

## Vorlesung



# Programmieren I und II (Unit zum Selbststudium)

## Unit 7

Konzepte objektorientierter Programmiersprachen  
Klassen und Objekte sowie Pakete und Exceptions

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

1

## Disclaimer



### Zur rechtlichen Lage an Hochschulen:

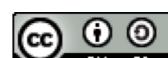
Dieses Handout und seine Inhalte sind durch den Autor selbst erstellt. Aus Gründen der Praktikabilität für Studierende lehnen sich die Inhalte stellenweise im Rahmen des Zitatrechts an Lehrwerken an.

Diese Lehrwerke sind explizit angegeben.

Abbildungen sind selber erstellt, als Zitate kenntlich gemacht oder unterliegen einer Lizenz die nicht die explizite Nennung vorsieht. Sollten Abbildungen in Einzelfällen aus Gründen der Praktikabilität nicht explizit als Zitate kenntlichgemacht sein, so ergibt sich die Herkunft immer aus ihrem Kontext: „Zum Nachlesen ...“.

### Creative Commons:

Und damit andere mit diesen Inhalten vernünftig arbeiten können, wird dieses Handout unter einer Creative Commons Attribution-ShareAlike Lizenz (CC BY-SA 4.0) bereitgestellt.



<https://creativecommons.org/licenses/by-sa/4.0>

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

2



## Prof. Dr. rer. nat. Nane Kratzke

*Praktische Informatik und  
betriebliche Informationssysteme*

- Raum: 17-0.10
- Tel.: 0451 300 5549
- Email: [nane.kratzke@th-luebeck.de](mailto:nane.kratzke@th-luebeck.de)



**@NaneKratzke**

Updates der Handouts auch über Twitter #prog\_inf und  
#prog\_itd

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

3

## Units

**Unit 1**  
Einleitung und  
Grundbegriffe

**Unit 2**  
Grundelemente  
imperativer Programme

**Unit 3**  
Selbstdefinierbare  
Datentypen und  
Collections

**Unit 4**  
Einfache I/O  
Programmierung

**Unit 5**  
Rekursive  
Programmierung und  
rekursive  
Datenstrukturen

**Unit 6**  
Einführung in die  
objektorientierte  
Programmierung und  
UML

**Unit 7**  
Konzepte  
objektorientierter  
Programmiersprachen  
(Selbststudium)

**Unit 8**  
Testen  
(objektorientierter)  
Programme

**Unit 9**  
Generische Datentypen

**Unit 10**  
Objektorientierter  
Entwurf und  
objektorientierte  
Designprinzipien

**Unit 11**  
Graphical User  
Interfaces

**Unit 12**  
Multithread  
Programmierung

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

4

## Abgedeckte Ziele dieser UNIT



Kennen existierender Programmierparadigmen und Laufzeitmodelle	Sicheres Anwenden grundlegender programmiersprachlicher Konzepte (Datentypen, Variable, Operatoren, Ausdrücke, Kontrollstrukturen)	Fähigkeit zur problemorientierten Definition und Nutzung von Routinen und Referenzen (insbesondere Liste, Stack, Mapping)	Verstehen des Unterschieds zwischen Werte- und Referenzsemantik
Kennen und Anwenden des Prinzips der rekursiven Programmierung und rekursiver Datenstrukturen	Kennen des Algorithmusbegriffs, Implementieren einfacher Algorithmen	Kennen objektorientierter Konzepte Datenkapselung, Polymorphie und Vererbung	Sicheres Anwenden programmiersprachlicher Konzepte der Objektorientierung (Klassen und Objekte, Schnittstellen und Generics, Streams, GUI und MVC)
Kennen von UML Klassendiagrammen, sicheres Übersetzen von UML Klassendiagrammen in Java (und von Java in UML)	Kennen der Grenzen des Testens von Software und erste Erfahrungen im Testen (objektorientierter) Software	Sammeln erster Erfahrungen in der Anwendung objektorientierter Entwurfsprinzipien	Sammeln von Erfahrungen mit weiteren Programmiermodellen und -paradigmen, insbesondere Multithread Programmierung sowie funktionale Programmierung

Am Beispiel der Sprache JAVA

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

5

## Themen dieser Unit

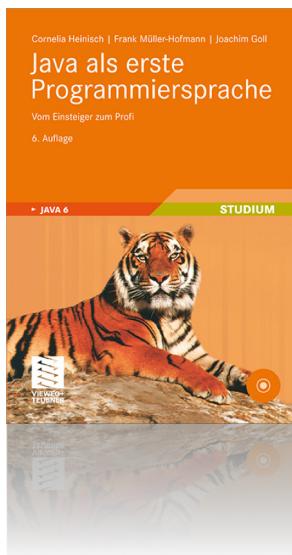


<b>Objekte</b> <ul style="list-style-type: none"><li>• Zugriffsmodifikatoren</li><li>• Initialisierung</li><li>• Instantiierung</li><li>• <code>this</code> Referenz</li><li>• Klasse <code>object</code></li></ul>	<b>Klassen</b> <ul style="list-style-type: none"><li>• Klassen (Gruppen gleichartiger Objekte)</li><li>• Klassenhierarchien</li><li>• Schnittstellen</li></ul>	<b>Pakete</b> <ul style="list-style-type: none"><li>• Programming in the small/large</li><li>• Zugriffsmodifikatoren</li><li>• Benennung</li></ul>	<b>Exceptions</b> <ul style="list-style-type: none"><li>• Checked und Unchecked</li><li>• Fangen und Behandeln</li><li>• Ausnahmen werfen/propagieren</li></ul>
---	--	--	---

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

6

## Zum Nachlesen ...



### Kapitel 10

#### Klassen und Objekte

- 10.1 Information Hiding
- 10.3 Die this-Referenz
- 10.4 Initialisierung von Datenfelder
- 10.5 Instantiierung von Klassen
- 10.7 Die Klasse Object

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

7

## Information Hiding



- Ein Ziel der Objektorientierung ist es, die Repräsentation der Daten und die Implementierung der Daten zu verbergen.
- Es soll kein Unbefugter die Daten verändern können.
- Nur Methoden des Objekts sollten auf die Daten des Objekts Zugriff haben.

Folgende Klasse ist zwar korrektes JAVA, befolgt aber nicht das Prinzip des Information Hiding.

```
class Person {  
    public String name;  
    public String nachname;  
    public int alter;  
  
    public void print() { ...  
        System.out.println(name);  
        System.out.println(nachname);  
        System.out.println(alter);  
    }  
}
```

Datenfelder des Objekts, sind von „außen“ zugreifbar und veränderbar.

```
Person p = new Person();  
p.name = „Max“;  
p.nachname = „Mustermann“;  
p.alter = „35“;  
p.print();
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

8

## Information Hiding (II)



„Objektorientierter“ wäre eine Realisierung, wie die folgende:

```
class Person {  
    private String name;  
    private String nachname;  
  
    public Person(String n, String nn) {  
        name = n; nachname = nn;  
    }  
  
    public void print() { ...  
        System.out.println(name);  
        System.out.println(nachname);  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public String getNachname() {  
        return nachname;  
    }  
}
```

- Somit kein direkter Zugriff mehr auf Datenfelder von Personenobjekten
- **private** ist ein sogenannter Zugriffsmodifikator

Da `name` und `nachname` als `private` deklariert wurden, können Sie nur innerhalb durch Objekte der Klasse `Person` geändert werden, nicht von außen.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

9

## Information Hiding (III) Zugriffsschutz für Methoden und Datenfelder



Objektorientierte Sprachen kennen üblicherweise die folgenden Zugriffsmodifikatoren

public

protected

private

Restriktivere Zugriffsrechte

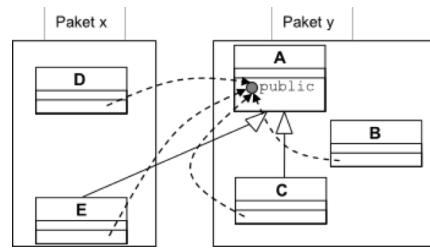
Zusätzlich gibt es noch den impliziten Zugriffsmodifikator `default`, der gilt, wenn keiner der drei oberen gesetzt wird. Darüberhinaus gibt es noch ein paar mehr Feinheiten im Zusammenhang mit Packages, diese werden aber erst in der Unit 9 behandelt.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

10

## Zugriffsschutz public

- Datenfelder und Methoden haben
- keinen Zugriffsschutz mehr
- Auf Datenfelder und Methoden kann aus allen Klassen zugegriffen werden



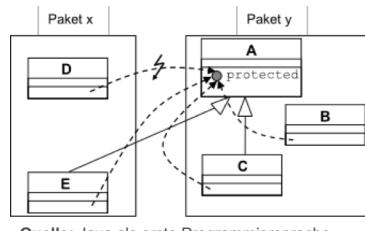
Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

11

## Zugriffsschutz protected

- Auf Datenfelder und Methoden kann
- aus derselben oder abgeleiteten Klassen zugegriffen werden
- (sowie von Klassen aus demselben Paket)



Quelle: Java als erste Programmiersprache

Auf Besonderheiten im Zusammenhang mit Paketen wird erst in Unit 9 eingegangen.

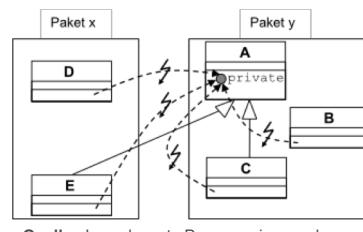
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

12

## Zugriffsschutz private



- Auf Datenfelder und Methoden kann
- nur aus derselben Klasse zugegriffen werden



Quelle: Java als erste Programmiersprache

Auf Besonderheiten im Zusammenhang mit Paketen wird erst in Unit 9 eingegangen.

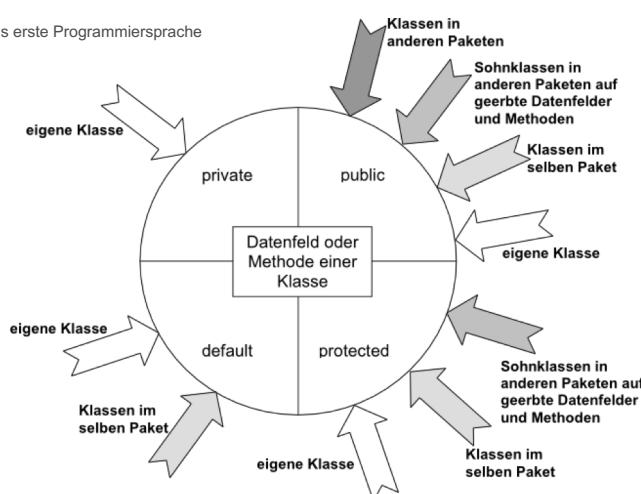
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

13

## Zugriffsschutz im Überblick



Quelle: Java als erste Programmiersprache



Auf Besonderheiten im Zusammenhang mit Paketen und dem Zugriffsmodifikator default wird erst in Unit 9 eingegangen.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

14

## Zugriffsmodifikatoren UML



Um die Zugriffsmodifikatoren  
 •public,  
 •protected,  
 •private und  
 •package/default  
 nicht immer in UML Diagrammen  
 ausschreiben zu müssen,  
 werden auch die folgenden  
 abkürzenden Symbole +, #, -, ~  
 genutzt.

### Example

```
+ public_datenfeld : Type
# protected_datenfeld : Type
- private_datenfeld : Type
~ package_datenfeld : Type

