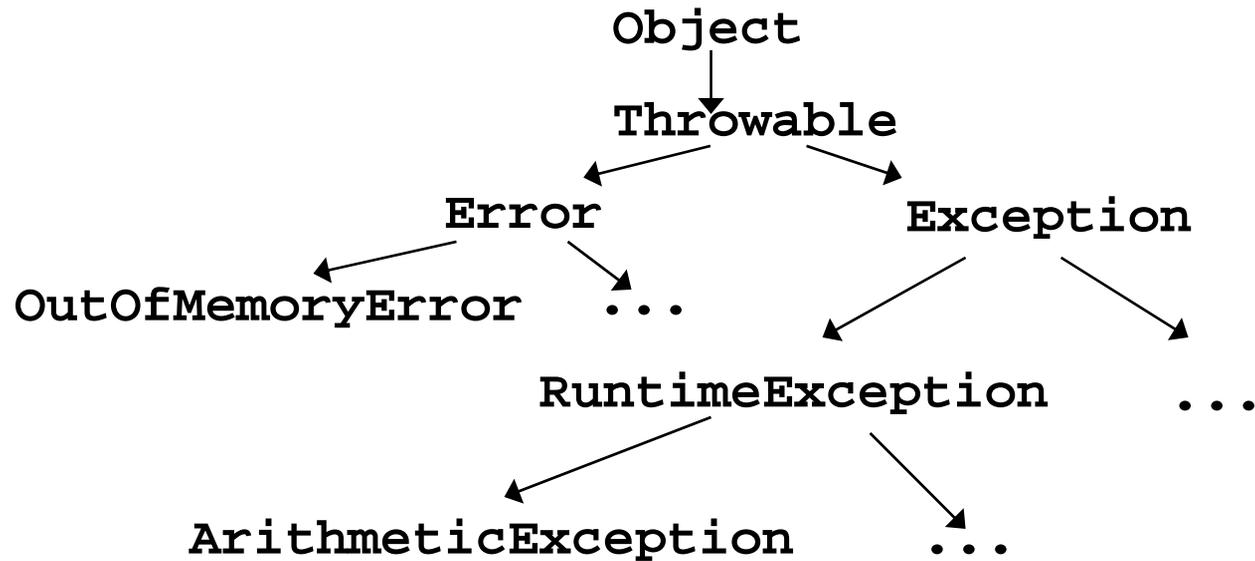
- zunächst "versuchen", die Methode auszuführen
- falls kein Fehler auftritt, normal weitermachen

```
static void doSomething() {  
    try {  
        int eingabe = IO.readInt();  
        int result = fak(eingabe);  
        for (int i=0; i<result; i++)  
            IO.println(i);  
    }  
}
```

- **Motivation:** Trennung von Normalfall und möglichen Fehlerfällen

- falls ein Fehler auftritt, das Fehlerobjekt abfangen und geeignete Fehlerbehandlung einleiten

```
static void doSomething() {
    try {
        int eingabe = IO.readInt();
        int result = fak(eingabe);
        for (int i=0; i<result; i++)
            IO.println(i);
    } catch (InvalidParameter errorObj) {
        System.err.println("ungültige Eingabe: " +
            errorObj.getParamValue() + "wiederholen");
        doSomething(); // erneuter Versuch
    } }
```



- **Error**: schwerwiegende Fehler
- **RuntimeException**: schwerwiegende Fehler
- Fehlerobjekte (*Exceptions*) sind Objekte von Fehlerklassen (von **Throwable** (zumeist indirekt) abgeleitete Klassen!)
- neue Fehlerklassen im Allgemeinen von **Exception** ableiten!
- Name einer Fehlerklasse sollte aussagekräftig sein!

```
package java.lang;

public class Exception extends Throwable {
    // Konstruktoren
    public Exception();
    public Exception(String message);

    // Methoden
    public String getMessage(); // liefert Beschreibung
    public String toString();   // ruft getMessage auf
}
```

- Deklaration von Fehlertypen einer Methode in der Methodensignatur mittels des Schlüsselwortes `throws`
- Eine Methode **muss** alle Fehlertypen deklarieren, die sie werfen kann, (oder sie muss sie selbst "abfangen") (Fehlerobjekte werden "weitergeleitet")
- **Error**- und **RuntimeException**-Fehler müssen nicht unbedingt deklariert werden (Unchecked-Exceptions)

```
public class Ex1 extends Exception {}
public class Ex2 extends Exception {}
public int f() throws Ex1, Ex2 {
    ... if (...) throw new Ex1(); ...
    ... if (...) throw new Ex2(); ...
}
public int g() throws Ex1 {
    try { ... int i = f(); ... } catch (Ex2 e) {...}
}
```

