Einführung in die Programmierung mit Java

Teil 9: Unified Modeling Language

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

7. Dezember 2017

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Teil 9: Unified Modeling Language

UML-Notation

UML = Unified Modelling Language

ISO Standard zur grafischen Veranschaulichung von Vorgängen bei der Softwareentwicklung:

- Objektdiagramme: visualisieren Speicherinhalt
- Klassendiagramme: visualisieren Klassen und ihre Beziehungen
- Use-case Diagramme: visualisieren Verwendungsszenarien
- Statecharts: visualisieren Zustandsübergänge
- ...

UML

UML-Diagramme werden primär als *Strukturierungs-* und *Modellierunsmittel* auf dem Weg von der informellen Beschreibung zum fertigen Programm eingesetzt:

- Welche Komponenten gibt es?
- Was leistet jede Komponente?
- Wie interagieren die Komponenten?
- Welche Abhängigkeiten gibt es?

Die Ideen werden zuerst in Diagrammen festgehalten, diskutiert und anschließend danach implementiert. Nicht umgekehrt!

Modellierung beinhaltet auch *Abstraktion*:
Wenn z.B. ein Konstruktor nicht aufgelistet wird, dann kann man darauf schließen, dass dessen Verhalten gewöhnlich ist; gibt es Besonderheiten zu beachten, wird dieser im Diagramm aufgeführt.

Speicherdiagramm aus den Übungen

Diagramm zeigt alle vorhandenen Objekte im Speicher („Heap“), und meistens noch alle lokale Variablen mit Inhalt („Stack“-Box).

Jedes Objekt im Heap durch Box mit zwei Teilen repräsentiert:

1. Klassenname
2. Attribute (Instanzvariablen) mit aktuellem Inhalt

Beziehung zwischen Objekten durch Pfeile dargestellt:

⟶ für Referenzen, also Verweise auf andere Objekte

⇒ Gut für erste Übungen, aber schnell umständlich bei vielen Objekten

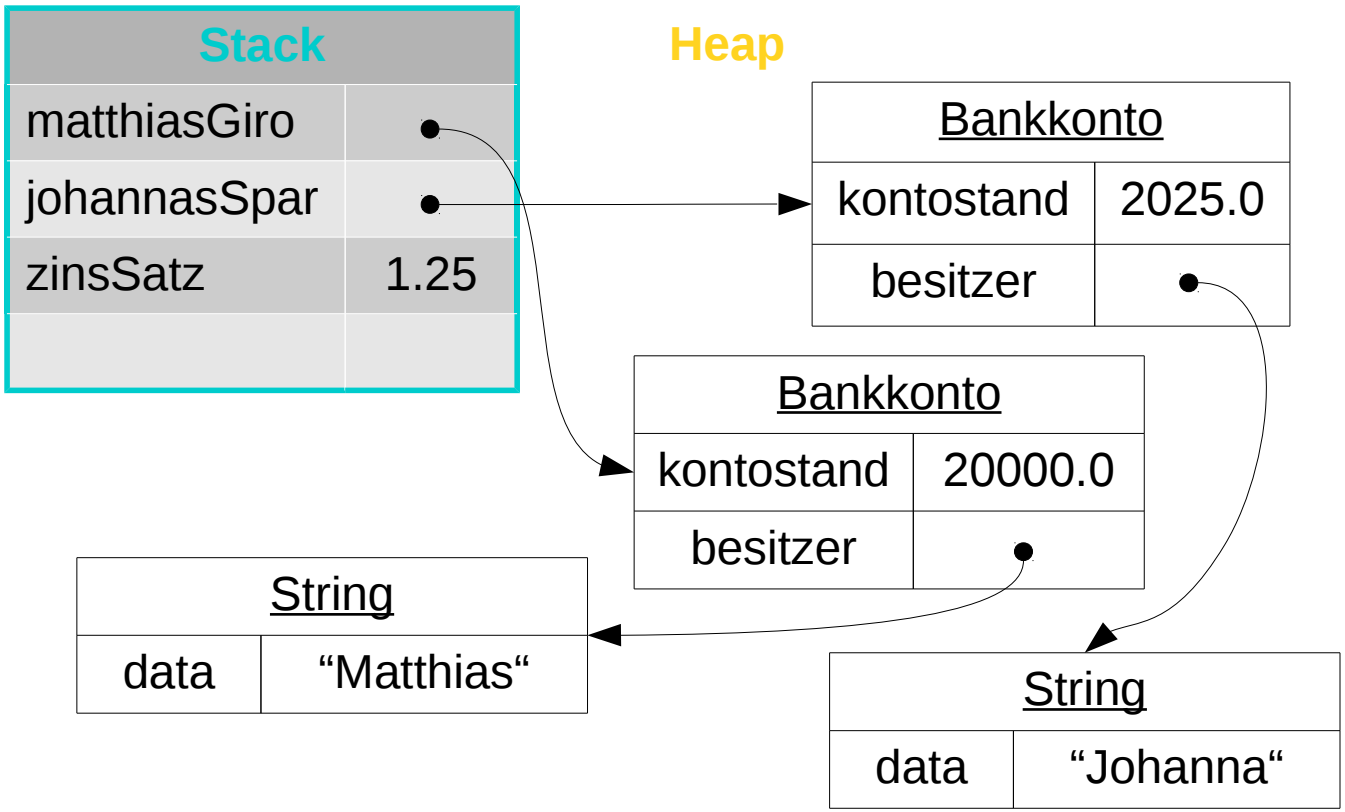
UML Objektdiagramm

Objektdiagramme sind ähnlich zu den in unseren Übungen verwendeten Speicherdiagrammen:

- Alle Objekte werden als zweigeteilte Rechtecke dargestellt.
- In der oberen Hälfte steht der Instanzname (falls vorhanden) und davon mit Doppelpunkt abgetrennt der unterstrichene Klassenname.
- In der unteren Hälfte steht *eine Auswahl* von Attributen des Objekts mit ihren Werten.
- Anstelle von Pfeilen wird eine einfache Linie verwendet, an deren Spitze der Name des Attributs steht.