+ public_methode() : Type
# protected_methode() : Type
- private_methode() : Type
~ package_methode() : Type
```

Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

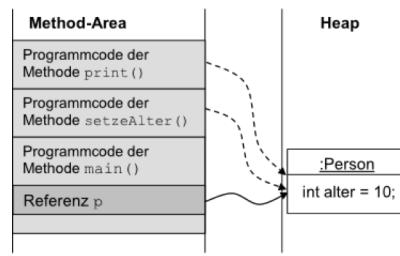
15

## Die this-Referenz (I)



- Jedes Objekt einer Klasse hat seine individuellen Instanzvariablen aber **dieselben** Methoden
- Aus Gründen der Speichereffizienz werden daher **Methoden an zentraler Stelle** im Speicher abgelegt
- **Datenfelder jedoch pro angelegtem Objekt** einer Klasse

Bei Ausführung eines Programms  
 liegen die **Methoden** in der  
 sogenannten **Method-Area** und  
 die **Objekte** auf dem sogenannten  
**Heap** Speicher



Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

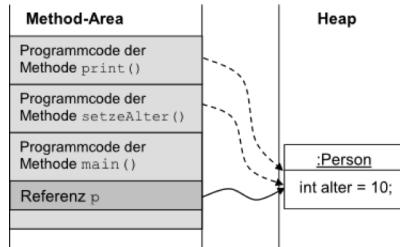
16

## Die this-Referenz (II)



Quelle: Java als erste Programmiersprache

Bei Ausführung eines Programms liegen die **Methoden** in der sogenannten **Method-Area** und die **Objekte** auf dem sogenannten **Heap Speicher**



- Daraus resultieren zwei Probleme:
  - **Problem 1:** Ein Objekt muss seine Klasse (und damit seine Methoden finden)
  - **Problem 2:** Methoden müssen das Objekt finden, für das sie aufgerufen werden.
- Problem 1 wird in Unit 7 (Stichwort: Polymorphie) behandelt werden.
- Problem 2 nun ...

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

17

## Die this-Referenz (III)



Jede Instanzmethode muss letztlich das Objekt kennen, für das es aufgerufen wird (um auf die Daten zugreifen zu können, die für dieses Objekt hinterlegt sind)

Ein Methode benötigt also zum Zeitpunkt der Abarbeitung eine Referenz auf das Objekt für das es aufgerufen wurde

Hierfür bräuchte die Methode eigentlich einen eigenen Parameter (JAVA realisiert dies – anders als bspw. Python – allerdings transparent für den Programmierer)

Dieser Parameter heißt **this** und ist in einer Instanzmethode immer eine Referenz auf das eigene Objekt

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

18

## This-Referenz (IV)



In folgenden Fällen ist die Nutzung der this Referenz erforderlich (Fall B, C oder D) oder üblich (nur Fall A).

### Benennung

- Fall A: Es soll deutlich gemacht werden, dass auf eine Instanzvariable oder -methode zugegriffen wird (dies ist nur Programmierstil und kein Erfordernis)
- Fall B: Ein Datenfeld des Objekts hat den gleichen Namen wie eine Methodenparameter (Parameter überdeckt Datenfeld)

### Objekt als Parameter oder Rückgabewert

- Fall C: Eine Referenz auf das aktuelle Objekt soll als Rückgabewert einer Methode zurückgegeben werden
- Fall D: Eine Referenz auf das aktuelle Objekt soll als Parameter an eine Methode übergeben werden

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

19

## This-Anwendungsfall A



### Benennung

- Fall A: Es soll deutlich gemacht werden, dass auf eine Instanzvariable oder -methode zugegriffen wird (dies ist nur Programmierstil und kein Erfordernis)
- Fall B: Ein Datenfeld des Objekts hat den gleichen Namen wie eine Methodenparameter (Parameter überdeckt Datenfeld)

```
class Person {  
    private String name;  
    private String nachname;  
    private int alter;  
  
    public void setName(String n) {  
        this.name = n;  
    }  
}
```

Hier ist die Nutzung von this eigentlich nur eine Frage des Stils und kein Erfordernis.  
Man hätte auch name = n schreiben können.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

20

## This-Anwendungsfall B

### Benennung

- Fall A: Es soll deutlich gemacht werden, dass auf eine Instanzvariable oder -methode zugegriffen wird (dies ist nur Programmierstil und kein Erfordernis)
- Fall B: Ein Datenfeld des Objekts hat den gleichen Namen wie eine Methodenparameter (Parameter überdeckt Datenfeld)

```
class Person {  
    private String name;  
    private String nachname;  
    private int alter;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Hier ist die Nutzung von `this` erforderlich, da der Parameter `name` das Datenfeld `name` in der Methode `setName()` überdeckt.

Um deutlich zu machen, dass man den Wert des Parameters `name` dem Datenfeld `name` des Objekts zuweisen will, muss man `this` nutzen.

## This-Anwendungsfall C

### Objekt als Parameter oder Rückgabewert

- Fall C: Eine Referenz auf das aktuelle Objekt soll als Rückgabewert einer Methode zurückgegeben werden
- Fall D: Eine Referenz auf das aktuelle Objekt soll als Parameter an eine Methode übergeben werden

```
class Person {  
    private String name;  
    private String nachname;  
    private int alter;  
  
    public Person setName(String n) {  
        name = n;  
        return this;  
    }  
}
```

Hier soll die Methode `setName()`, nach einer Änderung an internen Daten des Objekts eine Referenz auf das Objekt zurückgeben.

Dieser Stil wird manchmal genutzt, um mehrere Methodenaufrufe hintereinander auszuführen (method chaining).

## This-Anwendungsfall D

Objekt als  
Parameter  
oder  
Rückgabewert

- Fall C: Eine Referenz auf das aktuelle Objekt soll als Rückgabewert einer Methode zurückgegeben werden
- **Fall D: Eine Referenz auf das aktuelle Objekt soll als Parameter an eine Methode übergeben werden**

```
class Person {
    private String name;
    private String nachname;
    private int alter;

    public void print() {
        System.out.println(this);
    }
}
```

Hier soll die Methode print, das eigene Objekt mittels System.out.println() ausgeben.  
Hierzu muss der println Methode eine Referenz auf das eigene Objekt übergeben werden.

## Initialisierung von Datenfeldern

Datenfelder von Klassen und Objekten müssen mit Werten belegt werden. Die erste Zuweisung von Werten an Datenfelder wird Initialisierung genannt. JAVA kennt die folgenden Formen von Initialisierungen bei Datenfeldern:

Default-  
Initialisierungen von  
Datenfeldern

Manuelle  
Initialisierung

Initialisierungen mit  
einem  
Initialisierungsblock

Konstruktoren zur  
Initialisierung

Die einfachste Form der Initialisierung ist einfach keine Initialisierung von Datenfelder explizit vorzunehmen (anders als bei lokalen Variablen innerhalb einer Methode). Abhängig vom Typ des Datenfelds, erledigt dies JAVA dann automatisch.

Es wird jeweils das Datentyppendant zu „null“ dem Datenfeld zugewiesen.

Referenzdatentypen erhalten initial den Wert null, d.h. den Verweis auf **KEIN** Objekt.

Typ	Default-Wert
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0
float	0.0f
double	0.0d
Referenztyp	null

## Initialisierung von Datenfeldern

Default-Initialisierungen von Datenfeldern

Manuelle Initialisierung

Initialisierungen mit einem Initialisierungsblock

Konstruktoren zur Initialisierung

Sollen Datenfelder mit anderen Werten als den Default Werten initialisiert werden, kann dies explizit bei der Deklaration des Datenfelds angegeben werden:

```
class Person {  
    private String name = „Max“;  
    private String nachname = „Mustermann“;  
    private int alter; // Default Wert 0  
  
    private static anzPersonen = 1;  
    ...  
}
```

### Wichtig:

- Klassenvariablen werden **beim Laden der Klasse** einmalig initialisiert.
- Instanzvariablen werden **beim Erzeugen eines Objekts** pro Objekt initialisiert.

## Initialisierung von Datenfeldern

Default-Initialisierungen von Datenfeldern

Manuelle Initialisierung

Initialisierungen mit einem Initialisierungsblock

Konstruktoren zur Initialisierung

Datenfelder können auch in sogenannten **Initialisierungsblöcken** initialisiert werden. Solche Blöcke können beliebige Anweisungen beinhalten und sich auf **Klassen (static)** oder Objekte beziehen.

### Wichtig:

- Statische Initialisierungsblöcke werden **beim Laden der Klasse** einmalig ausgeführt.
- Nichtstatische Initialisierungsblöcke werden **beim Erzeugen eines Objekts** ausgeführt.

```
class Person {  
    private String name;  
    private String nachname;  
    private static int aufruf_initblock;  
  
    static { // Init für Klasse (static)  
        aufruf_initblock++;  
        System.out.println(„# Klasseninits:“ +  
            aufruf_initblock);  
    }  
}
```

## Initialisierung von Datenfeldern



Default-Initialisierungen von Datenfeldern

Manuelle Initialisierung

Initialisierungen mit einem Initialisierungsblock

Konstruktoren zur Initialisierung

Datenfelder können auch in sogenannten **Initialisierungsblöcken** initialisiert werden. Solche Blöcke können beliebige Anweisungen beinhalten und sich auf Klassen (static) oder **Objekte** beziehen.

**Wichtig:**

- Statische Initialisierungsblöcke werden **beim Laden der Klasse** einmalig ausgeführt.
- Nichtstatische Initialisierungsblöcke werden **beim Erzeugen eines Objekts** ausgeführt.

```
class Person {  
    private String name;  
    private String nachname;  
    private static anzahl;  
  
    { // Init für Objekte (kein static)  
        name = "Max";  
        nachname = "Mustermann";  
        System.out.println("Insgesamt " +  
            anzahl++ + " Personen erzeugt");  
    }  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

27

## Initialisierung von Datenfeldern



Default-Initialisierungen von Datenfeldern

Manuelle Initialisierung

Initialisierungen mit einem Initialisierungsblock

Konstruktoren zur Initialisierung

**Ein Konstruktor**

- dient zum Initialisieren eines Objekts
- wird ohne Rückgabewert deklariert (auch kein void)
- wird nicht an eine abgeleitete Klasse vererbt
- hat gleichen Namen wie Klasse
- gibt es ohne Parameter (default Konstruktor)
- Gibt es mit Parametern

```
class Person {  
    private String name;  
    private String nachname;  
  
    public Person() { // Default  
        name = "Max";  
        nachname = "Mustermann";  
    }  
  
    public Person(String vn, String nn) {  
        name = vn;  
        nachname = nn;  
    }  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

28

## Konstruktoren



- Der **Default-Konstruktor** ist ein Konstruktor ohne Parameter
- Der **voreingestellte Konstruktor** ist der vom Compiler zur Verfügung gestellte Konstruktor
- Der **selbst geschriebene Default-Konstruktor** ist ein Konstruktor ohne Parameter, der jedoch selbst geschrieben wurde
- **Konstruktoren mit Parametern** sind immer selbstgeschrieben und erlauben eine frei vorgebbare Initialisierung von Objekten

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

29

## Aufruf von Konstruktoren aus Konstruktoren (I)



Konstruktoren können

- andere Konstruktoren innerhalb derselben Klasse mittels **this()** aufrufen
- Konstruktoren der Basisklasse mittels **super()** aufrufen
- Diese Aufrufe müssen in JAVA als **erste Anweisung** im Konstruktor stehen

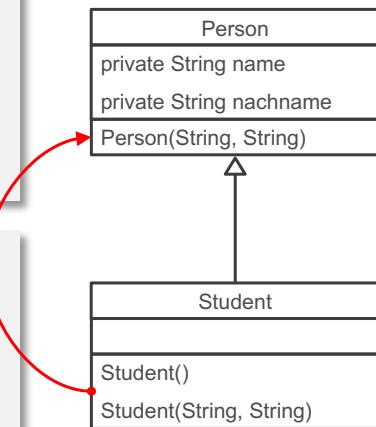
**Beispiel eines Konstruktorauftrags innerhalb derselben Klasse mittels this()**

```
class Person {  
    private String name;  
    private String nachname;  
  
    public Person() {  
        this("Max", "Mustermann");  
    }  
  
    public Person(String vn, String nn) {  
        name = vn;  
        nachname = nn;  
    }  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

30

## Aufruf von Konstruktoren aus Konstruktoren (II)



Beispiel eines Konstruktoraufrufs in der Basisklasse mittels super

```

class Person {
    private String name;
    private String nachname;

    public Person(String vn, String nn) {
        name = vn;
        nachname = nn;
    }
}

class Student extends Person {
    public Student() {
        super("Max", "Mustermann");
    }

    public Student(String vn, String nn) {
        super(vn, nn);
    }
}

```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

31



## Instantiierung von Klassen



- Erzeugen eines Objekts aus einer Klasse wird auch **Instantiierung** genannt.
- Es wird eine neue **Instanz** (Objekt) dieser Klasse geschaffen.
- Mittels des **new** Operators wird **Speicher** für das Objekt **allokiert**
- Durch Aufruf des Konstruktors wird dieser **Speicher initialisiert**

```
Person p = new Person();
```

Schritt 1: Referenzvariable p vom Typ Person wird angelegt

Schritt 2: durch new wird ein Objekt auf dem Heap erzeugt

Schritt 3: Für das Objekt wird die Initialisierungssequenz (vgl. vorherige Folie) durchgeführt

Schritt 4: Es wird eine Referenz auf das Objekt durch new erzeugt und der Variablen p zugewiesen

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

33

## Freigabe von Speicher erfolgt automatisch



### Speichermanagement

- nicht mehr referenzierte Objekte
- werden durch einen Garbage Collector automatisch freigegeben



