

# Robuste Programme durch Ausnahmebehandlung

---

Martin Wirsing

in Zusammenarbeit mit  
Michael Barth, Philipp Meier und Gefei Zhang

12/04

## Ziele

- Lernen robuste Programme zu schreiben
- Ausnahmen als Objekte verstehen lernen
- Bedeutung von Ausnahmen erkennen in der Signatur und im Rumpf einer Methode
- Lernen Ausnahmebehandlung durchzuführen

## Robuste Programme

**Definition:** Ein Programm heißt **robust**, wenn es für jede Eingabe eine wohldefinierte Ausgabe produziert.

Die Berechnung der Summe zweier ganzer Zahlen durch

```
int x = SimpleInput.readInt();
int y = SimpleInput.readInt();
while (x != 0)
    {
        y = y + 1;
        x = x-1;
    }
```

terminiert nicht für  $x < 0$ . Was tun?

---

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Robuste Programme



Robust, aber  
undurchsichtig

- **Einführung von von zusätzlicher Fallunterscheidung und Fehlermeldung**

```
int x = SimpleInput.readInt();
if (x < 0) System.out.println(„Falscher Eingabewert“);
else
    {
        int y = SimpleInput.readInt();
        while (x > 0)...
```

meldet Fehler durch Seiteneffekt.

---

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Robuste Programme

### ▪ Besser: Kontrolliertes Auslösen von Ausnahmen:

```
int x = SimpleInput.readInt();
if (!x>=0)throw new IllegalArgumentException(
    "Negativer Eingabewert");

int y = SimpleInput.readInt();
while (x != 0)...
```



Auch kein  
robustes  
Programm!

**löst** bei negativem x eine Ausnahme aus („abrupte“ Terminierung!): :

```
Exception in thread „main“
java.lang.IllegalArgumentExceptionError: Negativer Eingabewert
```

---

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Fehlerarten

Ein Programm kann aus vielerlei Gründen fehlerhaft sein. Man unterscheidet u.a.:

- **Entwurfsfehler:** Der Entwurf entspricht nicht den Anforderungen.
- **Programmierfehler:** Das Programm erfüllt nicht die Spezifikation.

---

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Fehlerarten

Programmierfehler können auch unterschiedlicher Art sein:

- **Syntaxfehler:** Die kontextfreie Syntax des Programms ist nicht korrekt.

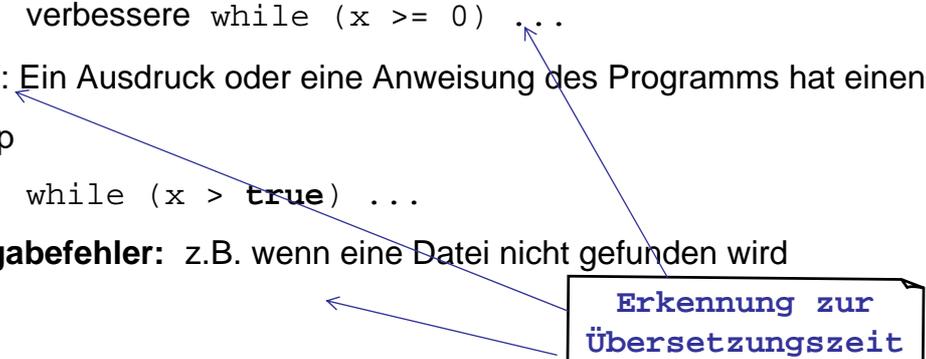
Beispiel: `whill (x > = 0) ...`  
verbessere `while (x >= 0) ...`

- **Typfehler:** Ein Ausdruck oder eine Anweisung des Programms hat einen falschen Typ

Beispiel: `while (x > true) ...`

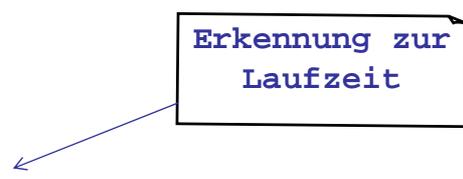
- **Ein/Ausgabefehler:** z.B. wenn eine Datei nicht gefunden wird

Erkennung zur  
Übersetzungszeit  
(Checked Error)



## Fehlerarten

Erkennung zur  
Laufzeit



- **Laufzeitfehler:** Ein Fehler, der während der Ausführung eines korrekt übersetzten Programms auftritt, wie z.B. Division durch Null, fehlende Datei oder Netzwerkfehler.