- Die Fehlertypdeklaration gehört mit zur Signatur einer Methode
- Werden Methoden mit Fehlertypen in abgeleiteten Klassen **überschrieben**, dann müssen die Fehlertypdeklarationen übereinstimmen oder es müssen Fehlerklassen angegeben werden, die von den Fehlerklassen der überschriebenen Methode abgeleitet sind!

```
public class Ex1 extends Exception {}
public class Ex2 extends Exception {}
public class Ex3 extends Ex2 {}
public class A {
    public int f(float v) throws Ex1, Ex2 { ... }
}
public class B extends A { // nur eine der Alternat.
    public int f(float v) throws Ex1, Ex2 { ... }
    public int f(float v) throws Ex1, Ex3 { ... }
}
```

- "Werfen" von Fehlerobjekten mittels des Schlüsselwortes **throw**
- nach dem Werfen eines Fehlerobjektes wird eine Methode (wie beim **return**) direkt verlassen
- Es können nur Objekte solcher Fehlerklassen geworfen werden, die auch in der Methodendeklaration aufgeführt sind

```
public class InvalidParam extends Exception {
```

```
public class Math {  
    public static int fak(int n) throws InvalidParam {  
        if (n < 0) throw new InvalidParam();  
        if (n == 0) return 1;  
        return n * fak(n-1);  
    }  
}
```

- Methoden, die (andere) Methoden aufrufen, die Fehlerobjekte werfen können, können diese entweder weiterleiten (an die aufrufende Methode dieser Methode) oder sie müssen sie abfangen (und behandeln)
- Zum Abfangen eines Fehlerobjektes muss die entsprechende Methode innerhalb eines so genannten **try-Blockes** aufgerufen werden
- Die Fehlerbehandlung wird in einem so genannten **"den Fehlertyp matchenden"-catch-Block** durchgeführt
- wird ein Fehlerobjekt geworfen, so wird die Bearbeitung des try-Blockes unmittelbar beendet
- existiert ein matchender-catch-Block, werden die Anweisungen dieses Blockes ausgeführt

```
try {  
    ...  
} catch (XYException obj) {  
    ...  
}
```

# "Abfangen" von Fehlerobjekten (2)

- existiert ein so genannter **finally-Block**, wird dieser auf jeden Fall ausgeführt
- wird ein Fehlerobjekt während der Abarbeitung eines try-Blockes geworfen und existiert kein matchender-catch-Block, so wird falls vorhanden der finally-Block ausgeführt und anschließend die Methode sofort verlassen und das Fehlerobjekt an die aufrufende Methode weitergeleitet

```
try {  
    ...  
} catch (XException obj) {  
    ...  
} catch (YException obj) {  
    ...  
} finally {  
    ...  
}
```

- Sehr große Ähnlichkeiten zu Prozeduren und Parametern
- muss einem try-Block (oder einem anderen catch-Block) folgen
- "formale Parameter" müssen Fehlerklassen/-Objektvariablen sein, die im try-Block auftreten können (bzw. Oberklassen der Fehlerklassen (→ Polymorphie))
- die Fehlerobjektübergabe ist identisch zur Parameterübergabe bei Funktionen, d.h. insbesondere können die formalen Fehlerobjekte wie lokale Variablen des catch-Blockes behandelt werden; sie werden mit dem geworfenen Fehlerobjekt initialisiert
- wird während der Ausführung des try-Blockes ein Fehlerobjekt geworfen, so wird der try-Block verlassen, und es wird der Reihe nach (!) überprüft, ob ein catch-Block mit dem Fehlerobjekt *matched*; dabei gilt:
- **ein catch-Block *matched*** ein aktuelles Fehlerobjekt, wenn die Klasse seines formalen Fehlerobjektes gleich der Klasse oder eine Oberklasse des aktuellen Fehlerobjektes ist