```
Person p = new Person(); // Speicher  
// wird allokiert  
...  
p = null; // p wird nicht mehr referenziert  
// und kann damit durch den GC  
// freigegeben werden
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

34

## Die Klasse Object


  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**

- Jede Klasse wird implizit von der Klasse Object abgeleitet
- Damit beinhaltet jede Klasse automatische alle Methoden der Klasse Object

**Die wesentlichen Methoden, die in Object definiert sind:**

```
public String toString()
public boolean equals()
protected Object clone()
```

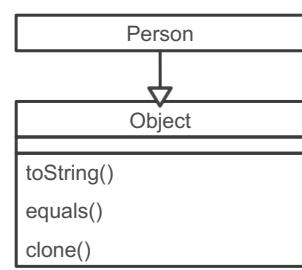
Schon  
behandelt

**D.h. jede Klassendefinition, wie z.B.**

```
class Person {
    ...
}
```

**ist eigentlich die Kurzform für ...**

```
class Person extends Object {
    ...
}
```



```

classDiagram
    class Person
    class Object
    Person -->|> Object
    Object <|-- toString()
    Object <|-- equals()
    Object <|-- clone()
  
```

Prof. Dr. rer. nat. Nane Kratzke | 35
   
 Praktische Informatik und betriebliche Informationssysteme

## Miniübung:


  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**





Gegeben ist folgende Klassendefinition.

```
class Time{
    private int hh = 23;
    private int mm;
    private int ss = 59;
    private String tz = "UTC";

    { this.tz = "MEZ"; }

    public Time() { hh = 0; }

    public Time(int h, int m) {
        hh = h; this.mm = m;
    }

    public Time(int h, int m, int s) {
        hh = h; mm = m; ss = s;
    }

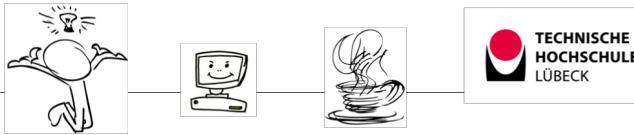
    public Time(int h, int m, String z) {
        hh = h; mm = m; tz = z;
    }
}
```

Geben Sie den Mikrozustand des erzeugten Objekts nach der Initialisierung an.

```
Time t = new Time(11, 23);
Time s = new Time(7, 59, 13);
Time r = new Time(0, 0, "ABC");
Time q = new Time();
```

Prof. Dr. rer. nat. Nane Kratzke | 36
   
 Praktische Informatik und betriebliche Informationssysteme

### Miniübung:



Gegeben ist folgende Klassendefinition.

```
class Auto {
    private double fuel = 0.0;
    private double kmstand = 0.0;

    public Auto() {
        this.fuel = 5.0;
    }

    public void tanke(double l) {
        this.fuel += l;
    }

    public void fahre(double km) {
        this.kmstand += km;
        this.fuel -= 7.0 * km / 100;
    }
}
```

Geben Sie nun sinnvolle getter und setter Methoden an, um den Kilometerstand und den Tankstand auslesen und setzen zu können. Achten Sie auf sinnvolle Zugriffsmodifikatoren!

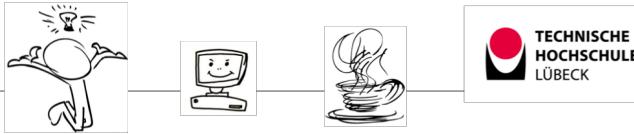
Tankstand

Kilometerstand

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

37

### Miniübung:



Gegeben ist folgende Klassendefinition plus gerade vorgenommener Ergänzungen.

```
class Auto {
    private double fuel = 0.0;
    private double kmstand = 0.0;

    public Auto() {
        this.fuel = 5.0;
    }

    public void tanke(double l) {
        this.fuel += l;
    }

    public void fahre(double km) {
        this.kmstand += km;
        this.fuel -= 7.0 * km / 100;
    }
}
```

Geben Sie nun eine sinnvolle Implementierung an, den Makrozustand hinsichtlich des Tankzustands (kaum noch Benzin) eines Autoobjekts zu bestimmen.

Kaum noch Benzin

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

38

### Miniübung:



Gegeben ist folgende Klassendefinition plus gerade vorgenommener Ergänzungen.

```
class Auto {  
    private double fuel = 0.0;  
    private double kmstand = 0.0;  
  
    public Auto() {  
        this.fuel = 5.0;  
    }  
  
    public void tanke(double l) {  
        this.fuel += l;  
    }  
  
    public void fahre(double km) {  
        this.kmstand += km;  
        this.fuel -= 7.0 * km / 100;  
    }  
}
```

Geben Sie nun eine sinnvolle Implementierung an, den Makrozustand hinsichtlich des Wartungsstands zu bestimmen (alle 20.000km zur Inspektion).

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

39

### Miniübung:



Geben Sie nun bitte die UML Notation der gerade definierten Klassen Auto und InspAuto an.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

40

## Zusammenfassung



- Zugriffsmodifikatoren und Information Hiding
  - public
  - protected
  - private
  - getter und setter Methoden
- **this Referenz**
  - Referenz auf das eigene Objekt
  - Parameterüberdeckung umgehen
- Initialisierung von Datenfelder
  - Default Initialisierung
  - Manuelle Initialisierung
  - Initialisierungsblöcke
  - Konstruktoren
- Instantiierung von Klassen
- Die „Mutter aller Klassen“ (Object)

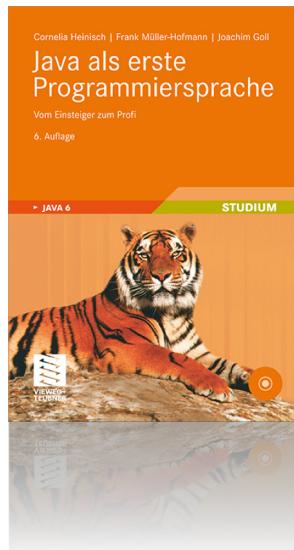


## Themen dieser Unit



Objekte	Klassen	Pakete	Exceptions
<ul style="list-style-type: none"><li>• Zugriffsmodifikatoren</li><li>• Initialisierung</li><li>• Instantiierung</li><li>• <b>this Referenz</b></li><li>• Klasse object</li></ul>	<ul style="list-style-type: none"><li>• Klassen (Gruppen gleichartiger Objekte)</li><li>• Klassenhierarchien</li><li>• Schnittstellen</li></ul>	<ul style="list-style-type: none"><li>• Programming in the small/large</li><li>• Zugriffsmodifikatoren</li><li>• Benennung</li></ul>	<ul style="list-style-type: none"><li>• Checked und Unchecked</li><li>• Fangen und Behandeln</li><li>• Ausnahmen werfen/propagieren</li></ul>

## Zum Nachlesen ...



### Kapitel 11

#### Vererbung

Das Konzept der Vererbung, Erweitern und Überschreiben, Besonderheiten bei der Vererbung

### Kapitel 14

#### Schnittstellen

Trennung von Spezifikation und Implementierung, Aufbau und Verwenden von Schnittstellen

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

43

## Wiederverwendung von Quellcode



Neben der Aggregation ist die Vererbung ein wesentliches Sprachmittel von OO-Sprachen, um Programmcode wiederzuverwenden.

Bei der Vererbung wird der Quellcode einer Superklasse in einer abgeleiteten Klasse wiederverwendet

Bei der Aggregation werden vorhandene Klassen von den aggregierenden Klassen genutzt und damit wiederverwendet.

OO ist sozusagen intelligentes Copy-Paste

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

44

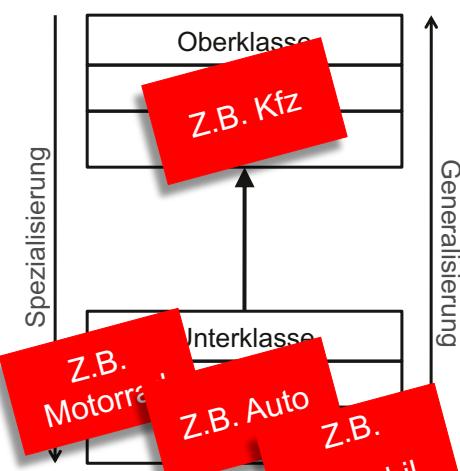
## Der Klassenbegriff



Klassen sind einerseits „Gruppierungen“ von gleichartigen Objekten

Zwischen Klassen können Vererbungsbeziehungen bestehen

- Unterklassen
- Oberklassen
- Unterklassen erben von Oberklassen



Prof. Dr. rer. nat. Nane Kratzke

45

## Konformität und Ersetzbarkeit von Klassen



### Konformität

- Jedes Exemplar einer Unterklasse ist auch ein Exemplar der Oberklasse
- Vererbung der Spezifikation
- Eine Unterklasse geht also alle Verpflichtungen (Kontrakte) der Oberklasse ein

### Ersetzbarkeit

- Jedes Exemplar einer Klasse muss deren Spezifikationen (Kontrakte) erfüllen
- Das gilt auch dann wenn das Objekt ein Exemplar einer Unterklasse ist.
- Unterklassen erben also alle Eigenschaften, Funktionalitäten, Beziehungen und Verantwortlichkeiten
- Prinzip: Objekte der Unterklassen können Objekte der Oberklassen ersetzen.

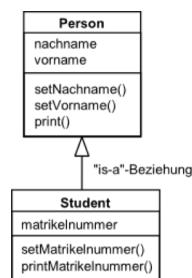
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

46

## Das Konzept der Vererbung



Darstellung einer Vererbung in **UML**. Der Ableitungspfeil zeigt von der Subklasse zu der Superklasse (kontraintuitiv)



**Superklasse** (auch Basis-, Ober- oder Vaterklasse genannt)

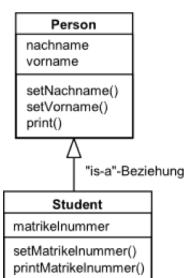
**Subklasse** (auch abgeleitete, Unter- oder Sohnklasse genannt)

- Eine Subklasse erbt alle Eigenschaften (Datenfelder und Methoden) der Superklasse
- Und fügt eigene Eigenschaften hinzu ohne die Eigenschaften der Superklasse wiederholen zu müssen (copy)
- **Beispiel:** Es gibt Personen und Studenten. Jeder Student ist auch immer eine Person.
- Was in Person bereits definiert wurde, muss für Studenten nicht wiederholt werden.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