Beispiel: Speicherdiagramm

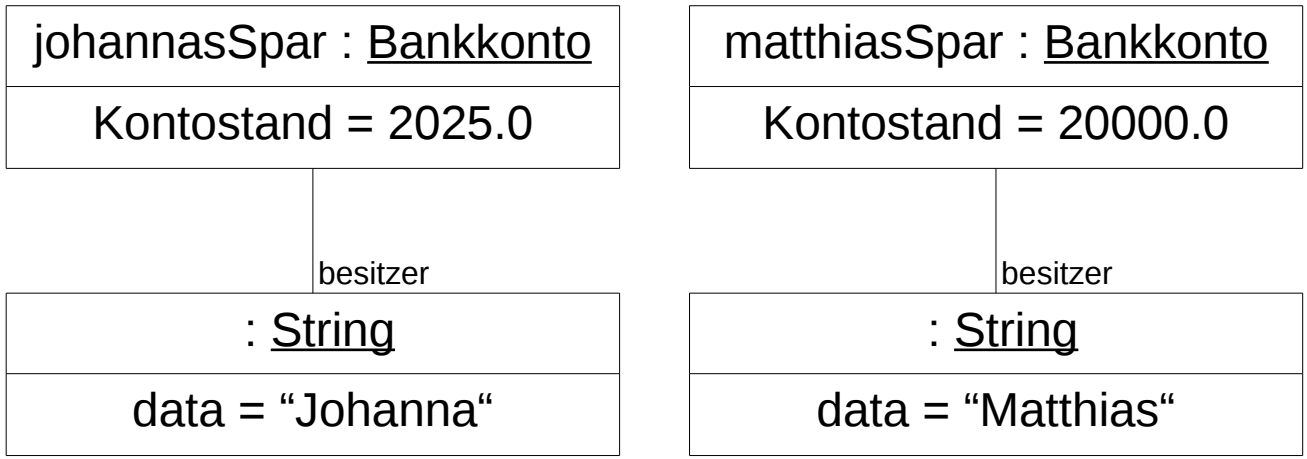
Nach Ende der Ausführung des Programmes von Folie 3.7 hatten wir in etwa folgendes Speicherdiagramm:



Verweise wurden durch einfach Pfeile dargestellt.
Alle lokalen Variablen des Stacks dargestellt.

Beispiel: Objektdiagramm

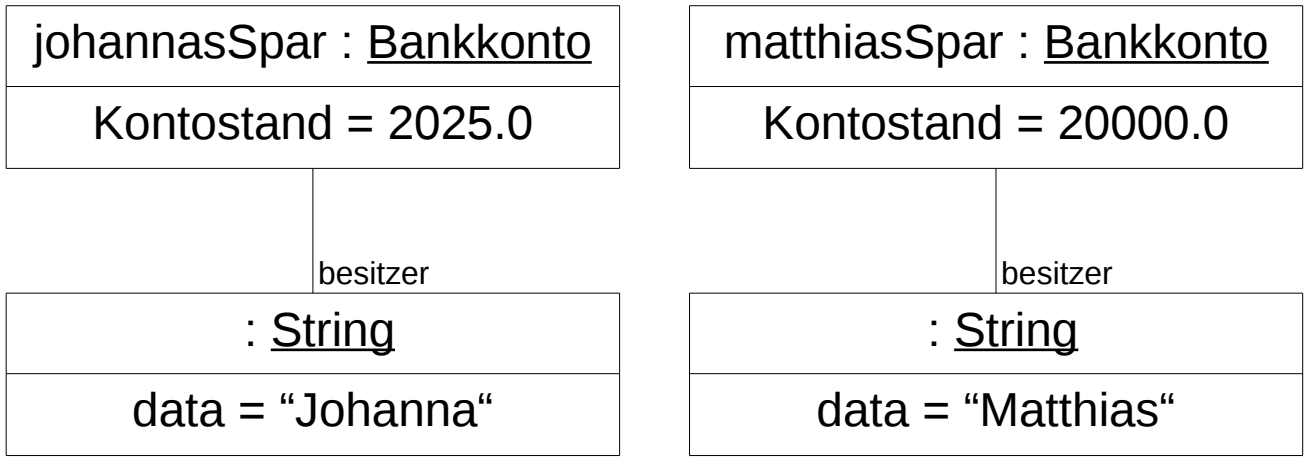
Der gleiche Programmzustand nun als UML-Objektdiagramm:



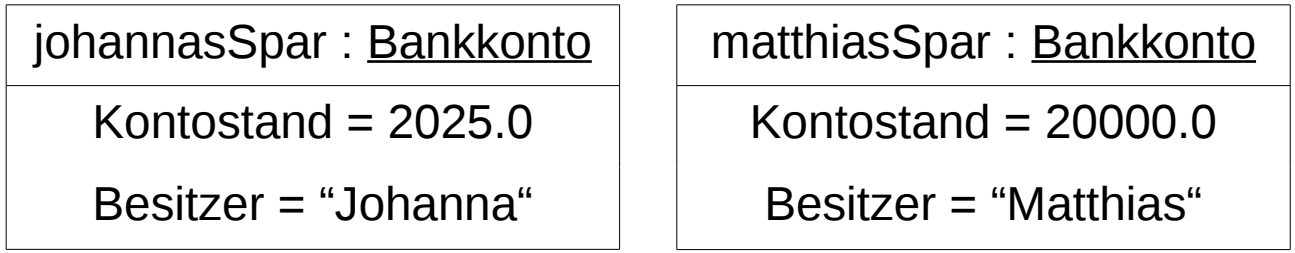
Es werden nur noch Objekte abgebildet.
Die Stack-Variable `zinsSatz` fehlt in dieser Darstellung.

Beispiel: Objektdiagramm

Der gleiche Programmzustand nun als UML-Objektdiagramm:



Strings werden aber meist nicht als eigene Objekte dargestellt:



Es werden nur noch Objekte abgebildet.
Die Stack-Variable `zinsSatz` fehlt in dieser Darstellung.

