

**Vorlesung**



**Programmieren I und II**

**Unit 6**

Objektorientierte Programmierung und Unified Modeling Language (UML)

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

1

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

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

3



## Units

<b>Unit 1</b> Einleitung und Grundbegriffe	<b>Unit 2</b> Grundelemente imperativer Programme	<b>Unit 3</b> Selbstdefinierbare Datentypen und Collections	<b>Unit 4</b> Einfache I/O Programmierung
<b>Unit 5</b> Rekursive Programmierung und rekursive Datenstrukturen	<b>Unit 6</b> Einführung in die objektorientierte Programmierung und UML	<b>Unit 7</b> Weitere Konzepte objektorientierter Programmiersprachen (Selbststudium)	<b>Unit 8</b> Testen (objektorientierter) Programme
<b>Unit 9</b> Generische Datentypen	<b>Unit 10</b> Objektorientierter Entwurf und objektorientierte Designprinzipien	<b>Unit 11</b> Graphical User Interfaces	<b>Unit 12</b> Multithread Programmierung

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

4

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 Referenztypen (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



Warum eigentlich OO?

- Beherrschung von Komplexität
- Kapselung
- Polymorphie
- Abstraktion

Objekte

- haben ein Verhalten
- haben einen (gekapselten) Zustand
- können kommunizieren
- sind unterschiedlich (aber ähnlich, bzw. polymorph)

Modellieren

- Objekte schützen
- Objekte verknüpfen
- Objekte abstrahieren

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

6

## Zum Nachlesen ...



**Kapitel 1**  
Einleitung

**Kapitel 2**  
Die Basis der Objektorientierung

• Objektorientierte Programmierung verständlich erklärt  
• Von den Prinzipien über den Entwurf bis zur Umsetzung  
• Praxisbeispiele in UML, Java, C#, C++, JavaScript, Ruby, Python und PHP

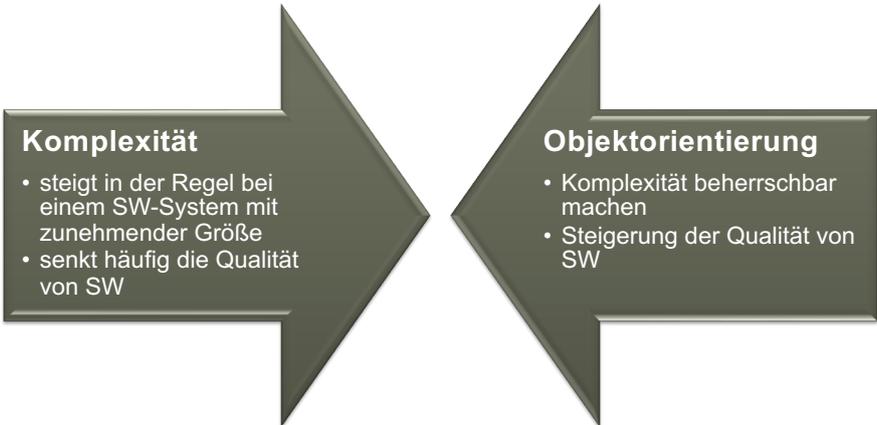
2., aktualisierte und erweiterte Auflage Galileo Computing

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

7

7

## Objektorientierung als Mittel zur Beherrschung von Komplexität



**Komplexität**

- steigt in der Regel bei einem SW-System mit zunehmender Größe
- senkt häufig die Qualität von SW

**Objektorientierung**

- Komplexität beherrschbar machen
- Steigerung der Qualität von SW

„Die Techniken der objektorientierten SW-Entwicklung unterstützen [...] dabei, Software einfacher erweiterbar, besser testbar und besser wartbar zu machen.“

[LR09, S. 27]

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

8

8

## Grundelemente der Objektorientierung



- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:

Kapselung      Poly-morphie      Abstraktion

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

9

9

## Vorläufer der objektorientierten Programmierung



- **Prozedurale Programmierung**
- Ausgangspunkt Inhalt eines Computerspeichers
  - Daten
  - Instruktionen

Strukturierung von Instruktionen	Strukturierung von Daten
<ul style="list-style-type: none"><li>• Verzweigungen</li><li>• Zyklen</li><li>• Routinen mit Aufruf- und Rückgabeparametern</li></ul>	<ul style="list-style-type: none"><li>• Datentypen</li><li>• Zeiger, Records, Arrays, Listen, Bäume, Mengen</li></ul>

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

10

10

## Prozedurale Programmierung



Typische (prozedurale) Programmiersprachen	Objektorientierte Erweiterungen
<ul style="list-style-type: none"><li>• C</li><li>• Pascal</li><li>• Fortran</li><li>• COBOL</li></ul>	<ul style="list-style-type: none"><li>• Kapselung von Daten</li><li>• Polymorphie</li><li>• Vererbung</li><li>• Bspw: geboten durch<ul style="list-style-type: none"><li>• C++, C#</li><li>• JAVA</li><li>• Python</li><li>• PHP</li></ul></li></ul>

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

11

11

## Verantwortlichkeit des Entwicklers bei prozeduralen Programmiersprachen



Das dies manchmal nicht funktioniert, lassen manche C Programme vermuten.

<ul style="list-style-type: none"><li>• ProgrammiererIn hat volle Kontrolle welche Routinen, welche Daten aufrufen.</li></ul>	<ul style="list-style-type: none"><li>• ProgrammiererIn hat auch die Verantwortung, dass die richtigen Routinen die richtigen Daten nutzen.</li></ul>
<b>Kontrolle</b>	<b>Verantwortung</b>

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

12

12

## Grundelemente der Objektorientierung



- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 13

13

## Kapselung von Daten



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 14

14

## Hintergrund der Datenkapselung



- Objekt sorgt für Konsistenz seiner Daten
- dient dem Zwecke:

- Konsistenz der Daten einfacher sicherzustellen
- Reduktion des Aufwands von Änderungen
- Änderungen lassen sich auf Einzelobjekte (bzw. deren Klassen) beschränken

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

15

15

## Prinzip der Kapselung



Daten	Methoden
<ul style="list-style-type: none"><li>• Satz von Variablen</li><li>• Für jedes Objekt neu angelegt (<b>Instanzvariablen</b>)</li><li>• Instanzvariablen repräsentieren den <b>Zustand</b> eines Objekts</li><li>• Zustand eines Objekts kann sich während Lebensdauer ändern</li><li>• Zugriff kann eingeschränkt werden</li></ul>	<ul style="list-style-type: none"><li>• Auf Daten operierende Routinen</li><li>• <b>Methoden</b> nur einmal vorhanden</li><li>• Methoden <b>operieren</b> aber <b>auf Instanzvariablen</b></li><li>• Methoden definieren das Verhalten eines Objekts</li><li>• Zugriff auf Methoden kann eingeschränkt werden</li></ul>

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

16

16

## Daten- und Methodensichtbarkeiten public, protected und private



```
class An_Object {  
  
    public Object forall;  
  
    protected Object forchildren;  
  
    private Object my_eyes_only;  
  
    public Object public_method() {};  
  
    protected Object protected_method() {};  
  
    private Object private_method() {};  
  
}
```

Details folgen ...

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

17

17

## Daten- und Methodensichtbarkeiten public, protected und private



Daten- und Methodensichtbarkeiten können dazu genutzt werden

- Daten zu verbergen (zu kapseln)
- Datenzugriffe einzuschränken
- Datenzugriffe nur über definierte Schnittstellen zuzulassen.
  
- Code zu verbergen (zu kapseln)
- Codeaufrufe einzuschränken
- Codebereiche festzulegen, die für zukünftige Anpassungen gesperrt sind.
- Codebereiche festzulegen, in denen zukünftige Anpassungen stattzufinden haben.

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

18

18

## Objekte werden geschützt



Lord Protector lässt nicht mehr alles zu ...

19

## Grundelemente der Objektorientierung

- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:

Kapselung

Poly-  
morphie

Abstraktion

20

## Prinzip der Polymorphie

Polymorphie bedeutet im Wortsinne „Vielgestaltigkeit“

Bsp.: Fassung und Leuchtmittel

Standardisierte Fassungen arbeiten sowohl mit

Klassischen Glühbirnen	Energiesparlampen	LED-Lampen
------------------------	-------------------	------------



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

21

21

## Prinzip der Polymorphie

- Einheitliche Schnittstellen
- unterschiedliche Ausprägungen von Funktionalitäten
- dient dem Zwecke:

- Bereiche im Code für „Plugins“
- Wiederverwendbarkeit von „Meta“funktionalitäten
- Wesentlich flexiblere Software
- Steigerung der Wartbarkeit und Änderbarkeit

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

22

22

## Polymorphie ist so etwas wie die Steckdose der OO-Programmierung



Schließe an was Du willst, Hauptsache es passt in die Steckdose.  
(implementiert eine Schnittstelle, bzw. Aufrufsignatur)

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

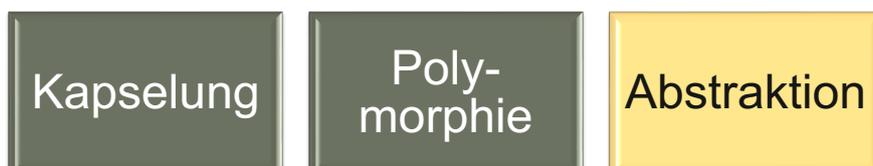
23

23

## Grundelemente der Objektorientierung



- **Objektorientierung** kann als ein Werkzeugkasten verstanden werden, um die Zielsetzungen der Entwicklung von Software anzugehen.
- **Basiswerkzeuge** sind:



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

24

24

Objekte sind unterschiedlich  
(aber ähnlich)



Vieles kann also **wiederverwendet** werden.

Klassen werden uns ermöglichen zu abstrahieren und wiederzuverwenden  
bzw. Polymorphie (Vielgestaltigkeit) in unseren Entwurf einzubetten.

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

25

25

D.h. wir müssen das Rad nicht neu erfinden!