47

## Vererbung UML und JAVA



```

class Person {
    private String vorname;
    private String nachname;

    public void setNachname(String nn) { nachname = nn; }

    public void setVorname(String vn) { vorname = vn; }

    public void print() {
        System.out.print("Name: ");
        System.out.println(vorname + " " + nachname);
    }
}

class Student extends Person {
    private int matrikelnummer;

    public void setMatrikelnummer(int mn) {
        matrikelnummer = mn;
    }

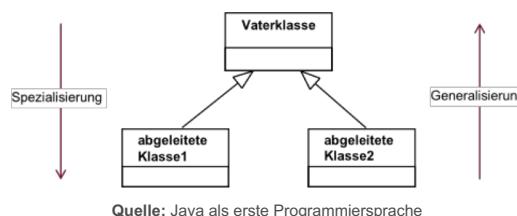
    public void printMatrikelnummer() {
        System.out.print("Matrikelnummer: ");
        System.out.println(matrikelnummer);
    }
}
  
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

48

## Spezialisierung und Generalisierung

- Vererbung erlaubt es automatisch alle Eigenschaften (Datenfelder und Methoden) einer Superklasse einer abgeleiteten Klasse bereitzustellen
- Die abgeleitete Klasse stellt eine **Spezialisierung** ihrer Vaterklasse dar (sie kann mehr und damit meist speziellere Dinge – Student ist ein spezieller Typ von Person)
- In der abgeleiteten Klasse sind nur die neuen spezifischen und zusätzlichen Eigenschaften festzulegen
- Umgekehrt stellt eine Vaterklasse eine **Generalisierung** ihrer abgeleiteten Klassen dar (das Konzept Person ist genereller als das Konzept Student)



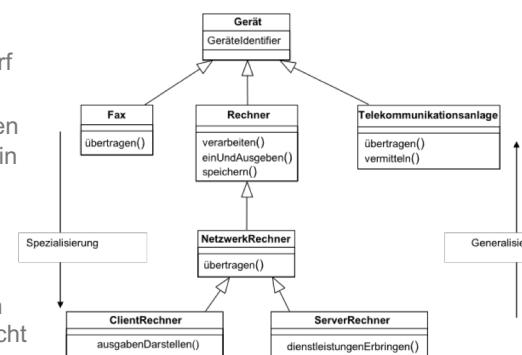
Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

49

## Wiederholungen vermeiden

- Mit dem Konzept der Vererbung können Wiederholungen im Entwurf vermieden werden
- Gemeinsame Eigenschaften mehrerer Klassen werden in gemeinsame Oberklassen ausgelagert
- In der imperativen Programmierung geht das mittels Routinen/Methoden zwar auch für Quelltext, nicht aber so ohne weiteres für Daten.

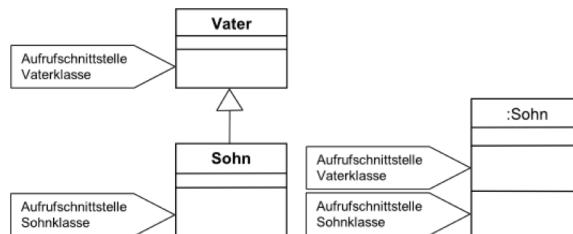


Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

50

## Erweitern von Methoden (und Datenfeldern) in einer Subklasse

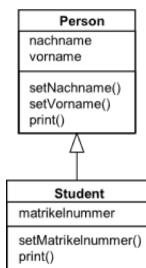


- Die **Methoden der Vaterklasse** werden im Rahmen der Vererbung von der Sohnklasse **unverändert übernommen**.
- Diese werden durch spezifische in der Sohnklasse definierte Methoden ergänzt.
- Ein Objekt der Sohnklasse besitzt also sowohl die **Aufrufsstelle der Vaterklasse** als auch die **Aufrufsstelle der Sohnklasse**.
- Zusätzlich werden Instanz- und Klassenvariablen vererbt.
- Eine Sohnklasse erbt also **alle Eigenschaften der Vaterklasse**.
- Nicht alle dieser Eigenschaften sind aber der Sohnklasse direkt zugänglich (vgl. Zugriffsmodifikatoren in Unit 3, insbesondere **private** Modifikator).

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

51

## Überschreiben von Methoden in einer Subklasse



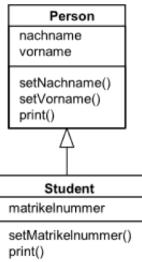
**UML:** Die Methode `print()` der Subklasse **Student überschreibt** die Methode `print()` der Superklasse `Person`.

- Methoden die in einer Vaterklasse definiert wurden, können in einer Sohnklasse überschrieben werden.
- D.h. der Code der Vaterklasse in der Methode wird sozusagen durch den Code der Sohnklasse ersetzt.
- Dabei müssen die formalen Parameter beider Methoden (der Methodenkopf oder die Methodensignatur) dieselbe Anzahl, denselben Typ und dieselbe Reihenfolge der Parameter sowie des Rückgabetyps haben.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

52

## Gründe für das Überschreiben von Methoden in einer Subklasse



**Überschreiben zur Verfeinerung**

- Z.B. weil zusätzliche Datenfelder in der Sohnklasse ausgewertet werden müssen, die in der Vaterklasse noch nicht bekannt waren.
- Weil Platzhalter in der Vaterklasse vorgesehen wurden, um spezifischere Funktionalität einzubetten (sogenannte abstrakte Klassen – Erläuterung folgt noch)

**Überschreiben zur Optimierung**

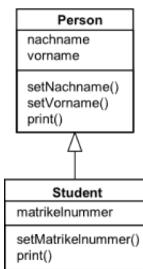
- In einer Sohnklasse müssen interne Datenstrukturen angepasst werden oder Algorithmen optimiert werden
- Bsp. Ersetzung des ineffizienten Suchalgorithmus Bubblesort durch Quicksort
- Außerverhalten der Schnittstelle darf sich aber nicht ändern!

**UML:** Die Methode print() der Subklasse Student **überschreibt** die Methode print() der Superklasse Person, um auch das zusätzliche Datenfeld matrikelnummer bei Studenten ausgeben zu können.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

53

## Überschreiben von Methoden UML und JAVA



```

class Person {
    private String vorname;
    private String nachname;

    public void setNachname(String nn) { nachname = nn; }

    public void setVorname(String vn) { vorname = vn; }

    public void print() {
        System.out.print("Name: ");
        System.out.println(vorname + " " + nachname);
    }
}

class Student extends Person {
    private int matrikelnummer;

    public void setMatrikelnummer(int mn) {
        matrikelnummer = mn;
    }

    public void print() {
        System.out.print("Name: ");
        System.out.println(vorname + " " + nachname);
        System.out.print("Matrikelnummer: ");
        System.out.println(matrikelnummer);
    }
}

```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

54

## Überschreiben von Methoden unter Rückgriff auf die ursprüngliche Funktionalität (I)

- Im gerade gezeigten Beispiel wurde `print()` überschrieben, um zusätzliche Datenfelder ausgeben zu können.
- Die ursprüngliche Logik musste dazu kopiert werden.
- Solche Fälle sollte man vermeiden – zudem kann aufgrund der gewählten Modifikatoren (`private`) nicht auf die auszugebenden Datenfelder in der Sohnklasse zugegriffen werden.
- Idealerweise muss also aus überschriebenen Methoden auf die Logik der Methoden der Superklassen zugegriffen werden können.

### Überschriebene Instanzmethoden

- Einer Basisklasse
- können mit `super.methode()`
- in der überschreibenden Instanzmethode aufgerufen werden

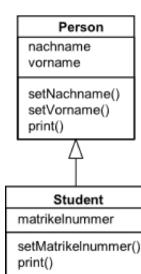
### Überschriebene Klassenmethoden

- einer bekannten Basisklasse
- werden einfach durch Angabe des Klassennamens
- mit `Klassenname.methode()` aufgerufen.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

55

## Überschreiben von Methoden unter Rückgriff auf die ursprüngliche Funktionalität (II)

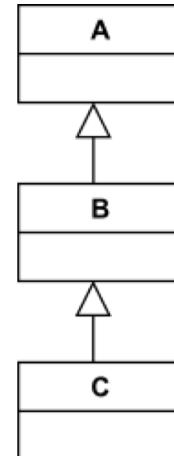


Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

56

## Besonderheiten bei der Vererbung

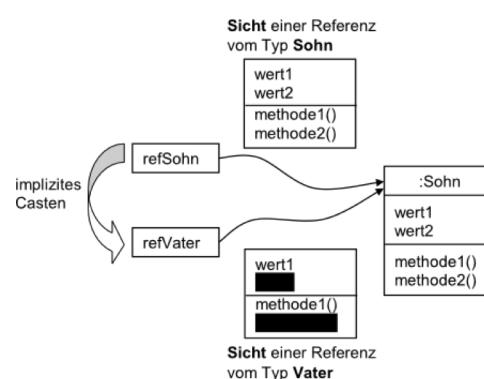
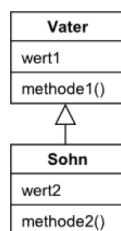
- Für den Einsatz der Vererbung muss man Kenntnisse über die Typkonvertierungen haben
- Wichtig: Ein Sohnobjekt ist immer vom Typ der eigenen Klasse, als auch vom Typ der Vaterklasse, der Vatervaterklasse, etc.
- Somit kann ein Objekt durchaus mehrere Typen haben.



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

57

## Implizites „Upcasten“



```

Sohn s = new Sohn();
Vater v = s;
  
```

Die Referenz vom Typ Sohn sieht das gesamte Objekt, die vom Typ Vater sieht nur die Vateranteile

s.wert1  
s.wert2  
s.methode1()  
s.methode2()

v.wert1  
v.wert2  
v.methode1()  
v.methode2()

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

58

## Explizites „Downcasten“

**TECHNISCHE HOCHSCHULE LÜBECK**

The diagram shows a class hierarchy with **Vater** at the top, followed by **Sohn**. **Vater** has attributes `wert1` and method `methode1()`. **Sohn** has attributes `wert2` and method `methode2()`. An arrow points from **Vater** to **Sohn**, indicating inheritance.

On the left, a code snippet is shown:

```
Sohn s = new Sohn();
Vater v = s;
Sohn s2 = (Sohn)v;
```

On the right, two views are shown under the heading "Sicht einer Referenz vom Typ **Vater**". The first view shows a reference `refVater` pointing to a box labeled `:Vater` containing `wert1` and `methode1()`. The second view shows a reference `refSohn2` pointing to a box labeled `:Sohn` containing `wert1`, `wert2`, `methode1()`, and `methode2()`.

A curved arrow labeled "explizites Casten" points from the `refVater` box to the `:Sohn` box, indicating the cast operation.

**Eine explizite Typkonvertierung (cast) von Referenzen muss immer dann erfolgen, wenn bei einer Zuweisung eine Referenzvariable vom Typ Vater auf ein Objekt der Klasse Sohn zeigt und einer Referenzvariablen vom Typ Sohn zugewiesen wird.**

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

## Casting im Überblick

**Zulässige implizite und explizite Type Casts**

**TECHNISCHE HOCHSCHULE LÜBECK**

The diagram shows a class hierarchy with **A** at the top, followed by **B**, then **C**. An arrow points from **A** to **B**, and another from **B** to **C**.

On the left, a text box states: "Wenn oben stehende Klassenhierarchie gilt, dann sind die neben stehenden Cast Operationen zulässig".

On the right, two sections illustrate casting operations:

- zulässige Up-Cast-Operationen:** Shows three references `refA`, `refB`, and `refC` each pointing to a box labeled `:A`, `:B`, and `:C` respectively. Arrows show the flow from `refA` to `:A`, `refB` to `:B`, and `refC` to `:C`.
- zulässige Down-Cast-Operationen:** Shows three references `refA`, `refB`, and `refC` each pointing to a box labeled `:A`, `:B`, and `:C` respectively. Arrows show the flow from `refA` to `:A`, `refB` to `:B`, and `refC` to `:C`. Additionally, arrows show `(B)` pointing to `refA` and `(C)` pointing to `refC`.

**Quelle:** Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

## Mini-Übung: Casting

```

classDiagram
    class Object
    class D
    class A
    class B
    class C

    Object <|-- D
    Object <|-- A
    Object <|-- B
    B <|-- C

    D --> Object
    A --> Object
    B --> Object
    C --> B
  
```

<code>B b = new C();</code>	Ja, impliziter Upcast
<code>A a = b;</code>	Ja, impliziter Upcast
<code>Object o = b;</code>	Ja, impliziter Upcast
<code>B b2 = new B(); C c = (C)b2;</code>	Nein, expliziter Downcast aber b2 vom Typ B nicht C
<code>C c = (C)b;</code>	Ja, expliziter Downcast und b vom Typ C
<code>D d = (D)b;</code>	Nein, expliziter Cast aber b vom Typ C nicht D

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

61

## Verdecken von Datenfeldern (I)

Nicht Überschreiben von Datenfeldern!!!