Klassendiagramme

Klassen werden in UML als dreigeteilte Rechtecke dargestellt:

1. Klassenname (unterstrichen oder fett gedruckt)
2. Instanzvariablen
3. Methoden (meist nur public)

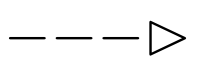
Instanzvariablen und Methoden werden manchmal mit ihrer Sichtbarkeit gekennzeichnet:

- $+$ public
- $-$ private
- $\sim$ package private
- $\#$ protected (nur für Erben sichtbar)

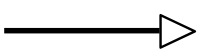
Ob man private Instanzvariablen und/oder Methoden angibt, hängt vom Anwendungsfall ab. Meist ist es zweckmäßig, alle Instanzvariablen und nur die öffentlichen Methoden anzugeben.

Beziehungen im Klassendiagramm

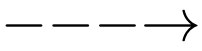
Beziehungen zwischen Klassen werden durch Linien und Pfeile mit verschiedenen Spitzen dargestellt:



Implementierung
von Klasse zu implementierter Schnittstelle (Interface)



Vererbung, „ist-ein“
von Unterklasse zu Oberklasse



Verwendung, „benutzt“
zu Klasse, deren Methoden aufgerufen werden

Aggregation, “im-Besitz-von” mehrere Besitzer möglich
Klasse an Rautenspitze besitzt Instanzvariable davon

Komposition, “ist-Teil-von”
besitzt exklusive Instanzvariable, d.h. Lebensspanne wird von Element an ausgefüllter Rautenspitze bestimmt

UML Notationen

In UML zeichnet man z.B. von der Klasse zur Schnittstelle einen gestrichelten Pfeil mit dreieckiger, hohler Spitze: −−−−▷

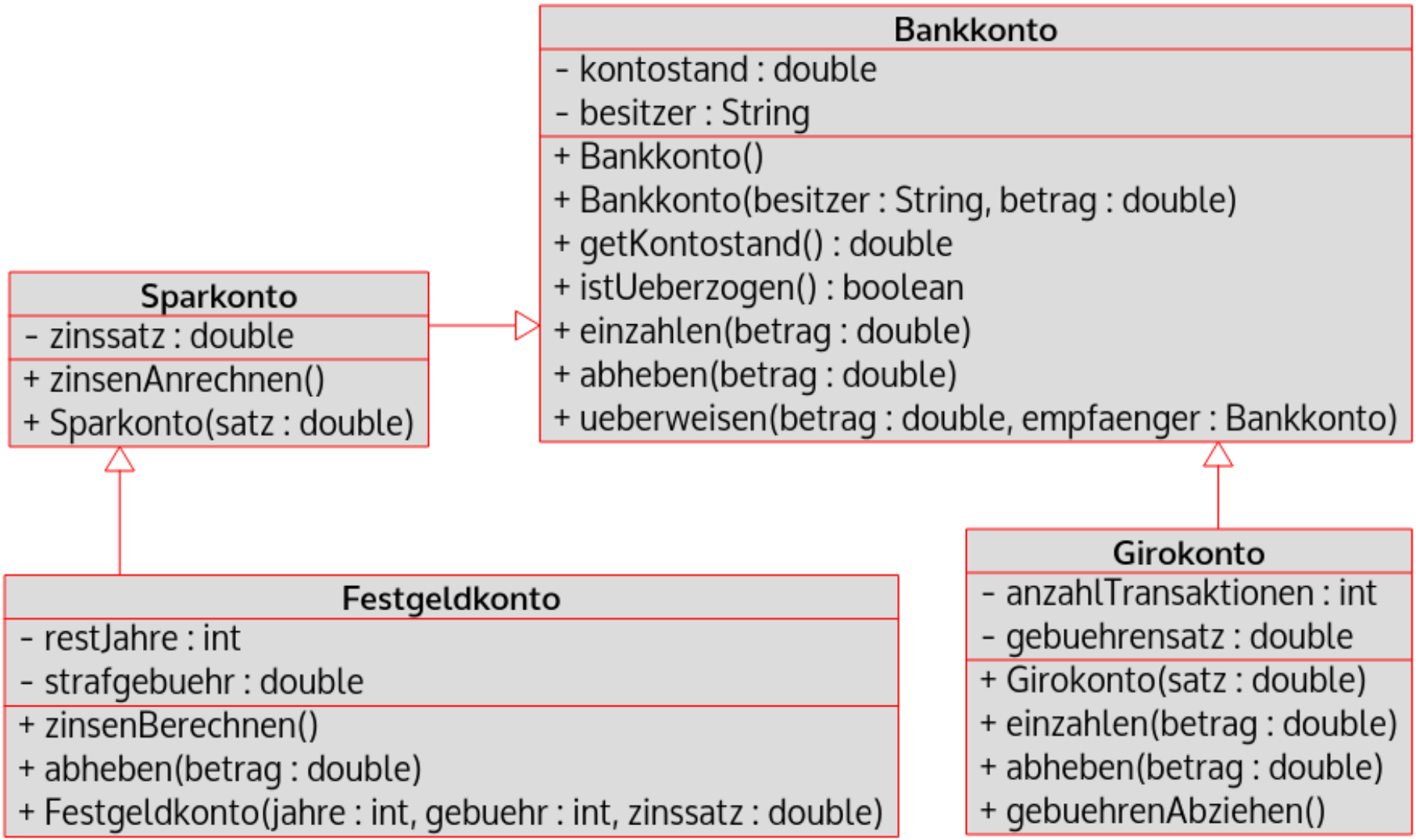
Da es im Diagramm nicht immer so einen Pfeil geben muss, wird die Schnittstelle selbst vor dem Namen auch noch mit dem Schlüsselwort `<<interface>>` gekennzeichnet.

Bemerkung:
UML ist nicht speziell für Java erdacht worden, sondern soll allgemein für alle Programmiersprachen verwendet werden.