*Auch wenn es vielleicht manchmal cool wäre*

Quelle: Pixabay

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

26

26

## Zusammenfassung

**A+**



- Objektorientierung ist ein Art Werkzeugkasten, um die Entwicklung und Wiederverwendung von Software zu optimieren (steigende Komplexität größerer SW-Systeme zu beherrschen)
- Einleitung in die Kernkonzepte der Objektorientierung
- **Einheit von**
  - Daten (Zustand eines Objekts) und
  - Code (Verhalten eines Objekts)
- **Kapselung**
- **Polymorphie**
- **Abstraktion**


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

27

27

## Themen dieser Unit



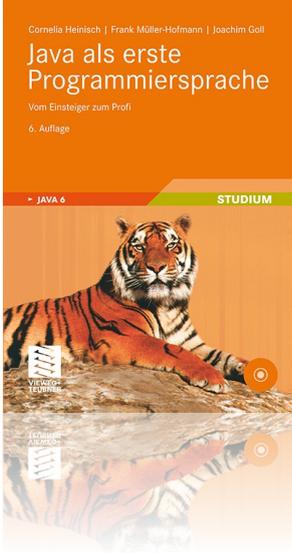
Warum eigentlich OO?	Objekte	Modellieren
<ul style="list-style-type: none"><li>• Beherrschung von Komplexität</li><li>• Kapselung</li><li>• Polymorphie</li><li>• Abstraktion</li></ul>	<ul style="list-style-type: none"><li>• haben ein Verhalten</li><li>• haben einen (gekapselten) Zustand</li><li>• können kommunizieren</li><li>• sind unterschiedlich (aber ähnlich, bzw. polymorph)</li></ul>	<ul style="list-style-type: none"><li>• Objekte schützen</li><li>• Objekte verknüpfen</li><li>• Objekte abstrahieren</li></ul>

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

28

28

**Zum Nachlesen ...**



**Kapitel 2**  
**Objektorientierte Konzepte**

- 2.1 Modellierung mit Klassen und Objekten
- 2.2 Das Konzept der Kapselung
- 2.3 Abstraktion und Brechung der Komplexität

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

29

29

**Noch mehr zum Nachlesen ...**



**Kapitel 4**  
**UML Grundlagen**

- 4.3.1 Klasse
- 4.4.1 Generalisierung, Spezialisierung
- 4.4.2 – 4.4.5 Assoziation (gerichtet, attribuiert, qualifiziert)
- 4.4.7 – 4.4.8 Aggregation und Komposition

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

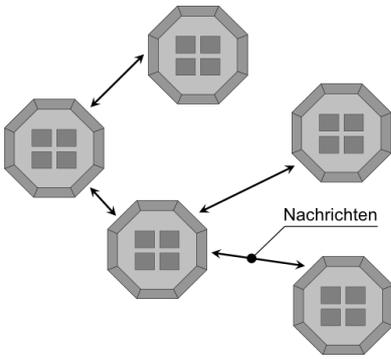
30

30

## Modellierung mit Klassen und Objekten



- Entscheidend für den objektorientierten Ansatz, ist nicht das objektorientierte Programmieren,
- sondern das Denken in Objekten
- Bei der objektorientierten Modellierung denkt man lange Zeit hauptsächlich im Problembereich



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

31

31

## Klassen und UML



- Eine Klasse
  - trägt einen **Klassennamen**
  - enthält **Datenfelder** (Attribute)
  - und **Methoden**, die auf diese Klasse zugreifen.

<b>Punkt</b>	Klassenname Punkt
x : int	Datenfeld x vom Typ int
y : int	Datenfeld y vom Typ int
zeichne( )	Methode zeichne( )
verschiebe( )	Methode verschiebe( )
loesche( )	Methode loesche( )

**Darstellung einer Klasse mittels UML**

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

32

32

## Exkurs: UML Unified Modelling Language



- Die Unified Modeling Language (UML) ist eine graphische Modellierungssprache zur
  - Spezifikation,
  - Konstruktion und
  - Dokumentation von (objektorientierter) Software
- UML hat sich insbesondere im OO-Umfeld als Quasistandard etabliert
- UML definiert graphische Notationen (Diagramme) für statische Strukturen und dynamischen Abläufen
- UML wird von der Object Management Group (OMG) entwickelt und ist zertifizierter ISO Standard (ISO/IEC 19501)



OBJECT MANAGEMENT GROUP  
OBJECT MANAGEMENT GROUP



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

33

33

## Exkurs UML: Diagrammarten



Strukturdiagramme	Verhaltensdiagramme
<ul style="list-style-type: none"><li>• <u>Klassendiagramme</u></li><li>• Montagediagramm</li><li>• Komponentendiagramm</li><li>• Verteilungsdiagramm</li><li>• Objektdiagramm</li><li>• Profildiagramm</li></ul>	<ul style="list-style-type: none"><li>• <u>Aktivitätsdiagramm</u></li><li>• Use Case Diagramm</li><li>• Interaktionsübersichtsdiagramm</li><li>• Kommunikationsdiagramm</li><li>• <u>Sequenzdiagramm</u></li><li>• Zeitverlaufdiagramm</li><li>• <u>Zustandsdiagramm</u></li></ul>

UML kennt die oben stehenden Diagrammarten. Die markierten Diagramme sind die gebräuchlichsten und werden im Rahmen der Vorlesung genutzt.

Die grafische UML-Notation wird Stück für Stück an den geeigneten Stellen im Verlaufe der Vorlesung eingeführt.

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

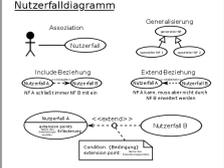
34

34

## Exkurs UML: Diagramm Übersicht nur zur Information



### Nutzerfalldiagramm



### Zustandsdiagramm

**Verhaltenszustandsmaschine (BSM)**  
 System: → Zustand (Ereignis/Übergang)  
 Zustand: → Ereignis (Ereignis/Übergang)  
 Ereignis: → System (Zustand/Übergang)  
 Übergang: → System (Zustand/Übergang)

**Protokollzustandsmaschine (PSM)**  
 System: → Ereignis (Ereignis/Übergang)  
 Ereignis: → System (Zustand/Übergang)  
 Übergang: → System (Zustand/Übergang)

Notationen:  
 ● Zustand    ○ Eintrittspunkt  
 ● Startzustand    ○ Aufrichtpunkt  
 ● Endzustand    ○ flache Historie  
 ◇ Kreuzung oder Entscheidung    ○ tiefe Historie

### Klassendiagramm

**Analysenmodell (fachliche Sicht)**

- Klassen sind Rechtecke
- Attribute → in einem Kasten
- 1:1, 1:n, n:n, n:m
- ggfs. mit Multiplicitäten
- Methoden ohne Parameter und Rückgabewert
- Bi-direktionale Assoziationen/Aggregationen
- Beschriftung von Assoziationen (Multiplicitäten, Rollennamen, Assoziationsnamen (mit Lesartzung))
- Assoziationsklassen und -line Beziehungen
- Generalisierung (Vererbung) (Abstrakte Attribute und Methoden)
- Aufzählungen (Enumerationen)

**Strukturmodell (fachlich-tech. Sicht)**

- Klassen → abstrakt, konkret, Stereotyp
- ggfs. Klassen direkt, Realisation, Stereotyp
- Attribute → Sichtbarkeit, Abstraktion, Klassenattribute, Initialisierung, weitere spezielle Eigenschaften
- Operationen → Parameter, Sichtbarkeit, Rückgabewert, Klassenoperation
- Assoziationen → general, generaliz, sortiert
- Auflösen von Assoziationsklassen / -line Beziehungen
- Abhängigkeiten
- Pakete
- Hilfsmethoden (Konstruktionen, getter/setter, toString, ...)

**Syntax für Attribute**  
 Sichtbarkeit Attributname (Rolle) : Typ (Multiplicität Ordnung) = Initialwert (Eigenschaftswerte)  
 Sichtbarkeit: (readOnly), (ordered), (composite)

**Syntax für Operationen**  
 Sichtbarkeit (Operationsname (Parameterliste) Rückgabewert)  
 Sichtbarkeit: Parameterliste; Rangfolge Name; Typ = Standardwert  
 # protected member    Richtung: in, out, inout  
 ~ private member  
 - package element

### Sequenzdiagramm

Notationen:  
 → Aufruf Nachricht  
 → Antwort Nachricht  
 → Aufruf (synchrone Nachricht)  
 → Antwort (synchrone Nachricht)  
 → Aufruf (asynchrone Nachricht)  
 → Antwort (asynchrone Nachricht)  
 → Aufruf (asynchrone Nachricht)  
 → Antwort (asynchrone Nachricht)

### Aktivitätsdiagramm

Notationen:  
 ● Startpunkt    ◇ Kreuzung oder Entscheidung  
 ○ Endpunkt    ▽ Verzweigung oder Gabelung

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

35

## Klassen und UML



<b>Punkt</b>	Klassenname Punkt
x : int	Datenfeld x vom Typ int
y : int	Datenfeld y vom Typ int
zeichne()	Methode zeichne()
verschiebe()	Methode verschiebe()
loesche()	Methode loesche()

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

    void zeichne() { ... }
    void verschiebe() { ... }
    void loesche() { ... }
}
```

**UML**

Derselbe Sachverhalt – andere Notation

Im Rahmen dieser Vorlesung wird UML primär zur Darstellung struktureller oder ablauf-orientierter Sachverhalte genutzt und JAVA für programmiertechnische Implementierungen.

Beide Formen werden aber parallel genutzt. Lauffähig programmieren lässt sich übrigens nur in JAVA.

**JAVA**

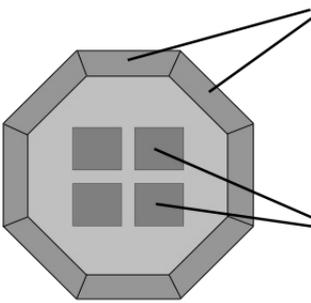
Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

36

## Klassen und Objekte



- Bei der Objektorientierung werden die
  - **Daten** eines Objektes und
  - Die Daten verändernden **Methoden**
  - als eine **Einheit** betrachtet – das **Objekt**.



Methoden sind die Schnittstellen des Objektes.

Daten sind gekapselt. Sie können nur von den eigenen Methoden des Objektes manipuliert werden.

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

37

37

## Klassen und Objekte



- **Methoden** erfüllen die Aufgaben:
  - Werte der Datenfelder **auszugeben**.
  - Datenfelder zu **verändern**.
  - Neue Ergebnisse mittels in Datenfeldern gespeicherter Werte zu **berechnen**.
- **Datenfelder** definieren mögliche **Zustände** der Objekte (Datenstruktur),
- die **Methoden** bestimmen das **Verhalten** der Objekte.

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

38

38

## Objekte haben ein Verhalten

drive()  
park()

fly()  
land()  
film()

drive()  
observe()  
drill()

Quelle: Pixabay

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

39

39

## Objekte haben ein Verhalten (I)

Bislang haben wir Objekte (Instanzen von Klassen) nur als strukturierte Datentypen ohne nennenswertes Verhalten kennen gelernt (bspw. Adresse). Objekte können jedoch auch ein Verhalten zeigen.

Dieses Verhalten wird durch die Methoden des Objekts (eigentlich der Klasse, dazu später mehr) definiert.

Wir definieren nun zwei Klassen, um freundliche und unfreundliche Personen erzeugen zu können (d.h. mit freundlichem und unfreundlichem Verhalten).

Objekte der Klasse `FriendlyPerson` zeigen ein anderes Verhalten als Objekte der Klasse `UnfriendlyPerson`.

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

40

40

## Objekte haben ein Verhalten (II)