```

classDiagram
    class Vater
    class Sohn

    Vater <|-- Sohn
  
```

- Vom Verdecken von Datenfeldern spricht man, wenn in der Sohnklasse ein Datenfeld angelegt wird, das den gleichen Namen trägt wie ein von der Vaterklasse geerbtes Datenfeld
- Das Verdecken von Datenfeldern erfolgt grundsätzlich bei Namensgleichheit, der Typ eines Datenfeldes ist hier nicht relevant!
- Zugriff auf solche Felder erhält man (wie auch bei Methoden) mit der super-Referenz

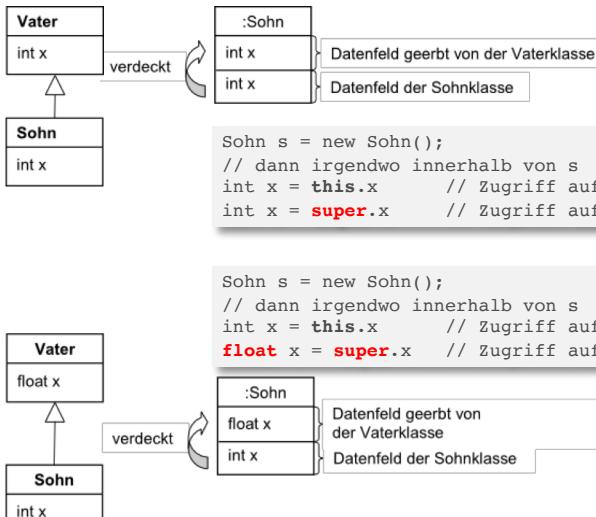
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

62

## Verdecken von Datenfeldern (II)

Nicht Überschreiben von Datenfeldern!!!


  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

63

## Finale Methoden und finale Klassen


  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**

- Finale Methoden können in einer Subklasse nicht überschrieben werden
- Finale Klassen sind Klassen, von denen man zwar Objekte instantiiert, aber keine weiteren Klassen ableiten kann
- Hierzu nutzt man in JAVA das Schlüsselwort final

<p><b>Deklaration finaler Methoden</b></p> <pre>class C {     public void aenderbareMethode() { ... }     public <b>final</b> void finaleMethode() { ... }</pre>	<p><b>Deklaration finaler Klassen</b></p> <pre><b>final</b> class C {     ... }</pre>
--	---

Meist sind es konzeptionelle Gründe des Designs um finale Methoden und Klassen zu nutzen, häufig Sicherheitsgründe um z.B. zu verhindern das Trojanische Pferde von Hackern eingeschleust werden können (ein abgeleitetes Objekt kann überall dort stehen, wo auch ein (vertrauenswürdiges) Vaterobjekt stehen kann).

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

64

## Abstrakte Klassen



- In Basisklassen kann nur die Schnittstelle (Signatur/Methodenrumpf) einer Methode festgelegt werden, aber nicht die Implementierung
- Solche Methoden nennt man abstrakte Methoden
- Eine Klasse mit mindestens einer abstrakten Methode nennt man abstrakte Klasse
- Abstrakte Klassen und Methoden sind mit dem Schlüsselwort **abstract** zu versehen
- Von abstrakten Klassen können keine Objekte instantiiert werden
- Abstrakte Methoden werden üblicherweise dazu genutzt, um Logik zwar vorzusehen, ansprechbar zu machen, aber noch nicht implementieren zu müssen.
- Sie stellen eine Art Pluginmöglichkeit für nachträglich zu ergänzenden Code dar.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

65

## Beispiel einer abstrakten Klasse (I)



Definition der abstrakten Klasse Person, die Ausgabefunktionalität print ist zwar vorgesehen, aber noch nicht implementiert

```
abstract class Person {  
    protected String vorname;  
    protected String nachname;  
  
    public void setNachname(String nn) { nachname = nn; }  
  
    public void setVorname(String vn) { vorname = vn; }  
  
    public abstract void print();  
}  
  
class Student extends Person {  
    private int matrikelnummer;  
  
    public void setMatrikelnummer(int mn) {  
        matrikelnummer = mn;  
    }  
  
    public void print() {  
        System.out.print("Name: ");  
        System.out.println(vorname + " " + nachname);  
        System.out.print("Matrikelnummer: ");  
        System.out.println(matrikelnummer);  
    }  
}
```

Ableitung der Klasse Student aus Person, nun muss auch die abstrakte Methode implementiert werden, damit Objekte instantierbar sind.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

66

## Beispiel einer abstrakten Klasse (II)



```
Person s = new Student();
s.setNachname("Mustermann");
s.setVorname("Max");
s.print();
```

Erzeugt folgende Ausgabe:

Name: Max Mustermann  
Matrikelnummer: 0

Obwohl die Implementierung von print() in der Klasse Person nicht vorgenommen wurde, sondern erst in Student!

Mittels abstrakten Methoden, kann man also Codebereiche vorsehen, die erst im Nachhinein hinzugefügt werden. Solche Arten von „Hooks“ werden gerne in der Frameworkprogrammierung genutzt, um Entwicklern im Nachhinein zu ermöglichen, problemspezifischen Code einzufügen.

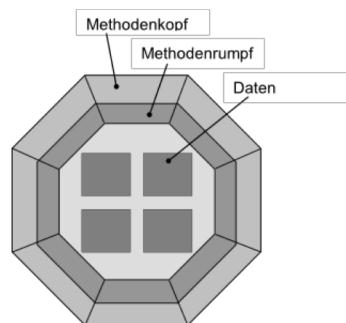
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

67

## Schnittstellen



- Sie haben gesehen, dass mittels abstrakter Methoden einige Methoden zwar vorgesehen sind, aber noch nicht implementiert werden müssen.
- Zu diesen Methoden wird also eine Aufrufschwelle vorgesehen, aber noch nicht die Implementierung.
- Was ist eigentlich eine abstrakte Klasse, ausschließlich mit abstrakten Methoden?
- Die Beantwortung dieser Frage führt uns zu dem Konzept der Schnittstelle



Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

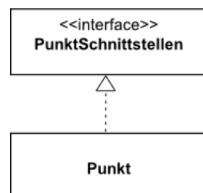
68

## Trennung von Spezifikation und Implementierung



In der Schnittstellenspezifikation stehen nur die zu implementierenden Methodensignaturen

```
interface PunktSchnittstellen {
    public int getX();
    public int getY();
    public int setXY(int, int);
}
```



```
class Punkt implements PunktSchnitstellen {
    int x; int y;

    public int getX() { return x; }
    public int getY() { return y; }
    public int setXY(int x, int y) {
        this.x = x;
        this.y = y;
    }
}
```

In der Klasse werden die zu implementierenden Methodensignaturen realisiert

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

69

## Schnittstellen als Datentyp



- Eine Schnittstelle ist ein Referenztyp. Von ihm können Referenzvariablen gebildet werden, die auf Objekte zeigen, deren Klassen diese Schnittstellen implementieren.
- Schnittstellen können daher als Datentyp für Referenzvariablen oder Aufrufparameter in Methoden angegeben werden
- Dies bedeutet dann: Ein entsprechendes Objekt muss die Schnittstelle implementiert haben, d.h. von einer Klasse instantiiert, die eine Schnittstelle implementiert

Beispiel der Nutzung einer Schnittstelle als Referenzvariablenotyp

```
Punktschnittstellen ps = new Punkt();
```

Beispiel der Nutzung einer Schnittstelle als Aufrufparametertyp

```
public void verschiebe(Punktschnittstellen ps, int x, int y) {
    ps.setXY(ps.getX() + x, ps.getY() + y);
}
```

**!!! Wichtig !!!** Ein Objekt von einer Klasse hat also nicht nur den Typ dieser Klasse (und aller Superklassen), sondern auch den Typ aller Schnittstellen, den diese Klasse implementiert (und aller Schnittstellen, die Superklassen implementieren).

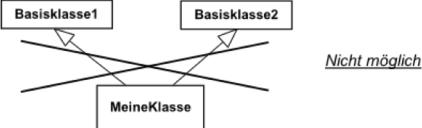
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

70

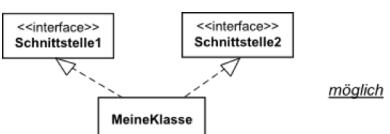
## Implementieren mehrerer Schnittstellen



- Eine Klasse kann nur von einer (abstrakten) Klassen abgeleitet werden

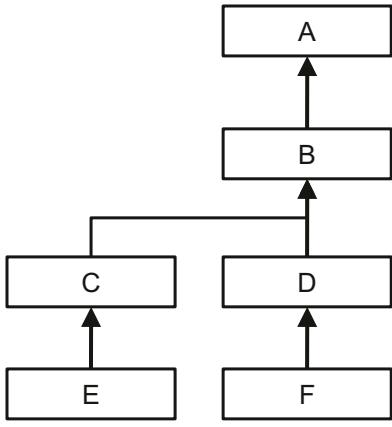


- Im Gegensatz dazu kann eine Klasse beliebig viele Schnittstellen implementieren



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme    71

## Miniübung: Typen bei Vererbungshierarchien



```
B var = new B();
```

 var hat die Typen: B, A

```
D var = new D();
```

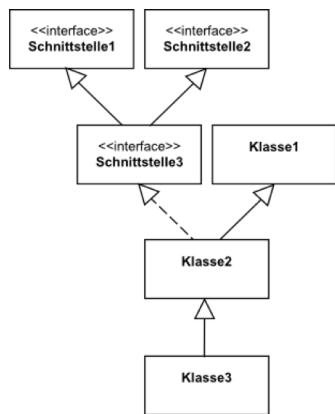
 var hat die Typen: D, B, A

```
E var = new E();
```

 var hat die Typen: E, C, B, A

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme    72

## Miniübung: Resultierende Typen bei Schnittstellen



Typen eines Objekts der Klasse1?  
Klasse1

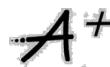
Typen eines Objekts der Klasse2?  
Klasse2, Klasse1, Schnittstelle3,  
Schnittstelle1, Schnittstelle2

Typen eines Objekts der Klasse3?  
Klasse3, Klasse2, Klasse1,  
Schnittstelle3, Schnittstelle1,  
Schnittstelle2

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

73

## Zusammenfassung



- **Klassen (Gruppierung von gleichartigen Objekten)**
  - Konformität
  - Ersetzbarkeit
- **Klassenhierarchien**
  - Superklassen
  - Subklassen
  - Überschreiben von Methoden
  - Ergänzen von Methoden und Datenfeldern
  - Finale und Abstrakte Klassen
- **Schnittstellen**
  - Trennung von Spezifikation und
  - Implementierung
- **Figurenbeispiel und Flächenberechnung**



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

74

## Themen dieser Unit



Objekte
<ul style="list-style-type: none"><li>• Zugriffsmodifikatoren</li><li>• Initialisierung</li><li>• Instantiierung</li><li>• this Referenz</li><li>• Klasse object</li></ul>

Klassen
<ul style="list-style-type: none"><li>• Klassen (Gruppen gleichartiger Objekte)</li><li>• Klassenhierarchien</li><li>• Schnittstellen</li></ul>

Pakete
<ul style="list-style-type: none"><li>• Programming in the small/large</li><li>• Zugriffsmodifikatoren</li><li>• Benennung</li></ul>

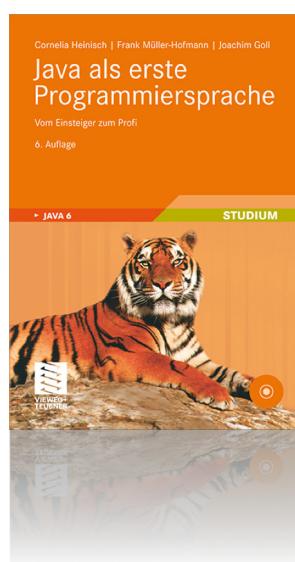
Exceptions
<ul style="list-style-type: none"><li>• Checked und Unchecked</li><li>• Fangen und Behandeln</li><li>• Ausnahmen werfen/propagieren</li></ul>



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

75

## Zum Nachlesen ...



### Kapitel 12

#### Pakete

- 12.1 Programmierung im Großen
- 12.2 Pakete als Entwurfseinheiten
- 12.3 Erstellung von Paketen
- 12.4 Benutzung von Paketen
- 12.5 Paketnamen
- 12.6 Gültigkeitsbereich von Klassennamen
- 12.7 Zugriffsmodifikatoren

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

76

## Pakete

- Eine moderne Programmiersprache soll das Design eines Programms unterstützen
- Wichtig hierbei ist es, Programme in Programmeinheiten zu unterteilen
- Dies dient der Strukturierung von Programmen oder der sogenannten „Programmieren im Großen“
- Programmeinheiten sind grobkörnige Teile eines Programms die einen Namen tragen
- In JAVA sind dies Pakete
- Pakete tragen einen Namen und können Klassen, Schnittstellen und Unterpakete umfassen

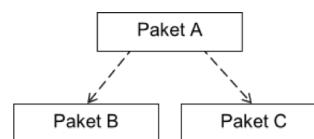


Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

77

## Pakete als Entwurfseinheiten

- Pakete stellen die größten Strukturierungseinheiten der OO-Technik dar
- Pakete werden im Rahmen des Entwurfs der Software konzipiert



Schichtenmodell für Pakete. Der Pfeil bedeutet hier benutzt.

Pakete bilden eigene Bereiche für den Zugriffsschutz (Information Hiding)

Pakete bilden Namensräume um Namenskonflikte bei identischen Namen für Klassen oder Schnittstellen zu vermeiden

Pakete sind größere Einheiten für die Strukturierung von objektorientierten Systemen als Klassen

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

78

## Erstellung von Paketen



- Ein Paket wird definiert, indem alle Dateien des Pakets mit der Deklaration des Paketnamens versehen werden
- Die Deklaration des Paketnamens erfolgt mittels des Schlüsselworts `package`

Datei: Artikel.java