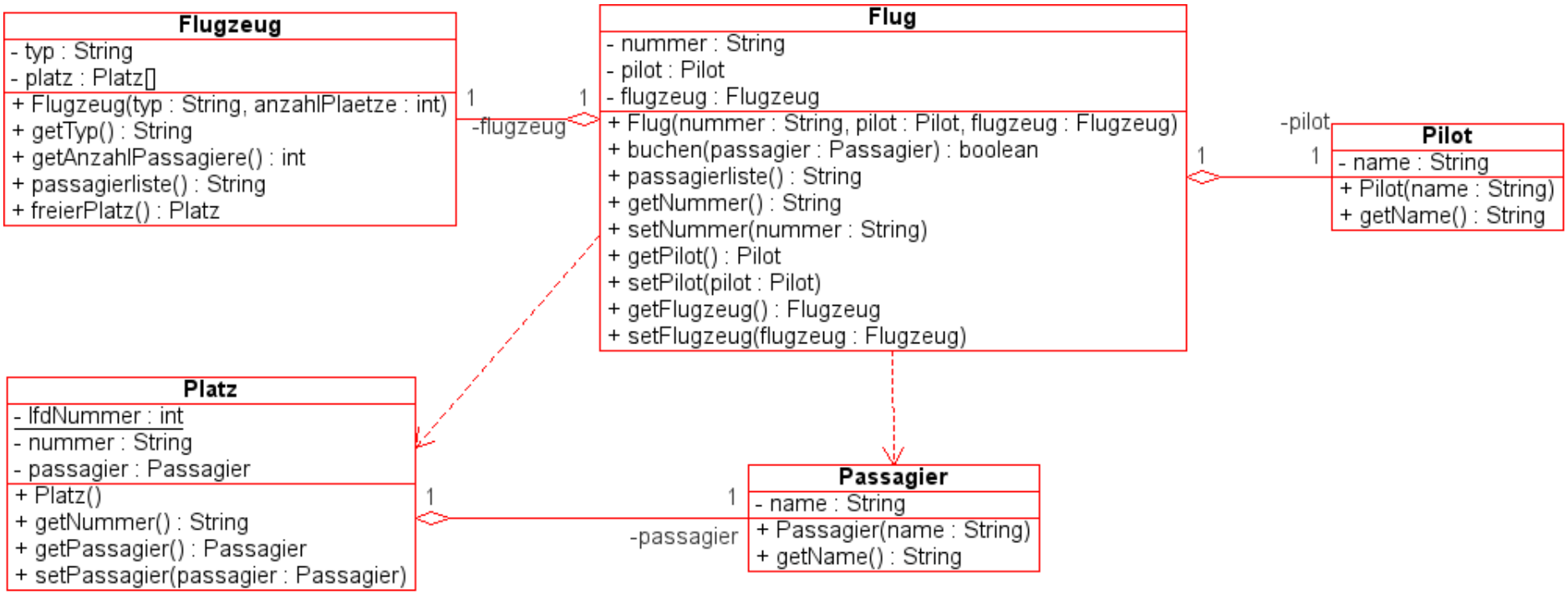
Entsprechend kann die Auslegung der UML Spezifikation je nach verwendeter Sprache etwas variieren. Für Interfaces in Java ist z.B. auch noch die alternative „Ball“-Notation gebräuchlich: Neben die Klasse zeichnet man eine Linie, welche in einem Kreis endet, unter dem einfach nur der Name des Interface steht.

Beispiel: UML Klassendiagramm I

Ein Pfeil mit hohler Dreieckspitze $\multimap$ von Unterklasse zu Oberklasse zeigt Vererbung im UML-Klassendiagramm an:

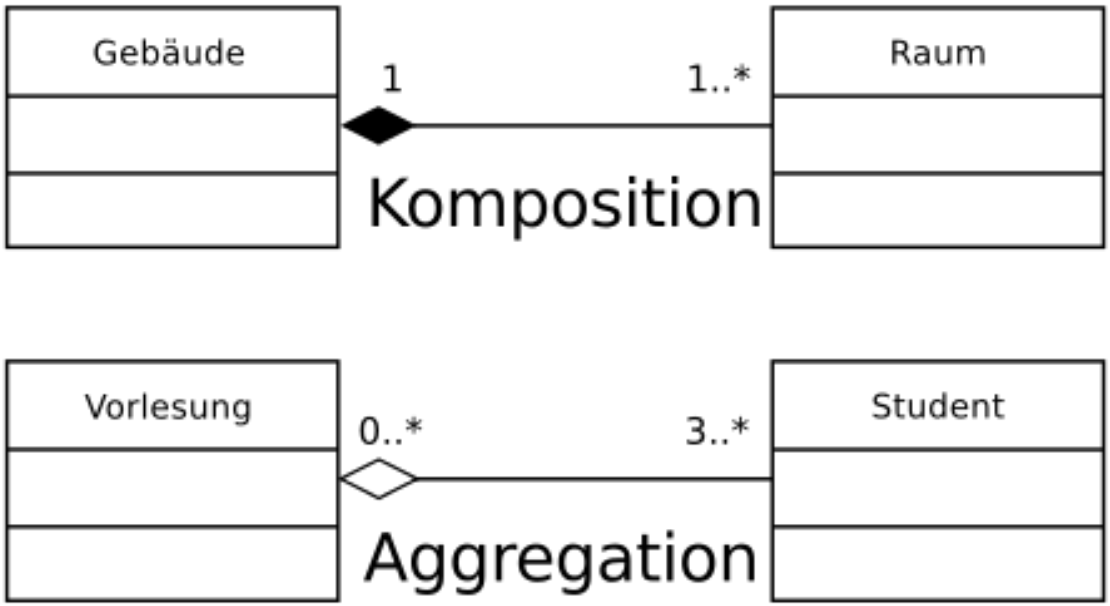


Beispiel: UML Klassendiagramm II



Aggregation vs Komposition

Der Unterschied zwischen Aggregation und Komposition liegt darin, ob der Teil unabhängig vom Besitzer der Instanz existieren kann:

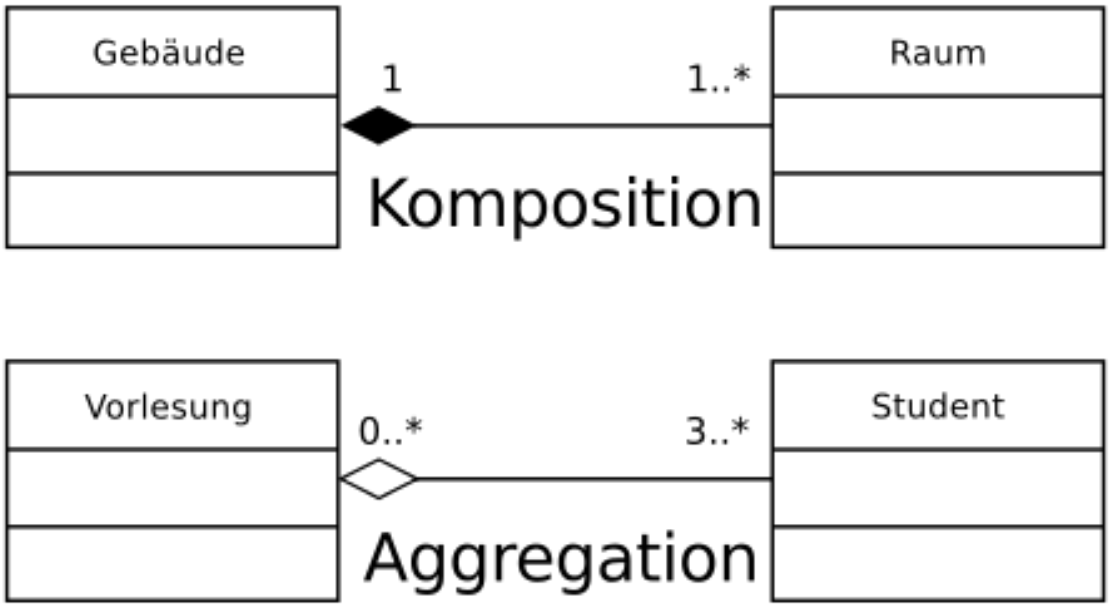


Quelle: Wikimedia, gemeinfrei

Kein Raum kann ohne Gebäude existieren, d.h. jeder Raum wird von einem Gebäude-Objekt als Attribut besessen, damit ist die Lebensspanne beider Objekte gleich.
Dagegen kann ein Student meist auch ohne Vorlesung gut leben; ein Student-Objekt wird also noch woanders verwendet.

Aggregation vs Komposition

Der Unterschied zwischen Aggregation und Komposition liegt darin, ob der Teil unabhängig vom Besitzer der Instanz existieren kann:



Quelle: Wikimedia, gemeinfrei

Am Pfeil kennzeichnet man oft noch die **Multiplizität**: Ein Gebäude hat mindestens einen Raum; jeder Raum hat genau ein Gebäude. Eine Vorlesung muss mindestens 3 Studenten haben; ein Student kann beliebig viele Vorlesungen hören, muss aber nicht.

Zusammenfassung UML

- UML Diagramme dienen der grafischen Veranschaulichung *während* des Softwareentwicklungsprozesses.
- UML-Klassendiagramme veranschaulichen Klassen und ihre Beziehungen/Abhängigkeiten untereinander.
- UML-Objektdiagramme veranschaulichen einen ausgewählten Teil des Zustands des Speichers zu einem speziellen Zeitpunkt während des Programmablaufs.
- UML Diagramme sollten sich immer nur auf eine Auswahl bzw. auf einen Teilaspekt des Projektes beschränken. Ein Diagramm des gesamten Projektes ist selten hilfreich.

Einführung in die Programmierung mit Java

Teil 10: Ausnahmebehandlung

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

7. Dezember 2017

Teil 10: Ausnahmebehandlung

Fehlerhafte Software

Software kann aus vielen Gründen unerwünschtes Verhalten zeigen:

- *Fehler beim Entwurf*, d.h. bei der Modellierung des Problems
- *Fehler bei der Programmierung* des Entwurfs
  - Programm bricht mit `NullPointerException` ab
  - Algorithmen falsch implementiert
  - ...
- *Ungenügender Umgang mit außergewöhnlichen Situationen*
  - fehlerhafte Benutzereingaben, z.B. Datum 31.11.2017
  - Abbruch der Netzwerkverbindung
  - Dateien können nicht gefunden werden
  - ...

Fehler vs Ausnahme

Ausnahmesituationen unterscheiden sich von **Programmierfehlern** darin, dass man sie nicht (zumindest prinzipiell) von vornherein ausschließen kann.

Immer möglich sind zum Beispiel:

- unerwartete oder ungültige Benutzereingaben
- Ein- und Ausgabe-Fehler beim Zugriff auf Dateien oder Netzwerk
- ...

Programmierfehler, welche nicht auftreten sollten, z.B.:

- Quadratwurzel einer negativen Zahl bestimmen
- Division durch 0
- Methodenaufruf mit einem Null-Pointer
- ...

Beispiel

```
public static int fakt(int x) {
  if (x < 0) {
    throw new IllegalArgumentException
                        ("fakt: Negatives Argument " + x);
  }
    else { ... }
}
```

Die Ausführung von `System.out.println(fakt(-2))` führt zu

```
Exception in thread "main" \
java.lang.IllegalArgumentException: \
fakt: Negatives Argument -2
        at Fakt.fakt(Fakt.java:7)
        at Fakt.main(Fakt.java:3)

Process Fakt exited abnormally with code 1
```

Was passiert hier?

Werfen einer Ausnahme mit dem Statement

`throw` ⟨*exp*⟩`;`

wobei ⟨*exp*⟩ ein Ausdruck des Typs `Throwable` ist.
Im Beispiel `new IllegalArgumentException("...")`

- Das Werfen der Ausnahme bricht die Programmabarbeitung sofort ab.
- Es kommt zu einer Fehlermeldung aus der die Art der geworfenen Ausnahme, ihr `String`-Parameter, sowie der Ort (Klasse, Methode, Programmzeile) ihres Auftretens hervorgeht.
- `IllegalArgumentException` ist eine Klasse, welche einen Konstruktor besitzt, der einen `String` als Argument erwartet. Davon wird mit `new` wie gewohnt ein Objekt erzeugt.