```
public class FriendlyPerson {  
    public String name;  
    public FriendlyPerson(String n) { this.name = n; }  
  
    public void sayHello() {  
        System.out.println("[ " + this + "]: Hi, I am " + this);  
    }  
    public String toString() { return name; }  
}  
  
public class UnfriendlyPerson {  
    public String name;  
    public UnfriendlyPerson(String n) { this.name = n; }  
  
    public void sayHello() {  
        System.out.println("[ " + this + "]: Go away. I am busy.");  
    }  
    public String toString() { return name; }  
}
```

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

41

41

## Objekte haben ein Verhalten (III)



```
FriendlyPerson p1 = new FriendlyPerson("Max");  
UnfriendlyPerson p2 = new UnfriendlyPerson("Moritz");  
p1.sayHello();  
p2.sayHello();
```

Ergibt auf der Konsole:

```
[Max]: Hi, I am Max.  
[Moritz]: Go away. I am busy.
```

D.h. Max und Moritz zeigen ein anderes Verhalten  
(aufgrund ihrer Programmierung).

**!!! Methoden definieren das Verhalten von Objekten !!!**

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

42

42

## Objekte haben einen (inneren) Zustand



Quelle: Pixabay

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

43

43

## Objekte haben einen Zustand (I)



```
public class Person {  
    public String name;  
    public Person(String n) { this.name = n; }  
  
    public void sayHello() {  
        System.out.println("[ " + this + "]: Hi, I am " + this);  
    }  
    public String toString() { return name; }  
}
```

```
Person p1 = new Person("Max");  
Person p2 = new Person("Maya");  
p1.sayHello();  
p2.sayHello();
```

**Ergibt auf der Konsole:**

```
[Max]: Hi, I am Max.  
[Maya]: Hi, I am Maya.
```

**Datenfelder  
eines Objekts  
definieren die  
Zustände, die  
ein Objekt  
annehmen  
kann.**

*Hier besteht  
der Zustand  
einer Person  
nur aus einem  
Namen.*

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

44

44

## Objekte haben einen Zustand (II)



- Ein Objekt hat einen Satz von Datenfeldern (und Methoden)
- Jedes Datenfeld hat Werte
- **Zustand eines Objekts == momentane Wertebelegung der Datenfelder des Objekts**
- **Beispiel Fahrstuhl**
  - Gewichtssensor im Fahrstuhl
  - **Mikroskopischer Zustand** des Fahrstuhls == aktueller Wert des Sensors
  - **Makroskopischer Zustand** des Fahrstuhls == Überladen oder nicht Überladen

**Mikroskopischer Objektzustand**  
Aus der Menge aller Wertebelegungen von Datenfeldwerten eines Objekts

↓

**Makroskopischer Objektzustand**  
Diejenigen Teilmengen von Zuständen mit Bedeutung für den Problembereich

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

45

45

## Objekte haben einen Zustand (III)



```
public class SemiFriendlyPerson {  
    public String name;  
    public int helloCounter;  
  
    public SemiFriendlyPerson(String n) { name = n; }  
  
    public void sayHello() {  
        helloCounter++;  
        if (helloCounter < 5) {  
            System.out.println(this + "Hi, I am " + name);  
        } else {  
            System.out.println(this + "Hi");  
        }  
    }  
    public String toString() { return "[" + name + "]: "; }  
}
```

Hier haben wir einen Zustand bestehend aus zwei Datenfeldern. `sayHello()` **ändert** nun zudem den **Zustand** des Objekts und sein **Verhalten** ist **abhängig** vom **Zustand**.

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

46

46

## Objekte haben einen Zustand (IV)



```
SemiFriendlyPerson p3 = new SemiFriendlyPerson("Willi");  
p3.sayHello();  
p3.sayHello();  
p3.sayHello();  
p3.sayHello();  
p3.sayHello();
```

Ergibt auf der Konsole:

```
[Willi]: Hi, I am Willi  
[Willi]: Hi
```

Das Verhalten von Willi ändert sich nach dem fünften Methodenaufruf von `sayHello()` aufgrund seines Zustands (*vielleicht ist er müde die ganze Zeit zu grüßen*).

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

47

47

## Objekte haben einen Zustand (V)



```
public class SemiFriendlyPerson {  
    public String name;  
    public int helloCounter;  
  
    public SemiFriendlyPerson(String n) { name = n; }  
  
    public boolean tiredToGreet() { return helloCounter >= 5; }  
  
    public void sayHello() {  
        helloCounter++;  
        if (tiredToGreet()) { System.out.println(this + "Hi"); }  
        else { System.out.println(this + "Hi, I am " + name); }  
    }  
  
    public String toString() { return "[" + name + "]: "; }  
}
```

„Zustandsgruppen“ (Makrozustand) die das Verhalten eines Objekts beeinflussen werden häufig (aber nicht immer, Klausur !!!) als boolesche Methoden definiert.

Gleichzeitig machen sie den Code so häufig lesbarer („natürlich sprachlicher“).

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

48

48

### Miniübung:



Es kann aber natürlich auch komplexere Zustände geben (aus mehr als einem Datenfeld). Methoden können den Zustand eines Objekts verändern.

```
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 den Mikrozustand des erzeugten Objekts nach den entsprechenden Methodenaufrufen an.

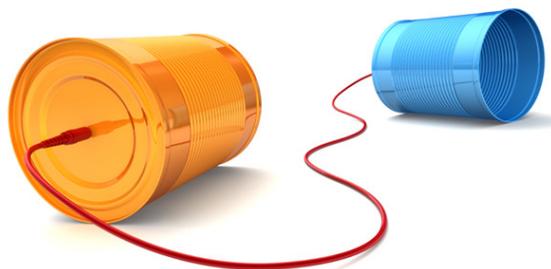
```
Auto car = new Auto();  
  
car.tanke(50.0);  
  
car.fahre(50.0);  
  
car.fahre(200.0);  
car.tanke(10.0);
```

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

49

49

### Objekte können kommunizieren



Quelle: Pixabay

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

50

50

## Objekte können kommunizieren



Damit Objekte miteinander kommunizieren (d.h. sich gegenseitig ihre Methoden aufrufen) können, müssen sie einander kennen.

Auf Ebene von UML kann man solch eine Kenntnisbeziehung als **Assoziation** modellieren. UML Assoziationen lassen sich programmieretechnisch als Referenzen auf (Listen von) Objekte(n) abbilden.

Dozent			Student
- String[] unit - String[] antworten - fragen : int + unterrichte(Student) : void + gebeUnit(int) : void + beantworte(String) : String	0..1	# unterrichtet	- String name - List<String> notizen - String[] fragen + Student(String) + hoere(Dozent) : void + notiere(String) : void + unverstaendnis() : void + toString() : String
		# lerntVon	0..*

Hier einmal das Beispiel, dass ein Dozent mehrere Studenten unterrichtet und ein Student von maximal einem Dozenten unterrichtet wird (zu einem Zeitpunkt). Studenten notieren dabei Inhalte und können Fragen stellen (bei Unverständnis). Dozenten geben Units und beantworten Fragen.

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

51

## Objekte kennen sich (mittels Referenzen)



Dozent			Student
- String[] unit - String[] antworten - fragen : int + unterrichte(Student) : void + gebeUnit(int) : void + beantworte(String) : String	0..1	# unterrichtet	- String name - List<String> notizen - String[] fragen + Student(String) + hoere(Dozent) : void + notiere(String) : void + unverstaendnis() : void + toString() : String
		# lerntVon	0..*

```
public class Student {
    protected Dozent lerntVon;
}
```

Jeder Student kennt also seinen Dozenten (lerntVon).

```
public class Dozent {
    protected List<Student> unterrichtet = new LinkedList<Student>();
}
```

Jeder Dozent kennt seine Studenten (unterrichtet)

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

52

## Ein kommunizierender Dozent

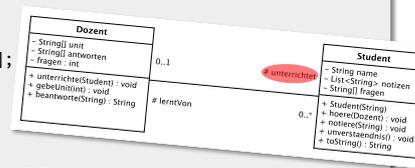


```
public class Dozent {
    private String[] unit = {
        "Ein Objekt hat ein Verhalten.", "Ein Objekt hat einen Zustand.",
        "Ein Objekt kann kommunizieren.", "Ein Objekt ist vielgestaltig." };
    private String[] antworten = { "Eine sehr gute Frage.",
        "Bitte arbeiten Sie dies zu Hause durch.", "Dazu kommen wir noch." };
    private int fragen;
    protected List<Student> unterrichtet = new LinkedList<Student>();

    public void unterrichte(Student s) { unterrichtet.add(s); s.hoere(this); }

    public String beantworte(String s) {
        return antworten[fragen++ % antworten.length];
    }

    public void gebeUnit(int n) {
        for (Student s : unterrichtet) { s.notiere(unit[n]); }
    }
}
```



Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 53

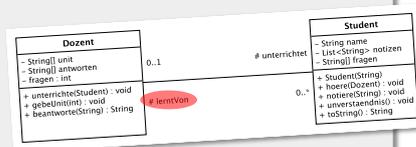
53

## Ein kommunizierender Student



```
public class Student {
    private String name;
    private String[] fragen = { "Gibt es dazu mal ein Beispiel?",
        "Das war mir viel zu schnell!", "Fehlt da nicht ein Semikolon?" };
    private List<String> notizen = new LinkedList<String>();
    protected Dozent lerntVon;

    public Student(String n) { this.name = n; }
    public void hoere(Dozent d) { this.lerntVon = d; }
    public void unverstaendnis() {
        if (this.lerntVon == null) return;
        Random r = new Random();
        String frage = this.fragen[r.nextInt(this.fragen.length)];
        String antwort = this.lerntVon.beantworte(frage);
        System.out.println(this.name + ": " + frage + " Dozent: " + antwort);
    }
    public void notiere(String s) { this.notizen.add("- " + s); }
    public String toString() {
        String ret = "Notizen von: " + name + "\n";
        for (String notiz : notizen) ret += notiz + "\n"; return ret;
    }
}
```



Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 54

54

## Eine exemplarische Kommunikation Dozent -> Student



Dozent		Student
- String[] unit - String[] antworten - fragen : int  + unterrichte(Student) : void + gebeUnit(int) : void + beantworte(String) : String	0..1	# unterrichtet
	# lerntVon	0..*

```

Dozent d = new Dozent();
Student[] students = {
    new Student("Max"),
    new Student("Maren"),
    new Student("Tessa")
};

for (Student s : students) d.unterrichte(s);

d.gebeUnit(0);
d.gebeUnit(2);
d.gebeUnit(1);

for (Student s : students) {
    System.out.println(s);
}
    
```

Notizen von: Max

- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Ein Objekt hat einen Zustand.