```
package lagerverwaltung;

public class Artikel {
    private String name;
    private float preis;

    public Artikel(String n, float p) {
        name = n;
        preis = p;
    }

    // weitere Methoden der Klasse
}
```

Datei: Regal.java

```
package lagerverwaltung;

public class Regal {
    ...
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

79

## Regeln bei der Erstellung von Paketen



- Paketbezeichner werden konventionsgemäß kleingeschrieben
- Für eine Datei kann nur eine Paketdeklaration angegeben werden
- Enthält eine Datei eine `public` Klasse, so muss der Dateiname gleich sein, wie der Klassename der `public` Klasse
- Enthält eine Datei keine `public` Klasse, so kann der Dateiname beliebig sein (syntaktische Zulässigkeit vorausgesetzt)
- Maximal eine Klasse einer Quellcode-Datei kann `public` sein
- Soll eine Klasse aus einem anderen Paket heraus nutzbar sein, so muss sie `public` sein, ansonsten ist sie nur innerhalb des Pakets als Serviceklasse (Hilfsklasse) nutzbar
- Ein Paket kann Unterpakete enthalten. Diese werden durch `.` voneinander getrennt. Bsp. Das Paket `teilpaket` ist Unterpaket des Pakets `hauptpaket`, so würde man schreiben
- `package hauptpaket.teilpaket;`



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

80

## Paketnamen Konvention Umgekehrte Domain-Namen



- Um bei großen Projekten Namenskollisionen zu vermeiden
- und um unkompliziert Klassenbibliotheken unterschiedlicher Hersteller verwenden zu können
- Gilt folgende Konvention für Anbieter von JAVA-Paketen
  - Gliederung der eigenen Pakete unter dem Internet-Domain-Namen des Herstellers in umgekehrter Reihenfolge
  - z.B. alle Pakete des Herstellers SUN (<http://www.sun.com>) werden unter
    - com.sun eingegliedert
- Für die Fachhochschule Lübeck gelte demnach?

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

81

## Paketnamen Konvention Beispiel: Softwarepakete der FH-Lübeck



Für Pakete die im Rahmen dieser Vorlesung entstehen, könnte z.B. gelten?

<http://www.fh-luebeck.de>



de.fhl.prog

**Hinweis:** Bindestriche dürfen in Paketnamen nicht auftauchen. In URLs sehr wohl.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

82

## Paketnamen und Verzeichnisstruktur



- Die Paketstruktur in JAVA wird in die Verzeichnisstruktur des Rechners umgesetzt
- Nach Paketen wird auf einem Rechner unterhalb eines sogenannten CLASSPATH gesucht
- Der CLASSPATH zeigt auf ein oder mehrere Verzeichnisse auf dem Rechner, in dem/denen nach JAVA Paketen gesucht wird
- Steht der CLASSPATH bspw. auf
  - C:\projekte\projekt1
- Und soll auf die Klasse Beispiel im Paket prinzip zugegriffen werden, so wird die Klasse Beispiel im Verzeichnis
  - C:\projekte\projekt1\prinzip
- gesucht



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

83

## Bedeutung der Paketnamen



- Paketnamen bestehen aus Komponenten, die durch Punkte voneinander getrennt sind
- Einerseits bietet dies eine visuelle Strukturierung
- Andererseits existiert eine Entsprechung im Dateisystem, die dem Compiler sagen, wo welche Klassendateien zu finden sind
- Beispiel:
  - Klassen aus dem Paket **com.sun.image.codec.jpeg**
  - Werden im relativ zum **CLASSPATH** im Verzeichnis
  - **com/sun/image/codec/jpeg** gesucht.



### !!! Wichtig zu wissen !!!

- Verzeichnisnamen entsprechen Paketnamen
- Paketnamen werden stets vollständig klein geschrieben
- Jedes Unterpaket stellt ein Unterverzeichnis dar

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

84

## Benutzung von Paketen



Genauso wie die Komponenten einer Klasse (Datenfelder und Methoden) mit Hilfe des Punktoperators angesprochen werden können, so können auch die Komponenten von Paketen (Klassen, Schnittstellen und Unterpakete) mit Hilfe des Punktoperators angesprochen werden.

Beispielsweise ist die Klasse **Vector** im Unterpaket **util**, welches sich wiederum im Paket **java** befindet. Es kann also wie folgt angesprochen werden:

**java.util.Vector**

### !!! Wichtig zu wissen !!!

- Mittels public deklarierte Klassen können mittels der import Vereinbarung in anderen Paketen sichtbar gemacht werden
- Eine (oder mehrere) import Anweisung steht immer hinter einer package Anweisung, aber vor dem Rest des Programms.



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

85

## Zwei Varianten des Importierens



### Import-Variante 1

```
{ggf. package Anweisung}  
import java.util.Date;  
  
Date d = new Date();  
java.util.Vector v = new java.util.Vector();
```

Importieren  
genau einer  
Klasse aus  
einem Paket

### Import-Variante 2

```
{ ggf. package Anweisung }  
import java.util.*;  
  
Date d = new Date();  
Vector v = new Vector();
```

Importieren aller  
Klassen aus  
einem Paket

**Hinweis:** Import-Variante 2 kann zu Namenskollisionen führen, wenn in zwei importierten Paketen dieselben Klassennamen genutzt werden. Dann muss auf die häufig schreibaufwändigere Import-Variante 1 zurückgegriffen werden.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

86

## Gültigkeitsbereich von Klassennamen



Unter einem Gültigkeitsbereich eines Namens in einer Programmiersprache versteht man den Quelltextbereich, in dem ein vergebener Name bekannt ist.

Der **Gültigkeitsbereich eines Klassennamens** in JAVA erstreckt sich über alle Dateien in einem Paket. Der Compiler geht in JAVA mehrfach über den Quellcode, bis er alle Klassendeklarationen gefunden hat.

### !!! Wichtig zu wissen !!!

- Innerhalb eines Pakets definierte Klassen müssen also nicht mittels import bekannt gemacht werden, auch wenn sie in unterschiedlichen Dateien definiert werden
- Dies gilt auch wenn Quelltextdateien nicht explizit einem Paket zugewiesen werden (dies entspricht dem Paket default)
- Imports werden erst sinnvoll wenn paketübergreifend gearbeitet wird.



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

87

## Zugriffsmodifikatoren



Zur Regelung des Zugriffsschutzes in JAVA gibt es die Zugriffsmodifikatoren **public**, **protected** und **private**.

Ohne Zugriffsmodifikator ist der Zugriffsschutz default. Beachten Sie das default kein Schlüsselwort in JAVA ist, aber einen gültigen Zugriffsschutz bildet. Default wird immer angenommen, wenn weder public, protected oder private angegeben wurden.



### Klassen und Schnittstellen

- Default (friendly)
- public

### Methoden und Datenfelder

- Default (friendly)
- public
- protected
- private

Der Zugriffsschutz in JAVA ist klassenbezogen, nicht objektbezogen definiert !!! (Beispiel folgt bei Erläuterung von private!)

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

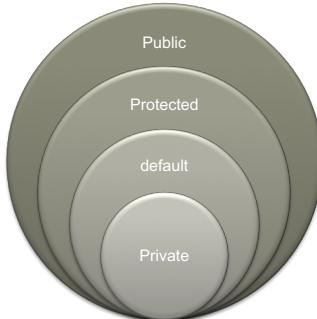
88

## Zugriffsschutz für Klassen, Schnittstellen und Datenfelder, Methoden



Unten stehendes Diagramm veranschaulicht mögliche Zugriffsfälle bezogen auf die Klasse A. Die Zugriffe erfolgen auf A von Klassen B, C, D und E.

- B ist eine Klasse **in demselben Paket** wie A.
- C ist eine von A **abgeleitete Klasse in demselben Paket** wie A.
- D ist eine Klasse **außerhalb des Pakets** der Klasse A.
- E ist eine **von A abgeleitete Klasse außerhalb des Pakets** von A.



Veranschaulichung der Erweiterung des Zugriffsumfangs bei den gezeigten Zugriffsmodifikatoren

Paket x: D, E  
Paket y: A, B, C

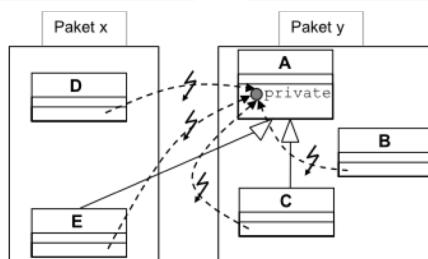
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

89

## Zugriffsmodifikator private (I)



<b>Datenfelder/Methoden</b> <ul style="list-style-type: none"> <li>Auf Datenfelder und Methoden kann nur innerhalb der Klassendefinition zugegriffen werden.</li> <li>Auch aus abgeleiteten Klassen oder Klassen desselben Pakets kann nicht zugegriffen werden</li> <li>Aus fremden Paketen (<b>auch keinen Unterpaketen</b>) kann nicht auf private Datenfelder und Methoden zugegriffen werden.</li> </ul>	<b>Klassen/Schnittstellen</b> <ul style="list-style-type: none"> <li>Es gibt keine private Klassen/Schnittstellen</li> </ul>
---	--



Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

90

## Zugriffsmodifikator private (II)



Auch folgender Zugriff ist dann erlaubt, da der **Zugriffsschutz in JAVA klassenbezogen** definiert ist.

```
public class Punkt {  
    private int x;  
  
    public void func(Punkt p) {  
        int help = p.x;  
        p.x = help;  
        x = help;  
    }  
}
```

**Hinweis:** Es gibt auch OO-Sprachen (z.B. Ruby), bei denen der Zugriffsschutz objektbezogen ist. Dann würde das obige Bsp. nicht funktionieren, da dann jedes einzelne Objekt nur auf seine private Datenfelder/Methoden mittels der this-Referenz zugreifen kann.



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

91

## Zugriffsschutz default

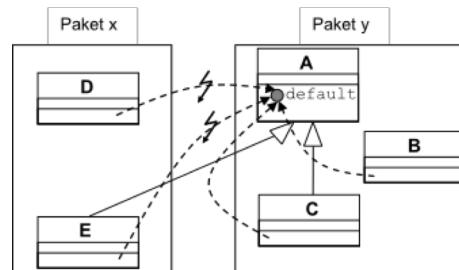


### Datenfelder/Methoden

- Auf Datenfelder und Methoden kann aus im gleichen Paket befindlichen Klassen zugegriffen werden.
- Aus fremden Paketen (**auch keinen Unterpaketen**) kann nicht auf default Datenfelder und Methoden zugegriffen werden.