Vordefinierte Ausnahmen

- Alle Ausnahmen gehören zur (vordefinierten) Klasse `Exception` oder einer Unterklasse.
- `Exception` selbst ist Unterklasse von `Throwable`. Das "Argument" der `throw`-Anweisung muss `Throwable` sein.
- Unterklassen von `Exception` sind `IOException`, `ClassNotFoundException`, `CloneNotSupportedException`, `RuntimeException`, u.v.a.m.
- Unterklassen von `IOException` sind `EOFException` und `FileNotFoundException` u.v.a.m.
- Unterklassen von `RuntimeException` sind `IllegalArgumentException`, `IndexOutOfBoundsException` u.v.a.m.
- Unterklasse von `IllegalArgumentException` ist z.B.: `NumberFormatException`

Deklaration eigener Ausnahmen

```
public static int fakt(int x) {
    if (x < 0) {
        throw new FakultaetAusnahme
                        ("fakt: Negatives Argument " + x);
    } else { ... }
}

public class FakultaetAusnahme extends Exception {
    public FakultaetAusnahme() {}
    public FakultaetAusnahme(String grund) {
        super(grund);
    }
}
```

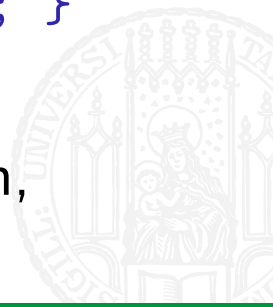
Ausnahmen sind normale Objekte von gewöhnlich deklarierten Klassen, welche Erben von `Exception` sind.

Deklaration eigener Ausnahmen

```
...
double foo = ...;
throw new MeineAusname(42, "Grund", foo);
...

public class MeineAusnahme extends Exception {
    private int    iFehler;
    private double dFehler;
    public FakultaetAusnahme(int i, String grund, double d) {
          super(grund);
          this.iFehler = i;
          this.dFehler = d;
    }
    public int    getIFehler() { return this.iFehler; }
    public double getDFehler() { return this.dFehler; }
}
```

Ausnahme-Objekte können zusätzliche Informationen speichern, die beim Auffangen des Fehlers ausgewertet werden kann.



Abfangen von Ausnahmen

Mit `try catch` werden Ausnahmen abgefangen.

```
public static void main(String[] args) {
    Scanner  konsole = new Scanner(System.in);
    try {
        System.out.println("Bitte die Zahl");
        String input = konsole.nextLine();
        int zahl = Integer.parseInt(input);
        System.out.println("Fakt("+zahl+")="+fakt(zahl));
    }
    catch (NumberFormatException ausnahme) {
        System.out.println("Falsche Eingabe.");
    }
    catch (IllegalArgumentException e) {
        System.out.print(e.getMessage());
    }
}
```

Abfangen von Ausnahmen

⟨*statement*⟩ ::= ··· | ⟨*try*⟩ | ···
⟨*try*⟩ ::= `try {`(⟨*statement*⟩)$^+$`}` (⟨*catch-clause*⟩)* [`finally {`⟨*statement*⟩`}`]
⟨*catch-clause*⟩ ::= `catch (` ⟨*exception-class*⟩ ⟨*variable*⟩ `) {`⟨*statement*⟩`}`

- Wird im `try` Block eine Ausnahme geworfen, so werden der Reihe nach alle `catch` Blöcke (**handler**) durchgegangen.
- Der erster Handler, dessen Klasse zur Ausnahme passt, kommt zur Anwendung. Das Ausnahmeobjekt wird an den Parameter gebunden und der entsprechende Code wird ausgeführt. Dabei kann erneut eine Ausnahme geworfen werden.
- Man sollte darauf achten, dass Handler die Ausnahmen sinnvoll bearbeiten!
  NIE sinnvoll: `catch (Exception e) {}`
- Jede Ausnahme versteht die Methode `printStackTrace()`. Dies gibt die Folge der Methodenaufrufe bis zu ihrer Auslösung auf der Konsole aus.

Abfangen von Ausnahmen

⟨*statement*⟩ ::= ··· | ⟨*try*⟩ | ···
⟨*try*⟩ ::= `try {`(⟨*statement*⟩)$^+$`}` (⟨*catch-clause*⟩)* [`finally {`⟨*statement*⟩`}`]
⟨*catch-clause*⟩ ::= `catch (` ⟨*exception-class*⟩ ⟨*variable*⟩ `) {`⟨*statement*⟩`}`

- Wird im `try` Block eine Ausnahme geworfen, so werden der Reihe nach alle `catch` Blöcke (**handler**) durchgegangen.
- Der erster Handler, dessen Klasse zur Ausnahme passt, kommt zur Anwendung. Das Ausnahmeobjekt wird an den Parameter gebunden und der entsprechende Code wird ausgeführt. Dabei kann erneut eine Ausnahme geworfen werden.
- Man sollte darauf achten, dass Handler die Ausnahmen sinnvoll bearbeiten!
  NIE sinnvoll: `catch (Throwable t) {}`
- Jede Ausnahme versteht die Methode `printStackTrace()`. Dies gibt die Folge der Methodenaufrufe bis zu ihrer Auslösung auf der Konsole aus.

Abfangen von Ausnahmen

Subtyping beachten! Ausnahmen in sinnvolle Hierarchie gliedern!