**Bemerkung:** Syntaxfehler und Typfehler werden zur Übersetzungszeit erkannt. Laufzeitfehler werden in Java durch das Laufzeitsystem dem Benutzer gemeldet. Üblicherweise terminiert ein Java-Programm „unnormale“ beim Auftreten eines Laufzeitfehlers an, was zur Robustheit von Java-Programmen beiträgt.

## Beispiel für Laufzeitfehler: Division durch 0

**Beispiel:** Die Klasse `Exc0`

```
/**
 * Diese Klasse illustriert das Auslösen einer Ausnahme.
 * Bei der Division durch 0 wird eine ArithmeticException ausgelöst
 */
public class Exc0
{
    /**
     * Die Methode main löst wegen der Division durch 0 eine
     * ArithmeticException aus:
     */
    public static void main(String args[])
    {
        int d = 0;
        int a = 42/d;

        System.out.println("d= " + d);
        System.out.println("a= " + a);
    }
}
```

Division durch 0 löst  
arith. Ausnahme aus:  
Abrupte Terminierung

wird nicht ausgeführt wg  
abrupter Terminierung

Aufrufkeller  
enthält nur  
`Exc0.main`

**Java-Ausgabe:**

```
> java Exc0
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:19)
```

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Ausnahmen und Fehler sind Objekte im Java

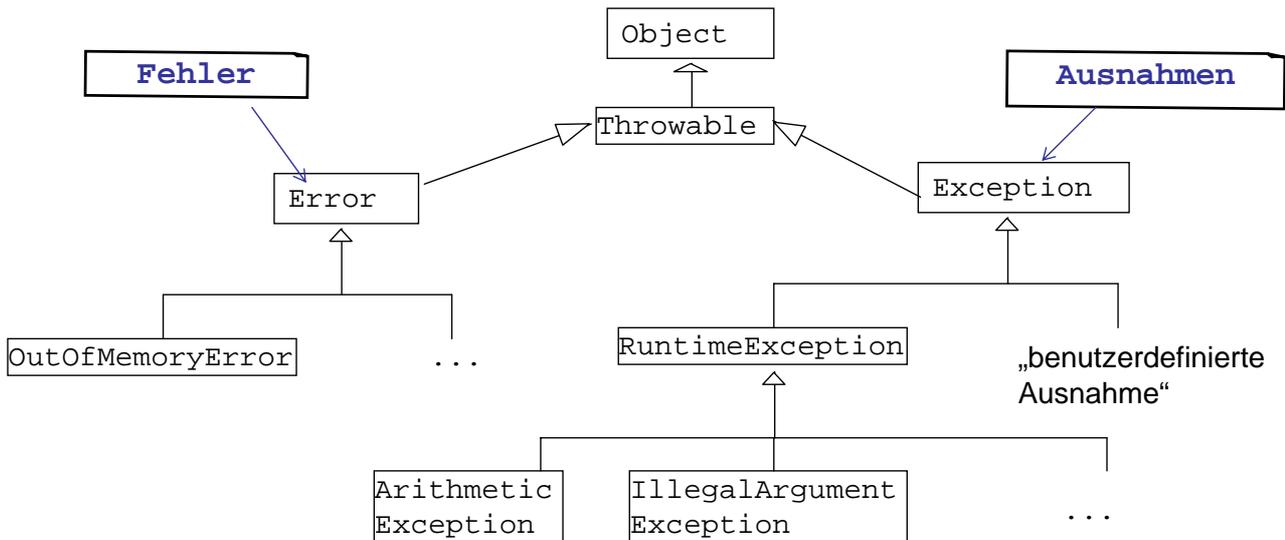
In Java sind auch (Laufzeit-)Fehler *Objekte*. Man unterscheidet zwischen

- Fehlern (Instanzen der Klasse `Error`) schwerwiegend  
nicht abfangen
- Ausnahmen (Instanzen der Klasse `Exception`) abfangen, durch  
Ausnahmebehandlung  
von Programmierer  
definierbar

**Bemerkung:** Ausnahmen können vom Programmierer im Programm durch Ausnahmebehandlung abgefangen werden (und sind vom Programmierer definierbar), Fehler deuten auf schwerwiegende Probleme hin und sollten nie behandelt werden. Ausnahmeobjekte werden vom Java-Laufzeitsystem automatisch erzeugt, wenn eine Fehlersituation auftritt.