Notizen von: Maren

- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Ein Objekt hat einen Zustand.

Notizen von: Tessa

- Ein Objekt hat ein Verhalten.
- Ein Objekt kann kommunizieren.
- Ein Objekt hat einen Zustand.

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

55

## Eine exemplarische Kommunikation Student -> Dozent



Dozent		Student
- String[] unit - String[] antworten - fragen : int  + unterrichte(Student) : void + gebeUnit(int) : void + beantworte(String) : String	0..1	# unterrichtet
	# lerntVon	0..*

```

Dozent d = new Dozent();
Student[] students = {
    new Student("Max"),
    new Student("Maren"),
    new Student("Tessa")
};

for (Student s : students) d.unterrichte(s);

for (Student s : students) {
    s.unverstaendnis();
}
    
```

**Nur so am Rande:**  
 Die Kommunikation bleibt in unserem Beispiel dieselbe, wenn wir die Stoffvermittlung sein lassen ;-)

Max: Gibt es dazu mal ein Beispiel? Dozent: Eine sehr gute Frage.  
 Maren: Das war mir viel zu schnell! Dozent: Bitte arbeiten Sie dies zu Hause durch.  
 Tessa: Fehlt da nicht ein Semikolon? Dozent: Dazu kommen wir noch.

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

56

## Objekte sind unterschiedlich (aber ähnlich)



*also polymorph (vielgestaltig)*

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

57

57

## Objekte sind vielgestaltig



Ein berechtigter Einwand an unserem Beispiel wäre, dass nicht alle Studierende gleich sind.

Es gibt bspw. unterschiedliche Strategien Notizen anzufertigen.

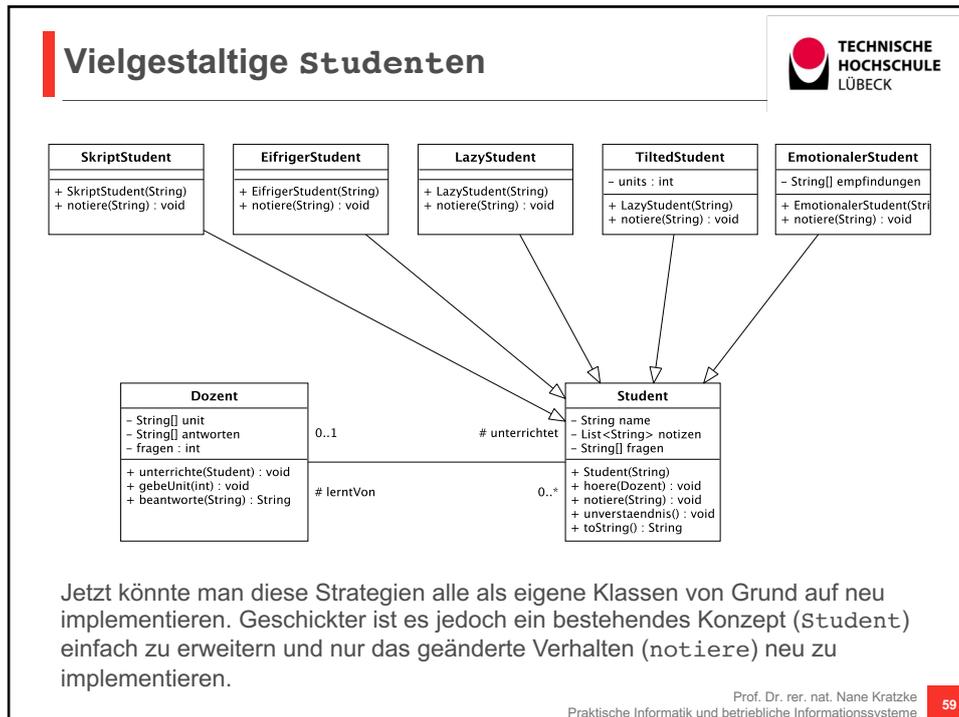
- Der `SkriptStudent` notiert sich gar nichts und vertraut aufs Skript.
- Der `EifrigeStudent` notiert alles und sicherheitshalber doppelt und mit Ausrufezeichen.
- Der `LazyStudent` notiert sich Teile (so zu etwa 50%).
- Der `TiltedStudent` schafft es nicht mehr als zwei Units zu notieren.
- Der `EmotionaleStudent` notiert mehr seine Empfindungen, weniger den Inhalt.

Alle Strategien ändern nichts an der Tatsache, dass Objekte dieser Klassen Studenten bleiben. Der Dozent nimmt auf diese unterschiedlichen Strategien auch gar keine Rücksicht, sondern behandelt alle weiter als `Student`.

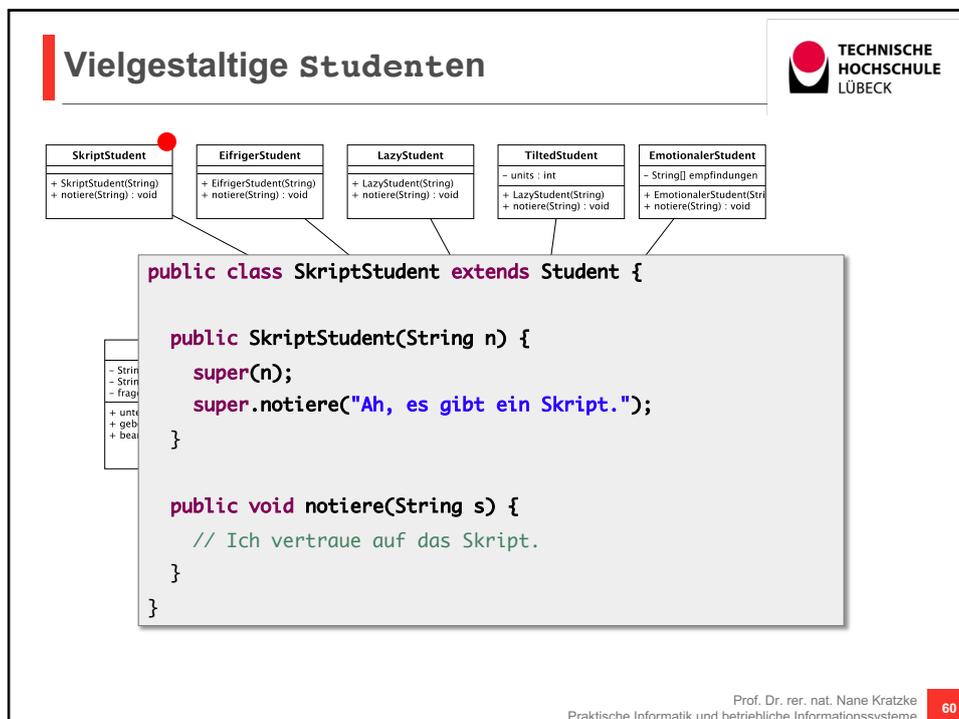
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

58

58



59



60

## Vielgestaltige Studenten



SkriptStudent
+ SkriptStudent(String) + notiere(String) : void

EifrigerStudent
+ EifrigerStudent(String) + notiere(String) : void

LazyStudent
+ LazyStudent(String) + notiere(String) : void

TiltedStudent
- units : int + LazyStudent(String) + notiere(String) : void

EmotionalerStudent
- String[] empfindungen + EmotionalerStudent(Str + notiere(String) : void

```

public class EifrigerStudent extends Student {

    public EifrigerStudent(String n) {
        super(n);
    }

    public void notiere(String s) {
        super.notiere(s + " !!!");
        super.notiere("!!! " + s + " (Nacharbeiten !)");
    }
}
        
```

Student
- Strin - Strin - fragi + unt + geb + bea

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

61

61

## Vielgestaltige Studenten



SkriptStudent
+ SkriptStudent(String) + notiere(String) : void

EifrigerStudent
+ EifrigerStudent(String) + notiere(String) : void

LazyStudent
+ LazyStudent(String) + notiere(String) : void

TiltedStudent
- units : int + LazyStudent(String) + notiere(String) : void

EmotionalerStudent
- String[] empfindungen + EmotionalerStudent(Str + notiere(String) : void

```

public class LazyStudent extends Student {

    public LazyStudent(String n) {
        super(n);
        super.notiere("Jamaica, man!");
    }

    public void notiere(String s) {
        super.notiere(s.substring(0, s.length() / 2));
    }
}
        
```

Student
- Strin - Strin - fragi + unt + geb + bea

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

62

62

## Vielgestaltige Studenten



SkriptStudent
+ SkriptStudent(String) + notiere(String) : void

EifrigerStudent
+ EifrigerStudent(String) + notiere(String) : void

LazyStudent
+ LazyStudent(String) + notiere(String) : void

TiltedStudent
- units : int + LazyStudent(String) + notiere(String) : void

EmotionalerStudent
- String[] empfindungen + EmotionalerStudent(Str + notiere(String) : void

```

public class TiltedStudent extends Student {
    private int units;

    public TiltedStudent(String n) { super(n); }

    private boolean overflow() { return this.units++ >= 2; }

    public void notiere(String s) {
        if (overflow()) { super.notiere("Häh? Tilt ..."); }
        else { super.notiere(s); }
    }
}
    
```

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

63

63

## Vielgestaltige Studenten



SkriptStudent
+ SkriptStudent(String) + notiere(String) : void

EifrigerStudent
+ EifrigerStudent(String) + notiere(String) : void

LazyStudent
+ LazyStudent(String) + notiere(String) : void

TiltedStudent
- units : int + LazyStudent(String) + notiere(String) : void

EmotionalerStudent
- String[] empfindungen + EmotionalerStudent(Str + notiere(String) : void

```

public class EmotionalerStudent extends Student {
    private String[] empfindungen = {
        "Was für ein schöner Sonnenaufgang.", "Wieso immer ich?",
        "Informatik ist so spannend!", "Wieso nur Informatik?",
        "Ich hasse Klausuren.", "Gruppenarbeit ist toll. Das ist so dynamisch.",
        "Objektorientierung ist super.", "Objektorientierung. Wie banal!"
    };

    public EmotionalerStudent(String n) { super(n); }

    public void notiere(String s) {
        Random r = new Random();
        super.notiere(empfindungen[r.nextInt(empfindungen.length)]);
    }
}
    
```

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

64

64

## Eine exemplarische Kommunikation Dozent -> Student



```

Dozent d = new Dozent();
Student[] students = {
    new SkriptStudent("Max"),
    new EifrigerStudent("Maren"),
    new LazyStudent("Tessa"),
    new TiltedStudent("Moritz"),
    new EmotionalerStudent("Maya")
};

for (Student s : students) d.unterrichte(s);
d.gebeUnit(0);
d.gebeUnit(2);
d.gebeUnit(1);
d.gebeUnit(3);

for (Student s : students) {
    System.out.println(s);
}
    
```

Notizen von: Max  
 - Ah, es gibt ein Skript.