⟨*statement*⟩ ::= ··· | ⟨*try*⟩ | ···
⟨*try*⟩ ::= `try {`(⟨*statement*⟩)$^+$`}` (⟨*catch-clause*⟩)* [`finally {`⟨*statement*⟩`}`]
⟨*catch-clause*⟩ ::= `catch (` ⟨*exception-class*⟩ ⟨*variable*⟩ `) {`⟨*statement*⟩`}`

- Wird im `try` Block eine Ausnahme geworfen, so werden der Reihe nach alle `catch` Blöcke (**handler**) durchgegangen.
- Der erster Handler, dessen Klasse zur Ausnahme passt, kommt zur Anwendung. Das Ausnahmeobjekt wird an den Parameter gebunden und der entsprechende Code wird ausgeführt. Dabei kann erneut eine Ausnahme geworfen werden.
- Man sollte darauf achten, dass Handler die Ausnahmen sinnvoll bearbeiten!
  NIE sinnvoll: `catch (Exception e) {}`
- Jede Ausnahme versteht die Methode `printStackTrace()`. Dies gibt die Folge der Methodenaufrufe bis zu ihrer Auslösung auf der Konsole aus.

Zusatzinformationen auswerten

Damit wird klar, wie Zusatzinformationen genutzt werden können:

```
try {
  ...
  if (...) { throw new MeineAusname(xyz, "Grund", foo); }
  ...
} catch (MeineAusnahme ma) {
  System.out.println("Fehlercode " + ma.getIFehler);
  System.out.println(ma.getMessage + ma.getDFehler);
}

public class MeineAusnahme extends Exception {
    private int    iFehler;
    private double dFehler;
    public FakultaetAusnahme(int i, String grund, double d) {
          super(grund); this.iFehler = i; this.dFehler = d;
    }
    public int    getIFehler() { return this.iFehler; }
    public double getDFehler() { return this.dFehler; }
}
```

Die `finally`-Klausel

Es kann passieren, dass bestimmte Anweisungen auf jeden Fall durchgeführt werden müssen, auch wenn eine Ausnahme geworfen wird. Netzwerkverbindung schließen, Dateien speichern, etc.

Dafür gibt es das `finally`-Konstrukt: Es wird immer ausgeführt, egal ob vorher ein passender Handler ausgeführt wurde oder nicht.

Beispiel:

```
try {
   /* Alle Kombinationen der Reihe nach
      durchprobieren. */
}
finally {
   /* Abhauen */
}
```

Im [Horstmann] stehen geringfügig sinnvollere Beispiele.

Fangen und Kontrollfluss

Die wesentliche Idee hinter Ausnahmen ist die Möglichkeit, eine Ausnahme an einer anderen Stelle im Programm zu behandeln:

Beispiel:
Die View eines Spiels muss vom Spieler einen Spielzug erfragen. Die View weiß aber nicht, ob der Zug gültig ist.

Das Modell setzt den Spielzug um und merkt dabei, dass der Zug ungültig ist. Es wirft eine spezielle Ausnahme.

Der Controller fängt die Ausnahme und entscheidet, ob die Spieler neu befragt werden, oder ob ein Standardzug gespielt wird.

Fazit: Ohne Ausnahmen müsste das Modell hier immer einen Wert zurückgeben, ob alle Spielzüge gültig waren und wenn nein, welche Probleme aufgetreten sind.

Ausnahmen kann man also als *zusätzliche Rückgabewerte* auffassen, welche implizit „nach Oben“ durchgereicht werden.

Fangen und Kontrollfluss

Die wesentliche Idee hinter Ausnahmen ist die Möglichkeit, eine Ausnahme an einer anderen Stelle im Programm zu behandeln:

Beispiel:

D
D
D
un
D
ne

Fa

Dieses Beispiel soll *nicht* sagen, dass man es so machen muss!

Wenn die Funktion des Modells, welche der Controller aufruft, ohnehin nur den Rückgabetyp `void` hat, dann kann man auch einfach diesen Rückgabetyp ändern und nutzen, anstatt eine Ausnahme zu werfen!

Ausnahmen sind eine bequeme Lösung, wenn man mehr als einen Rückgabetyp benötigt. Der Einsatz von Ausnahmen lässt sich immer vermeiden!

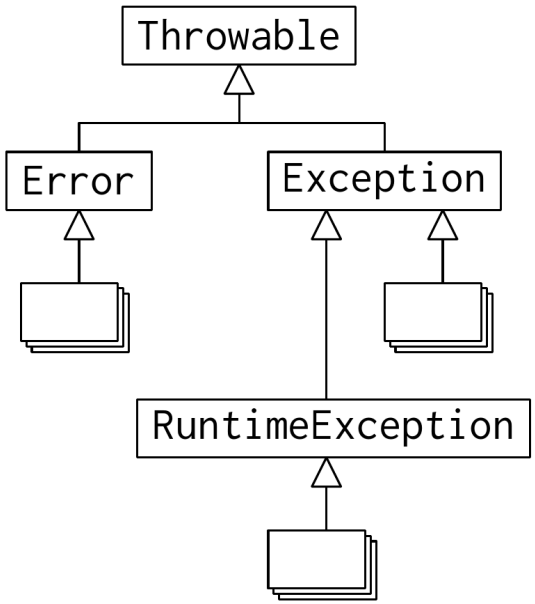
Wert zurückgeben, ob alle Spielzüge gültig waren und wenn nein, welche Probleme aufgetreten sind.

Ausnahmen kann man also als *zusätzliche Rückgabewerte* auffassen, welche implizit „nach Oben“ durchgereicht werden.

Fehler und Ausnahmen in Java

In Java werden verschiedene Arten von Ausnahmen und Fehlern durch verschiedene Unterklassen von `Throwable` repräsentiert:

- Instanzen von `Error`
  Unbehandelbare, unerwartete Fehler, z.B. Fehler der virtuellen Maschine
- Instanzen von `Exception`
  Vorhersehbare Ausnahmen, welche vom Programmierer zu behandeln sind
- Instanzen von `RuntimeException`
  Häufige Ausnahmen, die nicht behandelt werden müssen.



Alle Unterklassen von `Exception`, welche nicht auch Unterklassen von `RuntimeException` sind, werden als **geprüfte Ausnahmen** (engl. **checked exception**) bezeichnet.