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Vererbungshierarchie der Fehlerklassen



`OutOfMemoryError` gibt an, daß der Speicher voll ist.

`ArithmeticException` gibt einen arithmetischen Fehler an, wie Division durch Null .

## Throwable

`Throwable` ist die Standardklasse für Ausnahmen (Laufzeitfehler).

**Konstruktoren:** `Throwable()`, `Throwable (String message)` konstruieren Fehlerobjekte, eventuell mit einer speziellen Nachricht.

Ein Fehlerobjekt enthält:

- einen Schnappschuß des Aufrufkellers zum Zeitpunkt der Erzeugung des Objekts
- Ausnahmen (Instanzen der Klasse `Exception`)

**weitere Methoden:**

- `String getMessage()` : gibt die Fehlermeldung zurück
- `void printStackTrace()` : gibt den momentanen Stand des Aufrufkellers aus

## Beispiel: Schnappschuß des Aufrufkellers beim Auslösen einer Ausnahme

**Beispiel:** Die Klasse `Exc1`

```
/**
 * Bei der Division durch 0 wird eine ArithmeticException ausgelöst.
 * Anders als bei Exc0 wird die Ausnahme in der Methode subroutine
 * ausgelöst, die in main aufgerufen wird.
 */
public class Exc1
{
    public static void subroutine()
    {
        int d = 0;
        int a = 42/d;

        System.out.println("d = " + d);
        System.out.println("a = " + a);
    }
    public static void main(String args[])
    {
        Exc1.subroutine(); // Optionale zusätzliche Angabe von Exc1
    }
}
```

Division durch 0 löst  
arith. Ausnahme aus:  
Abrupte Terminierung

wird nicht ausgeführt wg  
abrupter Terminierung

**Java-Ausgabe:**

```
> java Exc1
Exception in thread „main“ java.lang.ArithmeticException: / by zero
 * at Exc1.subroutine(Exc1.java:19)
   at Exc1.main(Exc1.java:31)
```

Aufrufkeller

## Benutzerdefinierte Ausnahmeklassen

Ausnahmeklassen sind *Subklassen von `Exception`* und werden wie normale Klassen mit Attributen und Konstruktoren deklariert.

**Bemerkung:** Meist benötigt man nur die Methoden von `Throwable`, die es erlauben, den Aufrufkeller auszugeben und die Ausnahmenachricht zu lesen.

**Beispiel**

```
public class NegativeValueException extends Exception
{
    String text;
    int value;

    public NegativeValueException(String text, int value)
    {
        this.text = text;
        this.value = value;
    }
    public String toString()
    {
        return "NegativeValueException[" + text + value + "]"
-- Standardrueckgabe = ";
    }
}
```

## Kontrolliertes Auslösen von Ausnahmesituationen

### ▪ Beispiel:

```
...
int x = SimpleInput.readInt();
if (!x>=0) throw new NegativeValueException(
    "Negativer Eingabewert", x);

int y = SimpleInput.readInt();
while (x != 0)...
```

throw löst Ausnahme  
aus, wenn  $x < 0$

wird nach Ausführung  
von throw nicht mehr  
ausgeführt

## Kontrolliertes Auslösen von Ausnahmesituationen

- Mittels der „**throw**“-Anweisung kann man eine kontrollierte Ausnahme auslösen:

**Syntax:** „**throw**“ *Expression*;

- Der Ausdruck muß eine Instanz einer Subklasse von `Throwable` (d.h. eine Ausnahme oder ein Fehlerobjekt) bezeichnen.
- Die Ausführung einer „**throw**“-Anweisung stoppt den Kontrollfluß des Programms und löst die von Expression definierte Ausnahme aus.
- Die nächste Anweisung des Programms wird nicht mehr ausgeführt.

## Kontrolliertes Auslösen von Ausnahmesituationen

In Java sind kontrolliert ausgelöste Ausnahmen genauso wichtig wie normale Ergebniswerte.

Deshalb wird ihr Typ im Kopf einer Methode angegeben (mit Ausnahme von Subklassen von `Error` und `RuntimeException`). Dies geschieht mittels „**throws**“.