### Klassen/Schnittstellen

- Default Klassen sind nur für Klassen/Schnittstellen desselben Pakets sichtbar
- Default Klassen können nicht importiert werden
- Schnittstellen können nicht default sein
- Auch in Unterpaketen ist eine default Klasse/Schnittstelle nicht sichtbar



Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

92

## Zugriffsmodifikator protected (I)


  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**

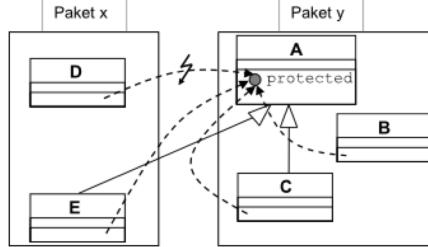
**Datenfelder/Methoden**

- Zugriff auf Datenfelder und Methoden
- aus allen Klassen desselben Pakets
- aus allen Unterklassen des definierenden Pakets (auch aus fremden Paketen/Unterpaketen)
- **sofern der Zugriff auf die geerbten Datenfelder und Methoden erfolgt**

**Klassen/Schnittstellen**

- Es gibt keine protected Klassen/Schnittstellen





```

classDiagram
    package Paket_x {
        D
        E
    }
    package Paket_y {
        A
        B
        C
    }
    D <|-- A
    E <|-- C
    A <|-- B
  
```

Quelle: Java als erste Programmiersprache

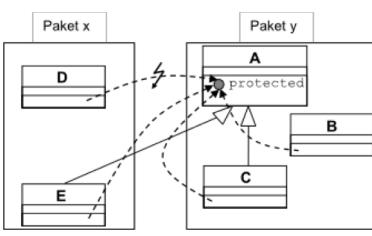
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

93

## Zugriffsmodifikator protected (II)


  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**

Zugriff auf Datenfelder und Methoden aus allen Unterklassen **sofern** der Zugriff auf die **geerbten** Datenfelder und Methoden erfolgt.



```

package y;
public class A {
    protected Object data;
    protected void m() { ... }
}
  
```

```

package x;
public class E extends y.A {
    protected void m() {
        data = null;
        // Zugriff auf geerbtes Datenfeld
        // funktioniert
    }
}
  
```

```

package x;
public class E extends y.A {
    protected void m() {
        A refA = new A();
        refA.data = null;
        // Zugriff über Referenz auf Datenfeld
        // funktioniert nicht (da in anderem
        // package definiert)
    }
}
  
```



Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

94

## Zugriffsmodifikator public

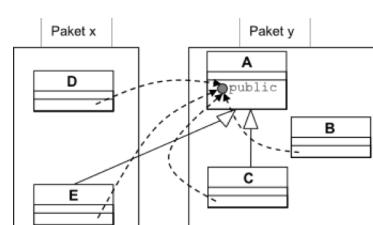

  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**

**Datenfelder/Methoden**

- Auf Datenfelder und Methoden kann von allen Klassen und aus allen Paketen zugegriffen werden

**Klassen/Schnittstellen**

- Public Klassen und Schnittstellen sind für Klassen/Schnittstellen alle Pakete sichtbar



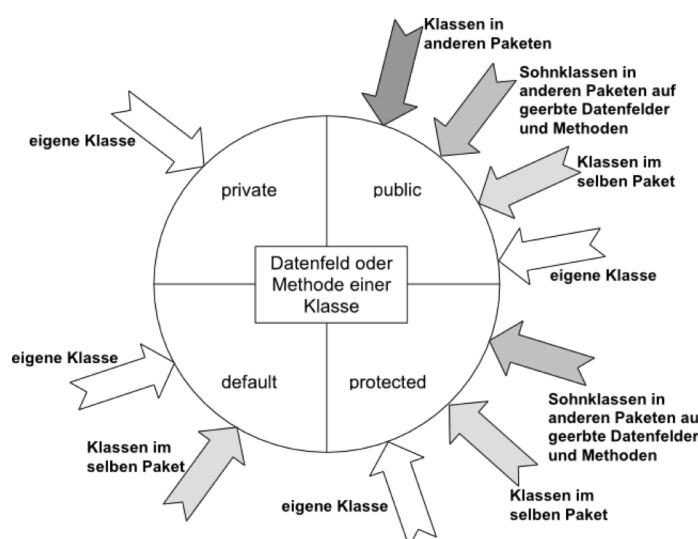
Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

95

## Übersicht aller Zugriffsmodifikatoren


  
**TECHNISCHE  
HOCHSCHULE  
LÜBECK**



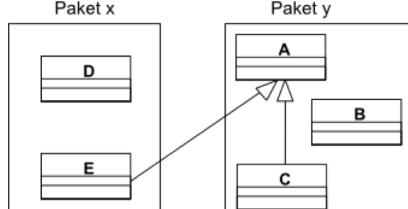
Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

96

## Tabellarische Übersicht aller Zugriffskombinationen


  
TECHNISCHE  
HOCHSCHULE  
LÜBECK



Bis auf **protected** ist der Zugriffsschutz gleich, egal ob auf geerbte Datenfelder und Methoden zugegriffen wird, oder ob in der entsprechenden Klasse ein neues Objekt der Klasse A angelegt wird und auf dessen Datenfeldern und Methoden zugegriffen wird (vgl. auch Zugriffsmodifikator **protected** in diesem Handout).

hat Zugriff auf	private Datenfelder und Methoden	default Datenfelder und Methoden	protected Datenfelder und Methoden	public Datenfelder und Methoden
Klasse A selbst	Ja	Ja	Ja	Ja
Klasse B gleiches Paket	Nein	Ja	Ja	Ja
Subklasse C gleiches Paket	Nein	Ja	Ja	Ja
Subklasse E anderes Paket	Nein	Nein	Ja/Nein	Ja
Klasse D anderes Paket	Nein	Nein	Nein	Ja



Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

97

## Auswirkungen des Zugriffsschutzes auf Konstruktoren


  
TECHNISCHE  
HOCHSCHULE  
LÜBECK

- private Konstruktoren**
  - Objekte dieser Klassen können nur aus der Klasse selber heraus instantiiert werden
- Konstruktoren mit Zugriffsschutz default**
  - Nur Klassen innerhalb des eigenen Pakets können den Konstruktor aufrufen und Objekte instantiiieren
- protected Konstruktoren**
  - Nur Klassen innerhalb des eigenen Pakets können den Konstruktor aufrufen und Objekte instantiiieren
  - Abgeleitete Klassen außerhalb des Pakets können keinen Objekte instantiiieren, nur im Rahmen der eigenen Instantiierung den Konstruktor der Vaterklasse mittels `super()` aufrufen
- Klassen ohne Konstruktoren**
  - Greifen auf den vom Compiler bereitgestellten default Konstruktor zurück
  - Dieser hat den Zugriffsschutz der Klasse (public oder default)

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

98

Prof. Dr. rer. nat. Nane Kratzke (Praktische Informatik)  
Fachhochschule Lübeck – Fachbereich Elektrotechnik und Informatik

Stand: 21.02.19

Seite 49

## Zugriffsmodifikatoren beim Überschreiben



**!!! Wichtig !!!** Man darf die Zugriffsmodifikatoren einer überschriebenen Methode zwar ändern, aber niemals einschränken.



Zugriffsmodifikatoren in der Superklasse	Zugriffsmodifikatoren in der Subklasse
private	Kein Überschreiben möglich, aber neue Definition im Sohn.
default	default protected public
protected	protected public
public	public

Dies hat mit dem Liskovschen Substitutionsprinzip zu tun. Würde man die Zugriffsrechte beim Überschreiben einer Methode einschränken, so könnte nicht an jeder Stelle, an der ein Vater verlangt wird, ein Sohn stehen – der Vertrag der Vaterklasse wäre verletzt, da die Vorbedingung verschärft wurde.

Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke

99

## Zusammenfassung



- Pakete als Entwurfseinheiten
  - Strukturierung
  - Information Hiding
- Erstellung von Paketen und Benennungskonventionen
- Importieren von Paketen
- Zugriffsmodifikatoren bei Paketen
- Liskovsches Prinzip beim Ändern der Zugriffsmodifikatoren überschriebener Methoden (und Klassen)



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

100

## Themen dieser Unit



- Objekte**
- Zugriffsmodifikatoren
  - Initialisierung
  - Instantiierung
  - this Referenz
  - Klasse object

- Klassen**
- Klassen (Gruppen gleichartiger Objekte)
  - Klassenhierarchien
  - Schnittstellen

- Pakete**
- Programming in the small/large
  - Zugriffsmodifikatoren
  - Benennung

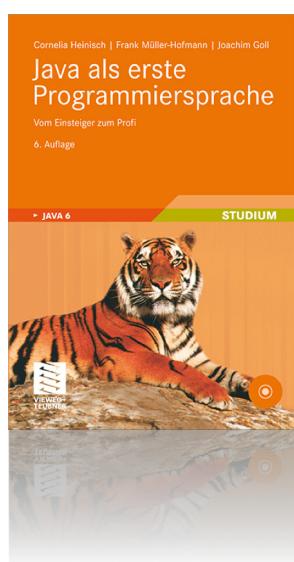
- Exceptions**
- Checked und Unchecked
  - Fangen und Behandeln
  - Ausnahmen werfen/propagieren



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

101

## Zum Nachlesen ...



### Kapitel 13

#### Ausnahmebehandlung

- 13.1 Das Konzept des Exception Handling
- 13.2 Implementierung von Exception-Handlern
- 13.3 Ausnahmen vereinbaren und auswerfen
- 13.4 Exception-Hierarchie
- 13.5 Ausnahmen behandeln
- 13.6 Vorteile des Exception-Konzeptes

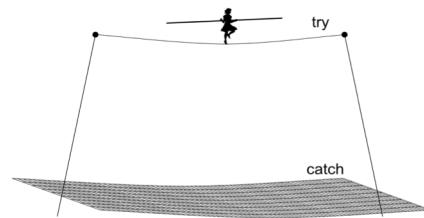
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

102

## Das Konzept des Exception Handling



- Eine Exception stellt ein Laufzeitereignis dar, das zum Versagen eines Programms führen kann, bspw.
  - Arithmetischer Überlauf
  - Mangel an Speicherplatz
  - Verletzung von Array-Grenzen, etc.
- Formal betrachtet, tritt in einer Methode eine Exception auf, wenn trotz erfüllter Voraussetzung die Nachbedingung der Methode verletzt wird.



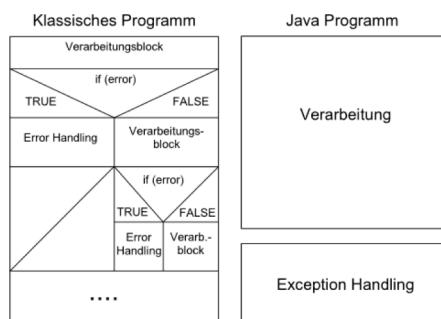
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

103

## Ziel des Exception Handling Konzepts



- Trennung des normalen Code (Logik)
- vom Fehler behandelndem Code



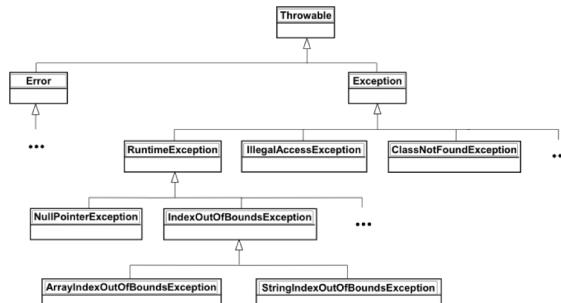
*Ideale Darstellung*  
*Exception Handling soll die eigentliche Programmlogik von der Fehlerbehandlung befreien.*

**Exceptions ermöglichen einer Bibliothek, Ausnahmezustände an das aufrufende Programm zu melden und Daten über die Ursachen bereitzustellen.**

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

104

## Die Exception Hierarchy (JAVA)



- Alle Exceptions haben in JAVA eine gemeinsame Basisklasse, die Klasse `Throwable`.
- Es lassen sich drei Kategorien von Exceptions unterscheiden.
  - Errors (unchecked)
  - Runtime Exceptions (unchecked)
  - Normale Exception (checked)

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

105

## Checked und unchecked Exceptions

Nicht alle Exceptions müssen vom Programmierer abgefangen und bearbeitet werden. Man unterscheidet die folgenden Exceptionarten:

### Checked Exceptions

- Müssen abgefangen werden
- Bzw. muss in `throws` Klausel angegeben werden
- Compiler prüft (check – daher der Name) die Behandlung dieser Ausnahmen zur Compilezeit

Alle anderen Exceptions

### Unchecked Exceptions

- Müssen nicht abgefangen werden
- Müssen nicht in `throws` Klauseln angegeben werden
- Compiler prüft nicht gegen diese Exceptions, da sie nur zur Laufzeit entstehen können und durch statische Codeanalysen nur schwer vorhersagbar sind

Error

`RuntimeException`

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

106

## Die drei Exception Arten (Klassen)

### Error

- Schwerwiegender Fehler in der virtuellen Maschine
- Solche Fehler sollten zur Laufzeit nicht auftreten und sind durch Programme schwer zu heilen, wenn sie auftreten
- Bspw.: OutOfMemoryException
- Error können, müssen aber nicht dem Exception Handling unterworfen werden

### RuntimeException

- Laufzeit Fehler in der virtuellen Maschine
- Keine „harten“ Fehler der virtuellen Maschine, sondern durch die Programmierung begründet
- Bspw.: NullPointerException, ArrayIndexOutOfBoundsException
- Solche Fehler können prinzipiell bei jedem Zugriff auf Datenfeld entstehen und es ist kaum praktikabel diese alle zu fangen
- RuntimeExceptions können daher, müssen aber nicht dem Exception Handling unterworfen werden

### Alles andere von Exception abgeleitete

- Exceptions die nicht in erster Linie auf Programmierfehler zurückzuführen sind, sondern bewusst von Bibliotheken ausgelöst werden, um auf Ausnahme-zustände von Bedeutung hinzuweisen
- Diese Exceptions müssen dem Exception Handling unterworfen werden.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