Geprüfte Ausnahmen

Überprüfte Ausnahmen (checked exceptions) müssen entweder aufgefangen werden oder explizit als möglich deklariert werden:

```
import java.io.*;
...
    public void m() {
        throw new IOException("");
    }
```

Kompilieren führt zu folgender Fehlermeldung:

```
/home/mhofmann/work/teaching/EiP/Fakt.java:19:\
 unreported exception java.io.IOException; must \
be caught or declared to be thrown
        throw new IOException("");
        ^
```

Deklaration überprüfter Ausnahmen

Kann in einer Methode eine überprüfte Ausnahme auftreten und wird sie nicht aufgefangen, so muss sie mit `throws` in der Signatur deklariert werden:

```
import java.io.*;
...
    public void m() throws IOException{
        throw new IOException("");
    }
```

Eine Methode muss eine geprüfte Ausnahme auch dann deklarieren, wenn Sie diese nicht selbst wirft, sondern lediglich eine andere Methode aufruft, welche eine geprüfte Ausnahme werfen könnte.

Der Compiler *überprüft* für uns, dass jede geprüfte Ausnahme irgendwo gefangen wird – oder in der Signatur der `main`-Methode als möglich deklariert wurde.

Unchecked Exceptions

Ausnahmen, deren Auftreten von Zeit zu Zeit unvermeidlich ist und daher vom Programmierer erwartet werden sollen (z.B. `FileNotFoundException`) werden überprüft.

Ausnahmen, deren Auftreten auf einen Programmierfehler hindeutet, werden nicht überprüft: **unchecked exceptions**. (Unterklassen von `RuntimeException` und `Error`).

Prominentes Beispiel ist die `NullPointerException`: Auftreten ist in Java praktisch überall möglich. Es macht aber wenig Sinn, dies in jeder Methodensignatur hinzuschreiben.
Stattdessen ist es Aufgabe des Programmierteams, durch geeignete *Klasseninvarianten* oder *Laufzeitprüfungen* (`if (a==null)` ...) sicherzustellen, dass diese nicht auftreten können.

Ausnahmen nicht missbrauchen

Ausnahmen sind ein relativ teurer Mechanismus, welcher nur in Ausnahmesituationen verwendet werden sollte.

So bitte nicht:

```
try {
  while (true) {
  ...
  if (something) { throw EndLoop(); }
} catch (EndLoop el) {}
```

Besser:

```
do {
  ...
} while (!something);
```

Mögliche Ausnahmen dokumentieren

Auch wenn ungeprüfte Ausnahmen nicht mit `throws` deklariert werden müssen, ist es gute Stil, mögliche ungeprüfte Ausnahmen im Javadoc zu dokumentieren.

Beispiel:

Im Javadoc der Methode `get` von `ArrayList` wird klar angegeben, dass die ungeprüfte Ausnahme `IndexOutOfBoundsException` geworfen wird, wenn auf einen nicht vorhandenen Index des Arrays zugegriffen wird.

```
/**
* Returns the element at the specified position in this list.
*
* @param index index of the element to return
* @return the element at the specified position in this list
* @throws IndexOutOfBoundsException {@inheritDoc}
*/
public E get(int index) { ... }
```

(aus dem Quelltext von `java.util.ArrayList`)

Das Singleton Pattern

Manchmal möchte man von einer Klasse nur eine einzige, eindeutige Instanz erzeugen. Es soll nicht möglich sein, mehr als ein Objekt diese Klasse zu erzeugen.

Beispiel

Ausnahmen bei der Chipkartenprogrammierung.
Chipkarten (JavaCard) haben keine Garbage Collection; man muss daher vermeiden, zuviele (Ausnahme-)Objekte zu erzeugen.

Idee

Klassenvariable speichert das Objekt. Ist die Referenz in der Klassenvariable `null`, dann wird das Objekt einmal erzeugt.

Durchführung

```
class Fehler extends Exception {
    private static Fehler instanz = null;
    private String extragrund;

    private Fehler() {}          // Konstruktor ist privat!!!

    public String getGrund() {
        return extragrund;
    }
    private void setGrund(String s) {
        this.extragrund = s;
    }
    public static Fehler getInstanz(String s) {
        if (instanz == null) instanz = new Fehler();
        instanz.setGrund(s);
        return instanz;
    }
}
```

Durchführung

```
class Fehler extends Exception {
    private static Fehler instanz = null;
    private String extragrund;

    private Fehler(String s) {
        this.extragrund = s;
    }
    public static Fehler getInstanz(String s) {
        if (instanz == null) instanz = new Fehler();
        instanz.setGrund(s);
        return instanz;
    }
}
```

`Throwable` vererbt bereits ein `String`-Attribut, welches mit `getMessage` ausgelesen werden kann.

`extragrund` ist hier nur zu Demonstrationszwecken eingeführt, wie ein Zustand gespeichert werden kann.

Durchführung

```
import javax.swing.*;
public class Anwendung {
    public static void main(String[] args) {
        try {
            String input = JOptionPane.showInputDialog
                                    ("Bitte die Zahl:");
            int n = Integer.parseInt(input);
            if (n < 0)
                throw Fehler.getInstanz("Negative Zahl " + n);
            System.out.println(n);
            System.exit(0);
        }
        catch (Fehler f) {
            System.out.println(f.getGrund());main(args);
}}}
```

Zusammenfassung

- Mit `throw` werden Ausnahmen geworfen.
- Eine nicht aufgefangene Ausnahme führt zum Programmabbruch.
- Ausnahmen sind Objekte von Unterklassen von `Exception`.
- Mit `catch` werden Ausnahmen abgefangen.
- Ein `finally` Block wird immer ausgeführt, auch wenn der vorangegangene `try`-Block eine Ausnahme wirft.
- Es gibt vordefinierte Ausnahmen und selbstdefinierte.
- Es gibt geprüfte und ungeprüfte Ausnahmen.
- Ausnahmen sinnvoll behandeln, niemals einfach ignorieren.
- Ausnahmen können als zusätzliche Rückgabewerte aufgefasst werden.
- Das Singleton Entwurfsmuster besteht darin, von einer Klasse nur einmal eine Instanz zu erzeugen.