Notizen von: Maren  
 - Ein Objekt hat ein Verhalten. !!!  
 - !!! Ein Objekt hat ein Verhalten. (Nacharbeiten !)  
 - Ein Objekt kann kommunizieren. !!!  
 - !!! Ein Objekt kann kommunizieren. (Nacharbeiten !)  
 - Ein Objekt hat einen Zustand. !!!  
 - !!! Ein Objekt hat einen Zustand. (Nacharbeiten !)  
 - Ein Objekt ist vielgestaltig. !!!  
 - !!! Ein Objekt ist vielgestaltig. (Nacharbeiten !)

Notizen von: Tessa  
 - Jamaica, man!  
 - Ein Objekt hat  
 - Ein Objekt kann  
 - Ein Objekt hat  
 - Ein Objekt ist

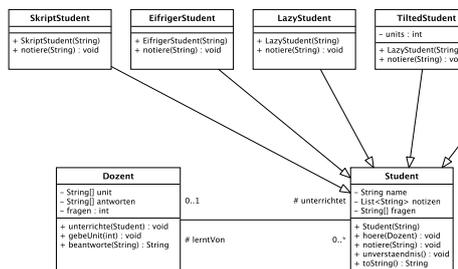
Notizen von: Moritz  
 - Ein Objekt hat ein Verhalten.  
 - Ein Objekt kann kommunizieren.  
 - Häh? Tilt ...  
 - Häh? Tilt ...

Notizen von: Maya  
 - Objektorientierung ist super.  
 - Ich hasse Klausuren.  
 - Was für ein schöner Sonnenaufgang.  
 - Objektorientierung. Wie banal!

Der Dozent spricht alle Objekte einheitlich als Student an. Aber jedes Objekt zeigt jetzt ein anderes Verhalten.

65

## Eine exemplarische Kommunikation Dozent -> Student



Notizen von: Max  
 - Ah, es gibt ein Skript.

Notizen von: Maren  
 - Ein Objekt hat ein Verhalten. !!!  
 - !!! Ein Objekt hat ein Verhalten. (Nacharbeiten !)  
 - Ein Objekt kann kommunizieren. !!!  
 - !!! Ein Objekt kann kommunizieren. (Nacharbeiten !)  
 - Ein Objekt hat einen Zustand. !!!  
 - !!! Ein Objekt hat einen Zustand. (Nacharbeiten !)  
 - Ein Objekt ist vielgestaltig. !!!  
 - !!! Ein Objekt ist vielgestaltig. (Nacharbeiten !)

Notizen von: Tessa  
 - Jamaica, man!  
 - Ein Objekt hat  
 - Ein Objekt kann  
 - Ein Objekt hat  
 - Ein Objekt ist

Notizen von: Moritz  
 - Ein Objekt hat ein Verhalten.  
 - Ein Objekt kann kommunizieren.  
 - Häh? Tilt ...  
 - Häh? Tilt ...

Notizen von: Maya  
 - Objektorientierung ist super.  
 - Ich hasse Klausuren.  
 - Was für ein schöner Sonnenaufgang.  
 - Objektorientierung. Wie banal!

Der Dozent spricht alle Objekte einheitlich als Student an. Aber jedes Objekt zeigt jetzt ein anderes Verhalten.

Den Großteil der Logik müssen wir also nicht anpassen. Den Dozenten interessiert es nicht einmal! Trotzdem funktioniert es.

66

## Themen dieser Unit



Warum eigentlich OO?	Objekte	Modellieren
<ul style="list-style-type: none"><li>• Beherrschung von Komplexität</li><li>• Kapselung</li><li>• Polymorphie</li><li>• Abstraktion</li></ul>	<ul style="list-style-type: none"><li>• haben ein Verhalten</li><li>• haben einen (gekapselten) Zustand</li><li>• können kommunizieren</li><li>• sind unterschiedlich (aber ähnlich, bzw. polymorph)</li></ul>	<ul style="list-style-type: none"><li>• Objekte schützen</li><li>• Objekte verknüpfen</li><li>• Objekte abstrahieren</li></ul>

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme **67**

67

## Objekte schützen

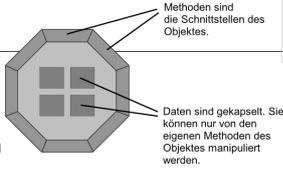




Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme **68**

68

## Konzept der Kapselung



Methoden sind die Schnittstellen des Objektes.

Daten sind gekapselt. Sie können nur von den eigenen Methoden des Objektes manipuliert werden.

In der Objektorientierung betrachtet man Daten und Methoden als eine zusammengehörende Einheit. Die folgenden Begriffe sind dabei von Bedeutung:

Abstraktion	Kapselung	Information Hiding
<ul style="list-style-type: none"> <li>• Komplexer Sachverhalt der realen Welt</li> <li>• wird auf das Wesentliche reduziert</li> <li>• und vereinfacht dargestellt</li> <li>• Datenfelder und Methoden eines Objekts repräsentieren diejenigen Daten und das Verhalten von Bedeutung für den Problemraum</li> </ul>	<ul style="list-style-type: none"> <li>• Objekt implementiert sein Verhalten in Schnittstellenmethoden</li> <li>• Ein Objekt sollte (im Idealfall) nur über definierte Schnittstellenmethoden mit seiner Umwelt in Kontakt treten</li> </ul>	<ul style="list-style-type: none"> <li>• Innere Daten eines Objekts sollen nach außen nicht direkt sichtbar sein</li> <li>• Innere Eigenschaften eines Objekts sollen verborgen sein</li> <li>• Ein Objekt sollten nichts von inneren Implementierungs-details eines anderen Objekts wissen müssen</li> </ul>

Ein Objekt sollte also keine Kenntnisse über den inneren Aufbau anderer Objekte haben. Programmiertechnische Änderungen innerhalb von Klassen (und daraus instantiierten Objekten) ziehen so keine Änderungen außerhalb der geänderten Klassen nach sich, solange die Schnittstellen gleich bleiben.

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

69

## Information Hiding

### 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

70

## 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\_datensfeld : Type  
 # protected\_datensfeld : Type  
 - private\_datensfeld : Type  
 ~ package\_datensfeld : Type

---

+ public\_methode() : Type  
 # protected\_methode() : Type  
 - private\_methode() : Type  
 ~ package\_methode() : Type

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

71

71

## Zugriffsschutz im Überblick



The diagram shows a central circle divided into four quadrants representing access modifiers: private, public, default, and protected. In the center is a box labeled 'Datenfeld oder Methode einer Klasse'. Arrows point from each quadrant to labels describing their visibility:
 

- private:** eigene Klasse (own class)
- public:** Klassen in anderen Paketen (classes in other packages)
- default:** Klassen im selben Paket (classes in the same package)
- protected:** Klassen im selben Paket (classes in the same package) and Sohnklassen in anderen Paketen auf geerbte Datenfelder und Methoden (subclass in other packages for inherited fields/methods)

Auf Besonderheiten im Zusammenhang mit Paketen und dem Zugriffsmodifikator default bitte Selbststudy Unit durcharbeiten.

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

72

72

## 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

73

73

## 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

74

74

**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 **75**

75

**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 **76**

76

**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

77

77

**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

78

78

**Miniübung:**    

Sie sollen nun Personen weiterhin wie folgt anlegen können.

```
Person p1 = new Person("Max", "Mustermann");  
Person p2 = new Person("Maren", "Musterfrau");  
Person p3 = new Person("Tessa", "Loniki");
```

Jedoch auf die einzelnen Namensbestandteile zielgerichtet zugreifen können.

```
System.out.println(p2.getNachname());  
System.out.println(p1.getVorname());  
System.out.println(p3.getVorname() + " " + p3.getNachname());
```

Es soll folgendes auf der Konsole ausgegeben werden.

```
Musterfrau  
Max  
Tessa Loniki
```

Bitte geben Sie eine Implementierung für Person an, die entsprechende getter Methoden implementiert.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme **79**

79

**Miniübung:**    

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme **80**

80

**Miniübung:**    

Sie sollen nun Personen weiterhin wie folgt anlegen können.

```
Person p1 = new Person("Max", "Mustermann");  
Person p2 = new Person("Maren", "Musterfrau");  
Person p3 = new Person("Tessa", "Loniki");
```

Jedoch nachträglich Nachnamen sinnvoll ändern können.

```
p2.setNachname("Mustermann");  
System.out.println(p2);
```

Es soll dann folgendes auf der Konsole ausgegeben werden.

```
Maren Mustermann
```

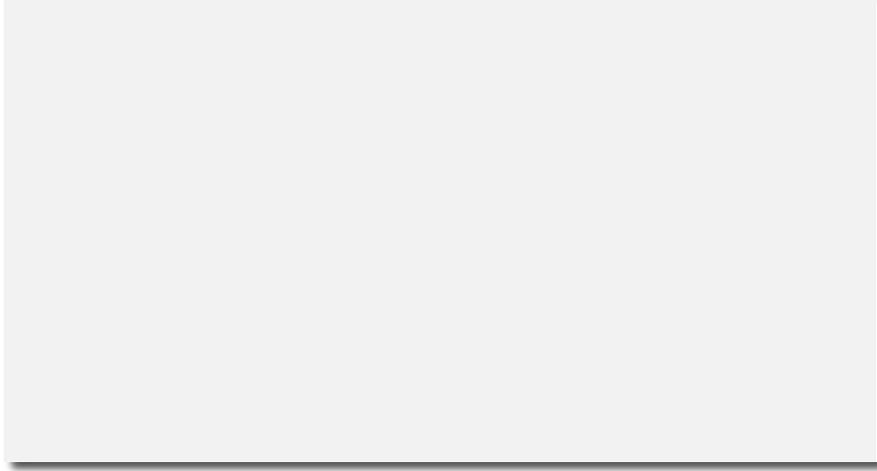
Werden sinnlose Werte wie "" oder null als Nachname gesetzt, soll nichts im Objekt geändert werden. Die Methode soll aber false als Rückgabe liefern. Wird etwas geändert, soll sie true liefern.

```
p1.setNachname("") == false    => p1 bleibt Max Mustermann  
p1.setNachname(null) == false  => p1 bleibt Max Mustermann  
p1.setNachname("Müller") == true => p1 wird Max Müller
```