Der Kopf einer Methode erhält folgende Form:

```
<returntype> m (<params>) throws <Exceptionlist>
```

- ! Die Typen der „**throw**“-Anweisungen des Rumpfs **müssen** im Kopf der Methode angegeben werden.

## Kontrolliertes Auslösen von Ausnahmesituationen

### Beispiel: Summe

```
int sum(int x, int y) throws NegativeValueException
{
    if (x<0) throw new NegativeValueException(
        "Negativer Eingabewert fuer x = ", x);
    while (x != 0)
    {
        y = y + 1;
        x = x-1;
    }
    return y;
}
```

## Defensive und Robuste Programme

**Definition:** Man spricht von **defensiver Programmierung**, wenn eine Operation für alle Elemente des Definitionsbereichs normal, d.h. nicht abrupt, terminiert und für alle anderen Eingaben eine Ausnahme auslöst.

### Beispiel: Summe

```
int sum(int x, int y) throws NegativeValueException
{
    if (x<0) throw new NegativeValueException(
        "Negativer Eingabewert fuer x = ", x);

    while (x != 0)
    {
        y = y + 1;
        x = x-1;
    }
    return y;
}
```

Defensiv aber nicht robust!

**Definition:** Ein Programm heißt **robust**, wenn es für jede Eingabe eine wohldefinierte Ausgabe produziert, d.h. wenn es für jede Eingabe normal terminiert.

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Abfangen von Ausnahmen

**Ausnahmebehandlung** geschieht in Java mit Hilfe der „try“-Anweisung, die folgende Grundform hat:

```
try {
    // Block fuer „normalen“ Code
}
catch (Exception1 e) {
    // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception1
}
catch (Exception2 e) {
    // Ausnahmebehandlung fuer Ausnahmen vom Typ Exception2
}
finally {
    // Code, der in jedem Fall nach normalem Ende und nach
    // Ausnahmebehandlung ausgefuehrt werden soll.
}
```

Programm terminiert NICHT „anormal“, sondern der Fehler wird abgefangen, das Programm arbeitet normal weiter

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Informelle Semantik

- In `try` wird der normale Code ausgeführt.
- Tritt eine Ausnahmesituation auf, so wird eine Ausnahme ausgelöst („`throw`“), die je nach Typ von einem der beiden Ausnahmebehandler („`Handler`“) abgefangen („`catch`“) wird.
- Falls die Handler nicht den passenden Typ haben, wird im umfassenden Block nach einem Handler gesucht.
- Falls kein benutzerdefinierter Handler gefunden wird, wird eine Ausnahme ausgelöst, die zu „abrupter“ Terminierung führt.
- Das „`finally`“-Konstrukt ist optional; darin stehender Code wird auf jeden Fall ausgeführt und zwar nach dem normalen Ende bzw. Nach Ende der Ausnahmebehandlung.
- Mindestens ein `catch`- oder `finally`-Block muß vorkommen.

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Beispiel: Abfangen von Ausnahmen

Die Beispielklasse `Exc2` zeigt, wie die Ausnahme bei Division durch 0 abgefangen werden kann. Das Programm arbeitet nach der `try`-Anwendung „normal“ weiter und gibt „Hurra!“ auf dem Bildschirm aus.

```
public class Exc2
{
    public static void subroutine()
    {
        try
        {
            int d = 0;
            int a = 42/d;
        }

        catch (ArithmeticException e)
        {
            System.out.println("division by zero");
        }
        System.out.println("Hurra!");
    }

    public static void main(String args[])
    {
        Exc2.subroutine();
    }
}
```

Division durch 0 löst arith. Ausnahme aus

Robustes Programm durch Ausnahmebehandlung!

Nach Abfangen der arithm. Ausnahme durch `catch` arbeitet das Programm normal weiter, so als ob der Fehler nie vorgekommen wäre!!

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Beispiel für die Wirkung von „finally“

Die Klasse Exc3 löst ähnlich wie Exc1 eine `ArithmeticException` aus, führt aber den Block von „finally“ aus, d.h. „Hallo!“ wird am Bildschirm ausgegeben. Da das Programm aber abrupt terminiert, wird „Hurra!“ **nicht** gedruckt..

```
public class Exc3
{
    public static void subroutine()
    {
        try
        {
            int d = 0;
            int a = 42/d;
        }
        finally
        {
            System.out.println(„Hallo!“);
        }
        System.out.println(„Hurra!“);
    }

    public static void main(String args[])
    {
        Exc3.subroutine();
    }
}
```

Nicht robust:  
abrupte  
Terminierung!