107

## Beispiele für Exceptions

### Error (checked)

Exception	Erklärung
AbstractMethodError	Versuch, eine abstrakte Methode aufzurufen
InstantiationException	Versuchtes Anlegen einer Instanz einer abstrakten Klasse oder einer Schnittstelle
OutOfMemoryError	Es konnte kein Speicher allokiert werden
StackOverflowError	Der Stack ist übergelaufen

### RuntimeException (unchecked)

Exception	Erklärung
ArithmetricException	Ein Integerwert wurde durch Null dividiert
ArrayIndexOutOfBoundsException	Auf ein Feld mit ungültigem Index wurde zugegriffen
ClassCastException	Cast wegen fehlender Typverträglichkeit nicht möglich
NullPointerException	Versuchter Zugriff auf ein Datenfeld oder eine Methode über die null-Referenz

### Checked Exceptions

Exception	Erklärung
ClassNotFoundException	Eine Klasse wurde weder im aktuellen Verzeichnis noch in dem Verzeichnis, welches in der Umgebungsvariable CLASSPATH angegeben ist, gefunden
CloneNotSupportedException	Ein Objekt sollte kopiert werden, welches das Cloning aber nicht unterstützt
IllegalAccessException	Ein Objekt hat eine Methode aufgerufen, auf die es keinen Zugriff hat

Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

108

## Grundprinzip des Exception Mechanismus

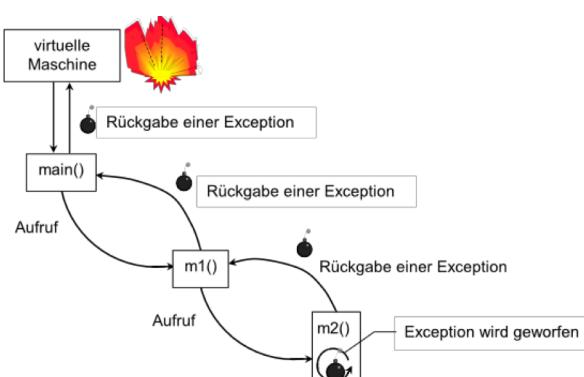


- Laufzeitfehler oder vom Entwickler gewollte Bedingung löst Exception aus
- Behandlung in einem Programmteil oder Weitergabe
- Bei Weitergabe hat der Empfänger erneut die Möglichkeit die Exception zu behandeln oder weiterzugeben
- Wird die Ausnahme von keinem Programmteil behandelt, führt sie zum Abbruch der Applikation

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

109

## Propagieren von Exceptions



Eine Methode muss Exceptions, die sie auslöst, nicht selber abfangen. Dies kann auch in einer sie aufrufenden Methode erfolgen. Man sagt, Exceptions werden **propagiert**. Nicht in einer Methode behandelte Exceptions werden also an den jeweiligen Aufrufer der Methode weitergereicht.

Quelle: Java als erste Programmiersprache

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

110

## Behandlung von Exceptions try-catch Anweisung

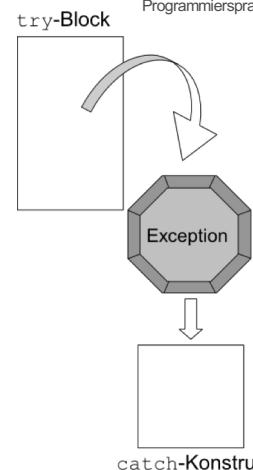


Quelle: Java als erste  
Programmiersprache

### Syntax:

```
try {  
    anweisung;  
} catch (Ausnahmetyp ex) {  
    anweisung;  
}
```

- Der **try-Block** enthält eine oder mehrere Anweisungen, bei deren Ausführung Exceptions entstehen können.
- In diesem Fall wird die normale Programmausführung unterbrochen und die Anweisungen im **catch-Block** zur Behandlung der Exception ausgeführt.



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 111

## Behandlung von Exceptions Beispiel (leere Liste)



```
try {  
    List v = new LinkedList();  
    int i = (Integer)v.elementAt(1);  
    System.out.println("Dies wird nicht mehr ausgegeben.");  
} catch (Exception e) {  
    System.out.println("Exception: " + e.getClass());  
    System.out.println("Message: " + e.getMessage());  
}  
System.out.println("Weiter - als wäre nichts gewesen.");
```

In diesem Beispiel löst die Methode `elementAt` eine `IndexOutOfBoundsException` aus, die in der `catch` Klausel gefangen und behandelt wird.

**!!! Wichtig !!!** Falls ein Handler gefunden wird, werden die Anweisungen des Handlers als nächstes ausgeführt und das Programm nach den Handlern fortgesetzt.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 112

## Exception-spezifische catch Klauseln



- Ausnahmen werden durch die Exception Klasse oder davon abgeleitete Unterklassen repräsentiert.
- Hierdurch ist es möglich mehrere Exceptionklassen durch mehrere catch Blöcke abzufangen und spezifisch zu behandeln.
- Z.B. Division by Zero anders als IO Exceptions bei Dateioperationen

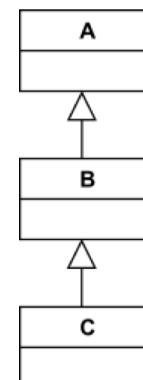
```
try {  
    ...  
}  
catch (ArrayOutOfBoundsException e) { ... }  
catch (NumberFormatException e) { ... }  
catch (IndexOutOfBoundsException e) { ... }
```

*Hinweis: Existieren mehrere Handler, so müssen diese unmittelbar hintereinander folgen. Normaler Code zwischen den Handlern ist nicht erlaubt.*

## Reihenfolge der Handler beachten



- Die Suche nach dem Handler erfolgt von oben nach unten
- Ein Handler für Exceptions der Klasse A passt infolge des Polymorphie-Konzepts der Objektorientierung auch auf Exceptions aller von A abgeleiteten Klassen
- Zuerst müssen die Handler für die spezialisierten Klassen der Exception-Hierarchie aufgelistet werden
- Je spezialisierter eine Exceptionklasse ist, desto früher muss Ihr Handler formuliert werden, andernfalls wird dieser Handler niemals aufgerufen, sondern der Handler für die generellere Exception.



## Miniübung: Handlerreihenfolge (I)



Es können in einem Stück Programm die drei gezeigten Exceptions A, B und C auftreten. Alle sollen so spezifisch wie möglich behandelt werden.

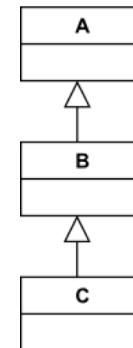
Wie müssen Sie die Handler schreiben?

```

try { ... }
catch (A e) { ... }      // wird bei A, B oder C betreten
catch (B e) { ... }      // wird nie betreten
catch (C e) { ... }      // wird nie betreten

try { ... }
catch (C e) { ... }      // wird nur bei C betreten
catch (B e) { ... }      // wird nur bei B betreten
catch (A e) { ... }      // wird nur bei A betreten

```



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

115

## Miniübung: Handlerreihenfolge (II)



Bei welcher Exception werden die entsprechenden Handler aufgerufen?

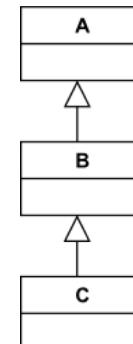
```

try { ... }
catch (C e) { ... }      // wird nur bei C betreten
catch (A e) { ... }      // wird bei A und B betreten
catch (B e) { ... }      // wird nie betreten

try { ... }
catch (B e) { ... }      // wird nur bei B und C betreten
catch (C e) { ... }      // wird nie betreten
catch (A e) { ... }      // wird nur bei Ex. A betreten

try { ... }
catch (B e) { ... }      // wird nur bei B und C betreten
catch (A e) { ... }      // wird nur bei A betreten
catch (C e) { ... }      // wird nie betreten

```



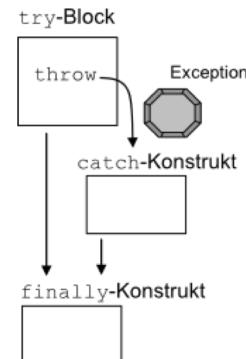
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

116

## Die finally Klausel

- Mit der optionalen `finally` Klausel kann ein Block definiert werden, der immer aufgerufen wird, wenn der zugehörige `try` Block betreten wurde.
- Der `finally` Block wird aufgerufen, wenn
  - Das normale Ende des `try`-Blocks erreicht wurde
  - Eine Ausnahme aufgetreten ist, die durch eine `catch` Klausel gefangen wurde
  - Wenn eine Ausnahme aufgetreten ist, die nicht durch eine `catch` Klausel gefangen wurde
  - Der `try`-Block durch ein `break` oder `return` Sprunganweisung verlassen werden soll.

```
try { ... }
catch (Exception e) { ... }
finally { ... }
```



*Hinweis: Es ist nicht zwingend erforderlich, dass Exceptions in einem catch Block gefangen werden. Werden keine catch Blöcke angegeben, ist jedoch ein finally Block anzugeben.*

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

117

## Ausnahmen werfen throw Klausel

- Mit Hilfe der `throw`-Anweisung können Exceptions erzeugt werden.
- Methoden in denen dies erfolgen kann, müssen dies in ihrer Signatur mittels `throws` deutlich machen.
- Die Behandlung solcher Ausnahmen folgt den gezeigten Regeln.

```
public boolean isPrim(int n) throws ArithmeticException {
    if (n <= 0) throw new ArithmeticException("n < 0");
    if (n == 1) return false;
    for (int i = 2; i <= n/2; i++)
        if (n % i == 0)
            return false;
    return true;
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

118

## Ausnahmen ankündigen throws Klausel



- **Wichtig zu wissen:**
  - Kann eine Methode eine Checked Exception auslösen, so muss Sie dies mittels einer throws Klausel angeben
  - Unchecked Exceptions können auch OHNE throws Klausel geworfen werden
- Durch die throws Klausel informiert eine Methode den Aufrufer (und den Compiler) über eine mögliche abnormale Rückkehr aus der Methode
- Guter Programmierstil ist es daher, jede mit einer throw Anweisung geworfene Exception auch in der throws Klausel des Methodenkopfes anzugeben (egal ob checked oder unchecked Exception).

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

119

## Eigene Ausnahmen definieren



- Haben Ausnahmen ganz bestimmte spezifische Eigenschaften, die im Klassenbaum der Exceptions noch nicht vertreten sind,
- so vereinbart man üblicherweise eine spezielle neue Ausnahmeklasse.
- Eine Ausnahmeklasse unterscheidet sich nicht von einer „normalen“ Klasse, außer dass sie von Throwable abgeleitet ist (zumeist von Exception).
- Die besondere Bedeutung erhält sie durch die Verwendung in throw Anweisungen und in catch Konstrukten.



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

120

## Eigene Ausnahmen definieren, werfen, fangen und behandeln

```
class MyException extends Exception { definieren
    public MyException() {
        // Aufruf des Konstruktors der Klasse Exception.
        // Ihm wird ein FehlerText übergeben und im Objekt
        // gespeichert (Logik geerbt von der Klasse Throwable).
        super("Fehler ist aufgetreten");
    }
}
```

```
try {
    MyException ex = new MyException();
    throw ex;
    // Nach werfen -weisungen würden nie
    // ausgeführt werden
}
catch (MyException ex) { fangen
    System.out.println(ex.getMessage());
}
behandeln
```



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

121

## Vorteile des Exception Handlings

- Eine saubere Trennung des Codes in „normalen“ Code und in Fehlerbehandlungscode
- Nachlässigkeiten beim Programmieren werden bei checked Exceptions bereits zur Kompilierzeit und nicht erst zur Laufzeit entdeckt.
- Propagieren einer Exception erlaubt, diese auch in einem umfassenden Block oder in einer aufrufenden Methode zu behandeln
- Da Exception-Klassen in einem Klassenbaum angeordnet sind, können spezialisierte Handler oder generalisierte Handler geschrieben werden.
- Durch die Möglichkeit der Definition eigener Exceptions, ist es einfacher möglich spezielle Fehlerbehandlungskonzepte für Frameworks, Bibliotheken und Applikationen zu entwickeln.



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

122

## Zusammenfassung



- Konzept und Ziele des Exception Handling
- Checked und Unchecked Exceptions
- Der Exception Mechanismus
- Propagieren von Exceptions
- Fangen und Behandeln von Exceptions
  - Try Klausel
  - Catch Klausel (und Handlerreihenfolge)
  - Finally Klausel
- Ausnahmen werfen (throw) und ankündigen (throws)
- Eigene Ausnahmen definieren, werfen, fangen und behandeln
- Vorteile des Exception Konzepts



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

123