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

81

**Miniübung:**    



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

82

## Objekte verknüpfen



Quelle: Pixabay

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

83

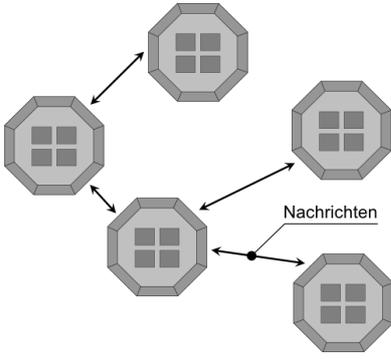
83

## Zusammenarbeit von Objekten

### Objektkommunikation



- Objektorientierte Systeme erbringen ihre Leistung durch das Zusammenwirken von Objekten
- in dem Nachrichten zwischen Objekten ausgetauscht werden
- (in JAVA entspricht dies Methodenaufrufen)



Nachrichten

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

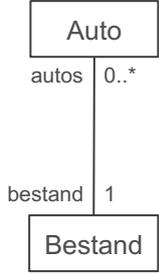
84

84

## Assoziation zwischen Objekten



### Assoziation in JAVA



```
class Auto {
    Bestand bestand; // Verweist auf einen Bestand
    ...
}

class Bestand {
    List<Auto> autos = new LinkedList<Auto>();
    // Verweist auf eine Liste von Autos
    ...
}
```

**Assoziationen sind erforderlich, damit Objekte miteinander kommunizieren können (hierzu benötigen sie eine Kenntnisbeziehungen von einander).**

Programmiertechnisch, wird üblicherweise eine Assoziation mit Hilfe zweier Variablen erzeugt, die Referenzen zwischen den Objekten halten.

- Für die Konnektivitäten 0..1 (keine oder eine Verbindung) und 1 (genau eine Verbindung) kann dabei einfach eine Referenzvariable genutzt werden.
- Für Konnektivitäten > 1 muss eine Datenstruktur gewählt werden, die mehr als einen Verweis aufnehmen kann. Üblicherweise wird hier eine Liste/Array genutzt.

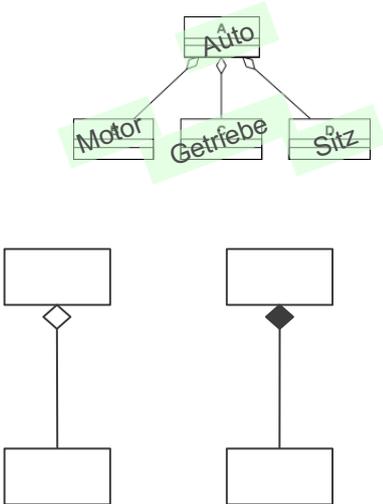
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

85

85

## Zerlegungshierarchie





- Ein Objekt kann als Datenfelder andere Objekte haben
- Z.B. ein Auto besteht aus einem Motor, Getriebe und Sitzen (sowie weiteren Teilen)
- Man kann ein Objekt in seine Teilobjekte und diese wiederum in ihre Teilobjekte zerlegen (usw.).
- Bei dieser Zerlegung unterscheidet man Aggregationen und Kompositionen (kommt gleich)

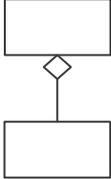
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

86

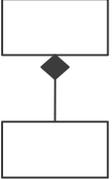
86

### Zerlegungshierarchie Aggregation und Komposition (Spezialformen von Assoziationen)





*Aggregation in UML*



*Komposition in UML*

- Bei einer Aggregation können die Bestandteile eines Objekts unabhängig von der Lebensdauer des Oberobjekts existieren
- **Teile können länger leben als das Ganze**
- **Beispiel:** Die Räder eines Autos können an andere Autos gebaut werden. Räder sind an ein Auto aggregiert (zugeordnet).

- Bei einer Komposition existieren die Bestandteile eines Objekts nur so lange wie auch das Oberobjekt existiert.
- **Teile können nicht länger leben als das Ganze**
- **Beispiel:** Die Seiten eines Buchs sind mit dem Buch untrennbar verbunden. Seiten und Buch sind komponiert.

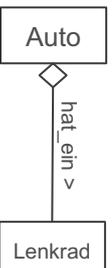
Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

87

### Aggregation/Komposition in UML/JAVA



*Aggregation in UML*



*Aggregation in JAVA*

```
class Auto {
    Lenkrad hat_ein;
    ...
}

class Lenkrad {
    ...
}

Auto auto = new Auto();
Lenkrad lenkrad = new Lenkrad();
auto.hat_ein = lenkrad;
```

Programmiertechnisch, wird üblicherweise eine Aggregation/Komposition mit Hilfe einer Variablen erzeugt, die eine Referenz auf das Teilobjekt enthält. Da JAVA nur Referenztypen kennt, geht dies in JAVA sehr einfach (siehe oben). Solch eine Variable wird auch **Referenzvariable** (ergänzend zu Instanz- und Klassenvariable genannt).

**Kompositionen werden in der Regel genauso umgesetzt, aber beim Löschen wird auch das Komposit mitgelöscht.**

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

88

## Multiplizitäten



Multiplizität	Beschreibung
1	Genau eine Verbindung
0..1	Höchstens eine Verbindung
0..*	Beliebig viele Verbindungen
1..*	Mindestens eine Verbindung
n..m	Mindestens n höchstens m Verbindungen. Eher ungewöhnlich, nur zu nutzen wenn die Obergrenze zweifelsfrei feststeht, z.B. die Anzahl an Reifen an einem PKW hätte die Multiplizität 0..4. Häufig nutzt man in solchen Fällen dennoch die Multiplizität 0..*.

Assoziationen erhalten neben einem Namen auch Anzahlangaben (Multiplizitätsangaben). Dies gibt an mit wievielen Objekten der gegenüberliegenden Assoziationsseite je ein Objekt der Ausgangsseite verbunden ist.

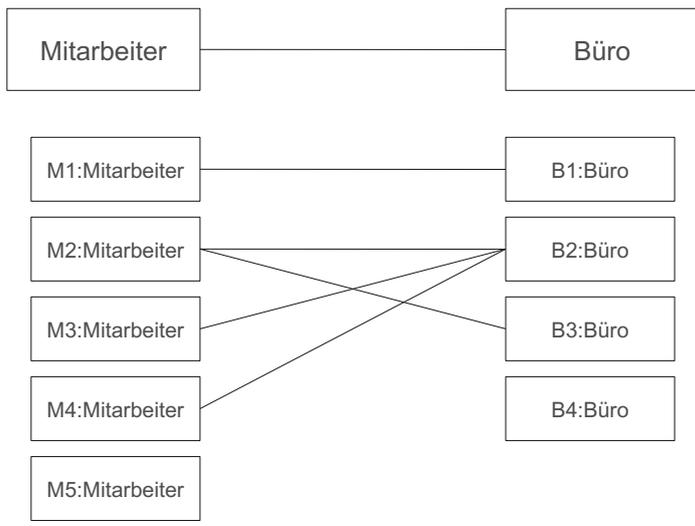
Letztlich entscheiden diese Angaben, ob zum Verwalten der Kenntnisbeziehungen zwischen Objekten eine einfache Referenzvariable oder eine Collection über den Typ des Assoziationspartners genutzt werden muss.

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

89

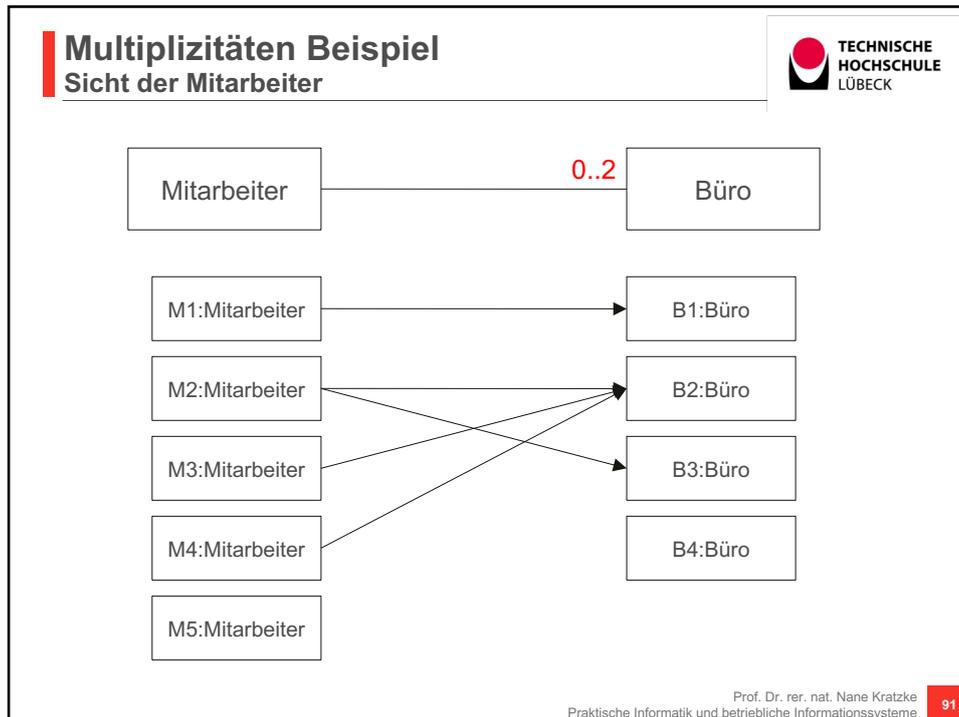
## Multiplizitäten Beispiel



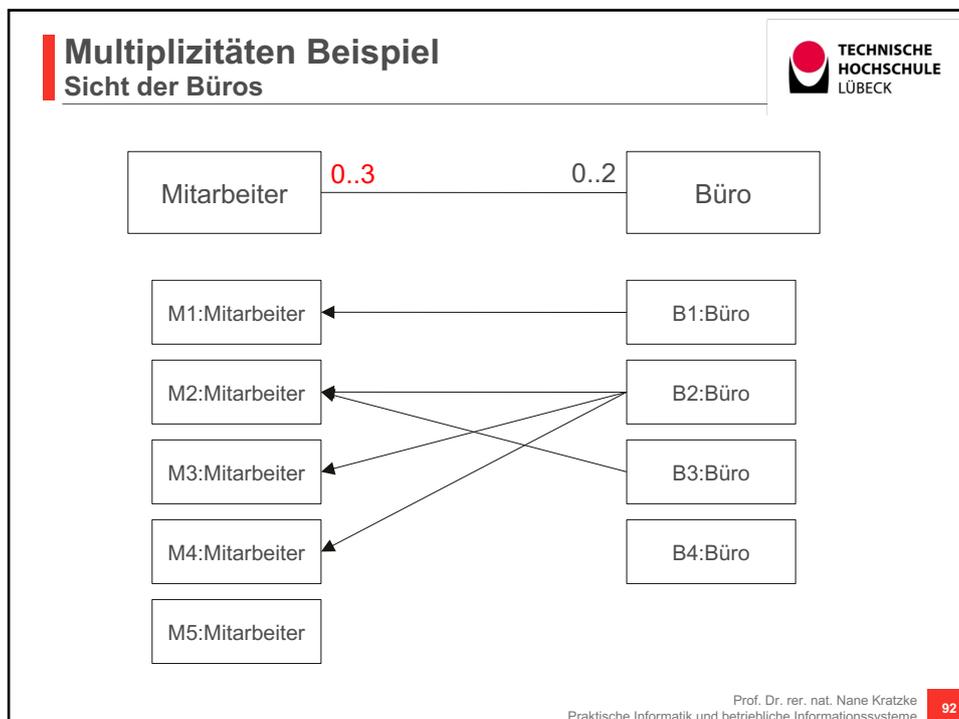


Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

90



91



92

### Multiplizitäten Beispiel Angabe der Multiplizitäten

Konkrete Multiplizitäten > 1 werden üblicherweise verallgemeinert.