Division durch 0 löst  
arith. Ausnahme aus

„finally“ wird trotz  
abrupter Terminierung  
ausgeführt

wird nicht ausgeführt wg  
abrupter Terminierung

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Beispiel für die Wirkung von „catch“ und „finally“

Die Klasse Exc4 erweitert Exc3 um ein Abfangen der Division durch 0. Deshalb wird sowohl „Hallo!“ als auch „Hurra!“ ausgegeben.

```
public class Exc4
{
    public static void subroutine()
    {
        try
        {
            int d = 0;
            int a = 42/d;
        }
        catch (ArithmeticException e)
        {
            System.out.println(„division by zero“);
        }
        finally
        {
            System.out.println(„Hallo!“);
        }
        System.out.println(„Hurra!“);
    }

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

Robustes  
Programm!

Division durch 0 löst  
arith. Ausnahme aus

„catch“ fängt die  
Ausnahme ab

„finally“ wird außerdem  
ausgeführt

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Robuste Summenmethode

### Beispiel: Summe

```
int sum(int x, int y)
{
    try
    {
        if (x < 0) throw new NegativeValueException(
            "Negativer Eingabewert fuer x = ", x);
        while (x != 0)
        {
            y = y + 1;
            x = x - 1;
        }
        return y;
    }
    catch (NegativeValueException e)
    {
        System.out.println(e.toString()); return 0;
    }
}
```

Kein „throws“, da die Ausnahme abgefangen wird

Robuste Summe terminiert normal: Der Fehler wird ausgegeben, dann arbeitet das umfassende Programm normal weiter.

Jeder Zweig von try braucht return Anweisung

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## SimpleInput ist robust

### Beispiel: SimpleInput

```
import java.io.*;

class SimpleInput
{
    ...
    static String readString()
    {
        BufferedReader input =
            new BufferedReader(new InputStreamReader(System.in));
        try
        {
            return input.readLine();
        }
        catch (IOException exception)
        {
            System.out.println("Fehler in der Eingabe.");
            return "";
        }
    }
}
```

Klasse mit Operationen zum Verarbeiten von Textströmen

Konvertiert byte-Strom in char-Strom

Liest Datenstrom von bytes von Konsole

Liest nächste Textzeile aus Datenstrom

Abfangen von IO-Fehlern

Jeder Zweig von try braucht return Anweisung

Vordefinierte Ausnahmeklasse

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## SimpleInput

### Fortsetzung

```
...
static int readInt()
{   String intString = readString();
    try
    {   return Integer.parseInt(intString);
    }
    catch (NumberFormatException exception)
    {   System.out.println("Keine Zahl " +
        "-- Rückgabewert 0");
        return 0;
    }
}
```

Liest String von  
Konsole

Konvertiert String in  
ganze Zahl

Vordefinierte  
Ausnahmeklasse

Jeder Zweig von try  
braucht return  
Anweisung

Abfangen von  
IO-  
Fehlern

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Auslösen von Ausnahmesituationen

### Bemerkung

Wenn man eine Methode aufruft, die einen Ausnahmetyp in der **throws**-Klausel (im Kopf) enthält, gibt es drei Möglichkeiten:

- Man fängt die Ausnahme mit **catch** ab und behandelt sie, um ein normales Ergebnis zu erhalten.
- Man fängt die Ausnahme mit **catch** ab und bildet sie auf eine Ausnahme (aus dem Kopf) der geeigneten Methode ab.
- Man deklariert die Ausnahme im Kopf der eigenen Methode.

M. Wirsing: Robuste Programme durch Ausnahmebehandlung

## Zusammenfassung

- Ausnahmen sind Objekte.
- Methoden können Ausnahmen auslösen und dann „abrupt“ terminieren.
- Ausnahmen können mit „**catch**“ behandelt werden, damit sie normal terminieren.
- Werden Ausnahmen nicht behandelt, müssen sie im Kopf der Methode erscheinen.
- Defensive Programme lösen für undefinierte Situationen Ausnahmen aus. Robuste Programme terminieren immer - und zwar mit einem wohldefinierten Ergebnis.