- ist "parameterlos"
- schließt einen try-`{catch}`\*-Block ab, d.h. kann hinter einem bzw. einer Menge von catch-Blöcken definiert werden
- ist ein finally-Block hinter einem try-`{catch}`\*-Block vorhanden, so wird dieser auf jeden Fall ausgeführt, d.h.
  - wenn kein Fehlerobjekt im try-Block geworfen wurde
  - wenn ein Fehlerobjekt im try-Block geworfen und nicht abgefangen wurde (nach dem finally-Block wird in diesem Fall die Funktion verlassen und das Fehlerobjekt an die aufrufenden Funktion weitergeleitet)
  - wenn ein Fehlerobjekt im try-Block geworfen und abgefangen wurde, d.h. nach der Ausführung des matchenden catch-Blockes

# "Abfangen" von Fehlerobjekten / Beispiel (1)

```
public int f(double v) throws Ex1, Ex2, Ex3 { ... }
public float g(int i) throws Ex1, Ex3, Ex4 { ... }
public void h() throws Ex3 {
    ... (*1)
    try {
        ... (*2)
        int r = f(2.0);
        ... (*3)
        float x = g(r);
        ... (*4)
    }
    catch (Ex1 errorEx1) { ... (*5) }
    catch (Ex2 errorEx2) { ... (*6) }
    catch (Ex4 errorEx4) { ... (*7) }
    finally { ... (*8) }
    ... (*9)
}
```

Erläuterungen (zum Beispiel vorher):

1. wird die Funktion `h` (bspw. von einer Funktion `s`) aufgerufen, dann wird zunächst `(*1)` ausgeführt
2. anschließend wird `(*2)` ausgeführt
3. wird während der Ausführung der Funktion `f` kein Fehlerobjekt geworfen, dann wird `(*3)` ausgeführt (weiter bei 6.)
4. wird während der Ausführung der Funktion `f` ein Fehlerobjekt vom Typ `Ex3` geworfen, dann wird zunächst `(*8)` ausgeführt und danach die Funktion `h` unmittelbar verlassen und das Fehlerobjekt an die aufrufende Funktion `s` weitergeleitet
5. wird während der Ausführung der Funktion `f` ein Fehlerobjekt vom Typ `Ex1` (oder `Ex2`) geworfen, dann wird dieses Objekt (wie bei der Parameterübergabe) als aktueller Parameter dem formalen Parameter `errorEx1` (bzw. `errorEx2`) übergeben und `(*5)` (bzw. `(*6)`) ausgeführt (weiter bei 10.)

# "Abfangen" von Fehlerobjekten / Beispiel (3)

6. nach der Ausführung von (\*3) wird die Funktion `g` aufgerufen
7. wird während der Ausführung der Funktion `g` kein Fehlerobjekt geworfen, dann wird (\*4) ausgeführt (weiter bei 10.)
8. wird während der Ausführung der Funktion `g` ein Fehlerobjekt vom Typ `Ex3` geworfen, dann wird zunächst (\*8) ausgeführt und danach die Funktion `h` unmittelbar verlassen und das Fehlerobjekt an die aufrufende Funktion `s` weitergeleitet
9. wird während der Ausführung der Funktion `g` ein Fehlerobjekt vom Typ `Ex1` (oder `Ex4`) geworfen, dann wird dieses Objekt (wie bei der Parameterübergabe) als aktueller Parameter dem formalen Parameter `errorEx1` (bzw. `errorEx4`) übergeben und (\*5) (bzw. (\*7)) ausgeführt
10. nach Ausführung von (\*4), (\*5), (\*6) oder (\*7) wird auf jeden Fall (\*8) ausgeführt
11. anschließend wird (\*9) ausgeführt und die Funktion beendet

- Fehlerquelle: werden während der Ausführung eines try-Blockes explizit Fehlerobjekte geworfen, können diese u.U. (d.h. es existiert ein matchender catch-Block) direkt wieder von einem catch-Block gefangen werden:

```
class Ex1 extends Exception {}  
public void f() {  
    try { ... throw new Ex1(); ...  
    } catch (Exception obj) { ... }  
}
```

- es kann passieren, dass auch während der Abarbeitung eines catch-Blockes wieder Fehler auftreten (→ Schachtelung):

```
...  
catch (Exception o) {  
    try { ... } catch (Exception obj2) { ... }  
}
```

- auch try-{catch}\*finally-Blöcke lassen sich schachteln
- rücksichtsloses Abbrechen eines Programms: `System.exit(0);`