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

93

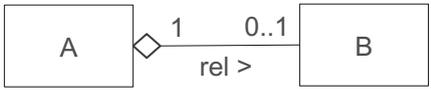
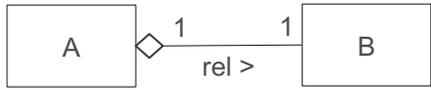
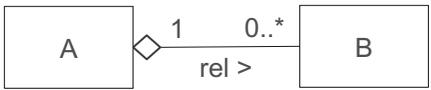
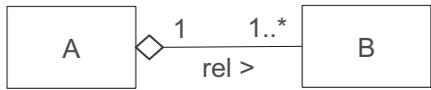
### Transformationsregeln von Assoziationen

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

94

### Transformationsregeln von Aggregationen/Kompositionen



 <pre>class A {     B rel;     ... }</pre> <pre>class B {     ... }</pre>	 <pre>class A {     B rel;     ... }</pre> <pre>class B {     ... }</pre>
 <pre>class A {     List&lt;B&gt; rel;     ... }</pre> <pre>class B {     ... }</pre>	 <pre>class A {     List&lt;B&gt; rel;     ... }</pre> <pre>class B {     ... }</pre>

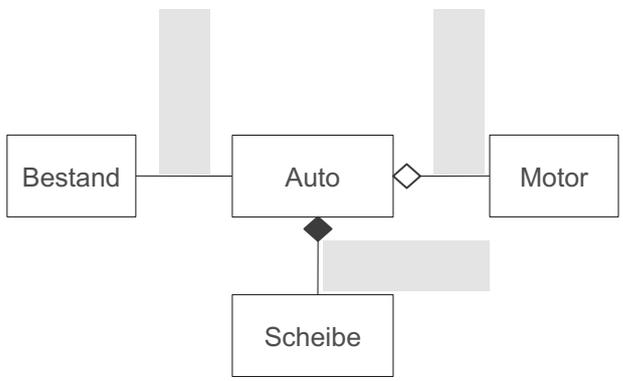
Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme **95**

95

### Miniübung:




Gegeben ist folgendes UML Diagramm. Welche Arten von Kenntnisbeziehungen sind zwischen den Klassen definiert worden?



Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme **96**

96

**Miniübung:**    

Studierende sollen wie folgt angelegt und ausgegeben werden können.

```
Student s = new Student("Max", "Mustermann", 123456);  
System.out.println(s);
```

```
Max Mustermann (MatrNr.: 123456)
```

Termine sollen wie folgt angelegt und ausgegeben werden können.

```
Termin t = new Termin(16, 15, 17, 45, "Übung VProg", "18-1.18");  
System.out.println(t);
```

```
16:15h bis 17:45h : Übung VProg in 18-1.18
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 97

97

**Miniübung:**    

Studierenden können ferner Termine wie folgt zugeordnet werden.

```
Student s = new Student("Max", "Mustermann", 123456);  
Termin t1 = new Termin(14, 30, 16, 00, "Vorlesung VProg", "18-0.01");  
Termin t2 = new Termin(16, 15, 17, 45, "Übung VProg", "18-1.18");  
s.insertTermin(t1);  
s.insertTermin(t2);  
s.insertTermin(t1); // Termin versehentlich doppelt eingegeben.
```

Werden nun Studierende ausgegeben, sollen auch die Termine mit ausgegeben werden, die einem Studierenden zugeordnet sind.

```
System.out.println(s);
```

```
Max Mustermann (MatrNr.: 123456)  
- 14:30h bis 16:00h : Vorlesung VProg in 18-0.01  
- 16:15h bis 17:45h : Übung VProg in 18-1.18
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 98

98

**Miniübung:**






Um sie zu unterstützen, ist Ihnen folgendes UML-Diagramm gegeben.

```

classDiagram
    class Student {
        - vorname : String
        - nachname : String
        - matrNr : int
        + Student(String, String, int)
        + insertTermin(Termin) : void
        + toString() : String
    }
    class Termin {
        - startHour : int
        - startMinutes : int
        - endHour : int
        - endMinutes : int
        - title : String
        - location : String
        + Termin(int, int, int, int, String, String)
        + toString() : String
    }
    Student "1" -- "0..*" Termin : # appointments
    
```

Implementieren Sie nun bitte `Student` und `Termin`.

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

99

**Miniübung:**






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

100

**Miniübung:**






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

101

101

**Miniübung:**






Gegeben sei folgendes UML Diagramm:

- Implementieren Sie die gezeigten Klassen inklusive ihrer Assoziationen.
- Bestimmen sie die TOP 10 der besten Kunden eines jeden Shops.
- `marketing()` schreibt 5% aller insgesamt vorhanden Kunden an und schlägt 25% aller Produkte vor.
- Kunden reagieren auf 10% aller vorgeschlagenen Produkte (Methode `mail()`) mit einem Kauf.

```

classDiagram
    class Shop {
        - String : name
        + Shop(String)
        + register(Customer) : boolean
        + marketing() : void
        + toString() : String
        + topCustomers() : List<Customer>
    }
    class Customer {
        - name : String
        - street : String
        - town : String
        - postCode : String
        + ALL : List<Customer> { static }
        + Customer(String, String, String, String, String)
        + buyList<Product>(Shop) : boolean
        + mail(List<Product>, Shop) : void
        + getOrders() : List<Order>
        + getValue() : double
        + getPostCode() : String
        + toString() : String
    }
    class Order {
        + Order(Customer, List<Product>, Shop)
        + getProducts() : List<Product>
        + getShop() : Shop
        + getCustomer() : Customer
        + getValue() : double
    }
    class Product {
        - name : String
        - double : price
        + ALL : List<Product> { static }
        + Product(String, double)
        + getPrice() : double
        + getName() : String
    }
    Shop "1" -- "0..*" Customer : customers
    Shop "1" -- "0..*" Order : - shop
    Customer "1" -- "0..*" Order : # customer
    Order "0..*" -- "1..*" Product : - products
    
```

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

102

102

## Miniübung:



Testen Sie ihre Implementierung mit folgendem Setting:

```
// Wir erzeugen 100 Kunden
for (int i = 1; i <= 100; i++) {
    new Customer("Max Mustermann " + i, "Beispielstr. " + (i % 27), "Luebeck", 26500 + (i % 43) + "");
}
// Wir erzeugen 1000 Produkte mit zufaelligen Preisen
for (int i = 1; i <= 1000; i++) { new Product("Testprodukt " + i, Math.random() * i); }
// Wir erzeugen 3 Shops
List<Shop> shops = Arrays.asList(new Shop("Amazon"), new Shop("Otto"), new Shop("Ebay"));
// 1. Marketing Runde
for (Shop shop : shops) { shop.marketing(); System.out.println(shop); }
// 2. Marketing Runde
for (Shop shop : shops) { shop.marketing(); System.out.println(shop); }
// 3. Marketing Runde
for (Shop shop : shops) { shop.marketing(); System.out.println(shop); }
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 103

103

## Objekte abstrahieren



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 104

104

## Abstraktion zur Bildung von Hierarchien



- Information Hiding ist ein effizientes Mittel um Komplexität zu beherrschen
- Ein weiteres Mittel ist die Bildung von **Hierarchien**
- Die Objektorientierung kennt im Kern zwei Hierarchieformen:

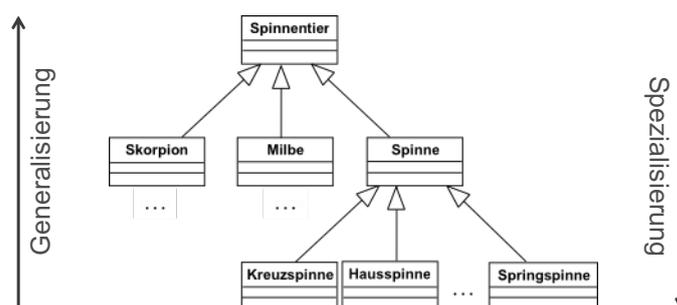
Vererbungshierarchie
<ul style="list-style-type: none"> <li>• Kind of-Hierarchie</li> <li>• Is a-Hierarchie</li> <li>• Anordnung von Klassen in Kategorieebenen(-bäumen)</li> </ul>

Zerlegungshierarchie
<ul style="list-style-type: none"> <li>• Part of-Hierarchie</li> <li>• Betrachtung von zusammengesetzten Objekten in Form von</li> <li>• Aggregationen</li> <li>• Kompositionen</li> </ul>

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 105

105

## Vererbungshierarchien (I)



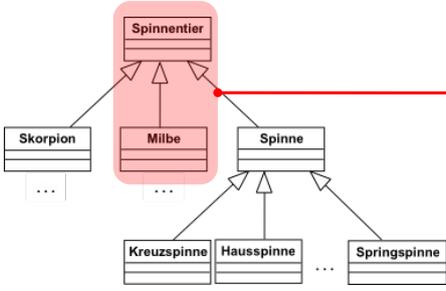
**Darstellung von Vererbungshierarchien in UML:**  
**Pfeil bedeutet bspw. Skorpion ist Unterklasse von Spinnentier**  
 Kann auch so gelesen werden: Skorpion (spezieller) ist ein Spinnentier (genereller), daher auch der Name „is a-Hierarchie“

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 106

106

## Vererbungshierarchien (II)





*Darstellung von Vererbungshierarchien in UML*

```
class Spinnentier {
    ...
}
class Milbe extends Spinnentier {
    ...
}
```

*Ausdrücken einer Vererbung in JAVA (nur der markierte Ausschnitt)*

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

107

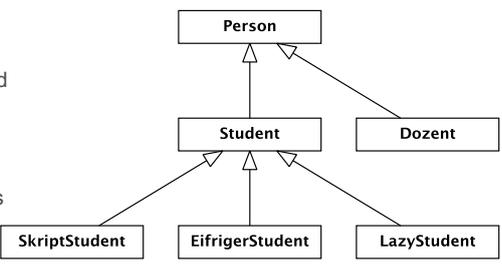
107

## Klassen sind Datentypen für Referenzen



Ist beispielsweise folgendes UML Diagramm gegeben, so ergibt sich daraus, das Studenten und Dozenten Personen sind. SkriptStudenten, EifrigeStudenten und LazyStudenten sind Studenten und damit ebenfalls Personen.

Ein EifrigerStudent kann damit generell als Person, spezifischer als Student oder auch sehr spezifisch als EifrigerStudent angesprochen (referenziert) werden.