# Exceptions im Hamster-Modell (1)

```
class HamsterException extends RuntimeException { ... }
class MauerDaException extends HamsterException { ... }
class KachelLeerException extends HamsterException { ... }
class MaulLeerException extends HamsterException { ... }

class Hamster { ...
    void vor() throws MauerDaException {
        if (!this.vornFrei()) throw new MauerDaException();
        switch (this.blickrichtung) { ... }
        ...
    }
    void nimm() throws KachelLeerException { ... }
    void gib() throws MaulLeerException { ... }
}
```

# Exceptions im Hamster-Modell (2)

```
void main() { // Demoprogramm
    Hamster paul = Hamster.getStandardHamster();
    try { // laufe zur Wand
        while (true) {
            paul.vor();
        }
    } catch (MauerDaException exc) {}
    paul.linksUm();
    int anzahl = 0;
    try { // alle Koerner ablegen
        while (true) {
            paul.gib(); anzahl++;
        }
    } catch (MaulLeerException exc) {}
    paul.schreib("Ich habe " + anzahl + " Koerner abgelegt");
}
```

# Beispiel ADT-Klasse Stack (1)

---

```
class Stack { // ohne Exceptions
    // Datenstruktur
    int[] store; // zum Speichern von Daten
    int current; // aktueller Index

    // Methoden
    Stack(int size) { // Konstruktor
        this.store = new int[size];
        this.current = -1;
    }

    boolean isFull() {
        return this.current == (this.store.length-1);
    }

    boolean isEmpty() {
        return this.current == -1;
    }
}
```

# Beispiel ADT-Klasse Stack (2)

```
void push(int value) {  
    this.store[++this.current] = value;  
}
```

```
int pop() {  
    return this.store[this.current--];  
}
```

```
// Testprogramm
```

```
public static void main(String[] args) {  
    Stack stack = new Stack(IO.readInt("Stackgroesse: "));  
    // Stack fuellen  
    while (!stack.isFull()) {  
        stack.push(IO.readInt("Zahl: "));  
    }  
    // Stack leeren  
    while (!stack.isEmpty()) {  
        IO.println(stack.pop());  
    } } }
```

## Beispiel ADT-Klasse Stack (3)

```
class IsFullException extends Exception {}
class IsEmptyException extends Exception {}

class Stack { // mit Exceptions
    // Datenstruktur
    int[] store; // zum Speichern von Daten
    int current; // aktueller Index

    // Methoden
    Stack(int size) { // Konstruktor
        this.store = new int[size];
        this.current = -1;
    }

    void push(int value) throws IsFullException {
        if (this.current == (this.store.length - 1))
            throw new IsFullException();
        this.store[++this.current] = value;
    }
}
```

# Beispiel ADT-Klasse Stack (4)

```
int pop() throws IsEmptyException {
    if (this.current == -1)
        throw new IsEmptyException();
    return this.store[this.current--];
}

// Testprogramm
public static void main(String[] args) {
    Stack stack = new Stack(IO.readInt("Stackgroesse: "));
    // Stack fuellen
    try {
        while (true) stack.push(IO.readInt("Zahl: "));
    } catch (IsFullException exc) {}
    // Stack leeren
    try {
        while (true) IO.println(stack.pop());
    } catch (IsEmptyException exc) {}
}
}
```

```
class IOException extends Exception
```

```
class BufferedReader { // java.io  
    String readLine() throws IOException  
}
```

```
class NumberFormatException extends RuntimeException
```

```
class Integer { // java.lang  
    Integer(String s) throws NumberFormatException  
}
```

# Beispiel Klasse IO (2)

```
class IO {
    public static int readInt() {
        try {
            BufferedReader input =
                new BufferedReader(new InputStreamReader(
                    System.in));
            String eingabe = input.readLine();
            Integer value = new Integer(eingabe);
            return value.intValue();
        } catch (Exception e) {
            return 0;
        }
    }
}
```

- Exceptions dienen zur Behandlung von potentiellen Laufzeitfehlern
- Motivation: Trennung von "Normalfall" und möglichen Fehlerfällen
- Exceptions sind Objekte von von der Klasse `Throwable` abgeleiteten Klassen
- Im Fehlerfall wird von einer Methode eine Exception geworfen
- aufrufende Methoden können diese Exception abfangen und eine Fehlerbehandlung einleiten
- Sinnvoller Einsatz von Exceptions: ADT-Klassen
- nicht sinnvoll: Ersetzen von "normalen" if-Anweisungen