```
EifrigerStudent s = new EifrigerStudent("Max");
Student t = new EifrigerStudent("Moritz");
Person p = new EifrigerStudent("Tessa");
```

*Referenztyp*
*Objektyp*

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

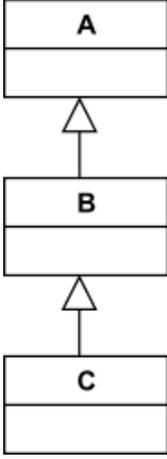
108

108

## Besonderheiten bei der Vererbung



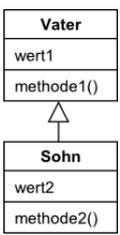
- Für den Einsatz der Vererbung muss man Kenntnisse über die Typkonvertierungen haben
- Wichtig: Ein Sohnobjekt ist immer vom Typ der eigenen Klasse, als auch vom Typ der Vaterklasse, der Vaternvaterklasse, etc.
- Somit kann ein Objekt durchaus mehrere Typen haben.



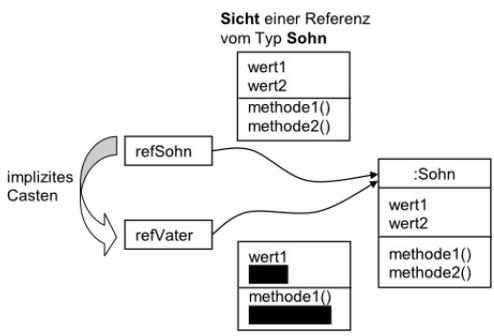
Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

109

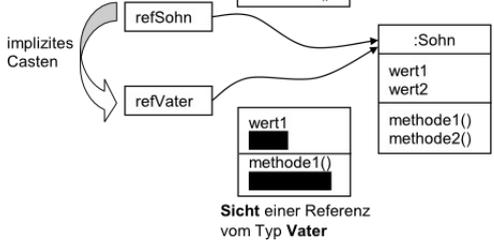
## Implizites „Upcasten“

**Sicht einer Referenz vom Typ Sohn**



**Sicht einer Referenz vom Typ Vater**



implizites Casten

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

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

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

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

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

110

## Explizites „Downcasten“



```

class Vater {
    wert1;
    methode1();
}
class Sohn {
    wert2;
    methode2();
}
            
```

```

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

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

111

## Casting im Überblick

### Zulässige implizite und explizite Type Casts



```

class A {
}
class B {
}
class C {
}
            
```

Wenn oben stehende Klassenhierarchie gilt, dann sind die neben stehenden Cast Operationen zulässig

Funktioniert eine explizite Cast Operation zur Laufzeit nicht, wird eine Exception vom Typ `ClassCastException` geworfen. Implizite Casts können bereits zur Kompilierzeit geprüft werden.

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

112

### Miniübung:






```

classDiagram
    class Object
    class D
    class A
    class B
    class C
    class Object <|-- D
    class Object <|-- A
    class A <|-- B
    class B <|-- C
            
```

<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 113

113

### Abstrakte Klassen



```

class SkriptStudent {
+ SkriptStudent(String)
+ notiere(String) : void
}
            
```

```

class EifrigerStudent {
+ EifrigerStudent(String)
+ notiere(String) : void
}
            
```

```

class LazyStudent {
+ LazyStudent(String)
+ notiere(String) : void
}
            
```

```

class TiltedStudent {
- units : int
+ LazyStudent(String)
+ notiere(String) : void
}
            
```

```

class EmotionalerStudent {
- String[] empfindungen
+ EmotionalerStudent(String)
+ notiere(String) : void
}
            
```

```

class Dozent {
- String[] unit
- String[] antworten
- fragen : int
+ unterrichte(Student) : void
+ gebeUnit(int) : void
+ beantworte(String) : String
}

class Student {
- String name
- List<String> notizen
- String[] fragen
+ Student(String)
+ hoere(Dozent) : void
+ notiere(String) : void
+ unverstaendnis() : void
+ toString() : String
}

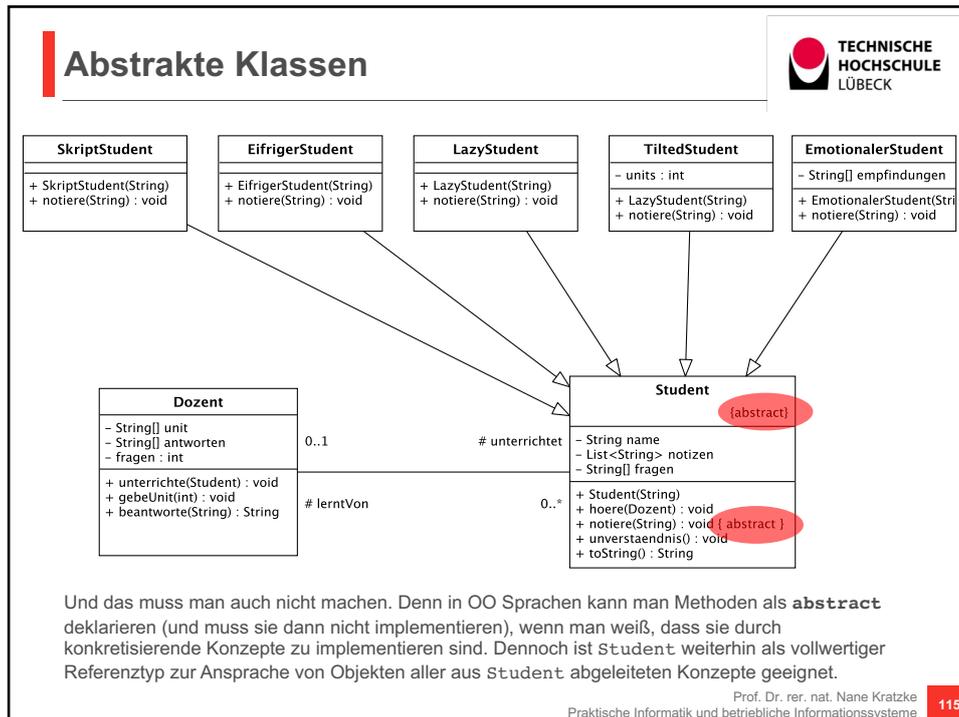
Dozent "0..1" -- "# unterrichtet" Student "0..*"
Dozent "0..1" -- "# lerntVon" Student "0..*"
            
```

In unserem Polymorphie Beispiel haben diverse Spezialisierungen des generellen Konzepts Student jeweils das `notiere()` Verhalten (Methode) neu implementiert. Die ursprüngliche `notiere()` Implementierung wird gar nicht mehr genutzt.

**Es stellt sich daher die Frage, wieso diese dann überhaupt implementieren?**

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 114

114



115

## Ein abstrakter Student



```

public abstract class Student {
    private String name;
    private String[] fragen = { "Gibt es dazu mal ein Beispiel?",
        "Das war mir viel zu schnell!", "Fehlt da nicht ein Semikolon?" };
    private List<String> notizen = new LinkedList<String>();
    protected Dozent lerntVon;

    public Student(String n) { this.name = n; }
    public void hoere(Dozent d) { this.lerntVon = d; }
    public void unverstaendnis() {
        if (this.lerntVon == null) return;
        Random r = new Random();
        String frage = this.fragen[r.nextInt(this.fragen.length)];
        String antwort = this.lerntVon.beantworte(frage);
        System.out.println(this.name + ": " + frage + " Dozent: " + antwort);
    }
    public abstract void notiere(String s);
    public String toString() {
        String ret = "Notizen von: " + name + "\n";
        for (String notiz : notizen) ret += notiz + "\n"; return ret;
    }
}
    
```

Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 116

116

## 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 (bspw. für Extension Points).

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme 117

117

## Finale Methoden und finale Klassen Das Gegenstück zu **abstract**



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

### Deklaration finaler Methoden

```
class C {  
  
    public void aenderbareMethode() { ... }  
  
    public final void finaleMethode() { ... }  
}
```

### Deklaration finaler Klassen

```
final class C {  
  
    ...  
}
```

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 118

118

## Schnittstellen

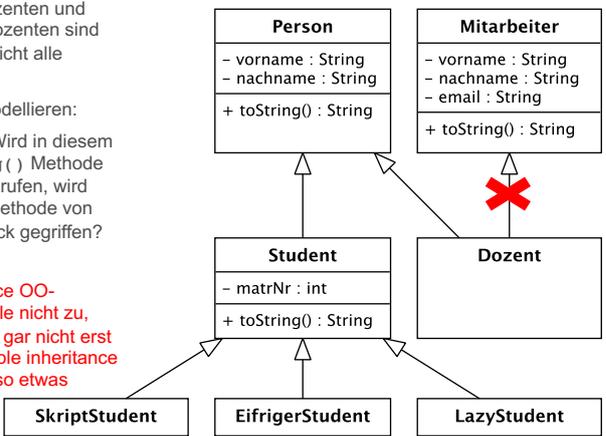


Nun zu diesem Problem: Dozenten und Studenten sind Personen. Dozenten sind aber auch Mitarbeiter. Aber nicht alle Studenten sind Mitarbeiter.

Man könnte dies wie folgt modellieren:

Es bleibt aber ein Problem. Wird in diesem Beispiel bspw. die `toString()` Methode eines Dozentenobjekts aufgerufen, wird dann auf die `toString()` Methode von `Person` oder `Mitarbeiter` zurück gegriffen?

Java ist eine single inheritance OO-Sprache und lässt solche Fälle nicht zu, um oben stehendes Problem gar nicht erst entstehen zu lassen (in multiple inheritance Sprachen, bspw. C++, kann so etwas jedoch auftreten).



Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 119

119

## Schnittstellen



Bei solchen Problemen bietet sich der Einsatz von Schnittstellen an.

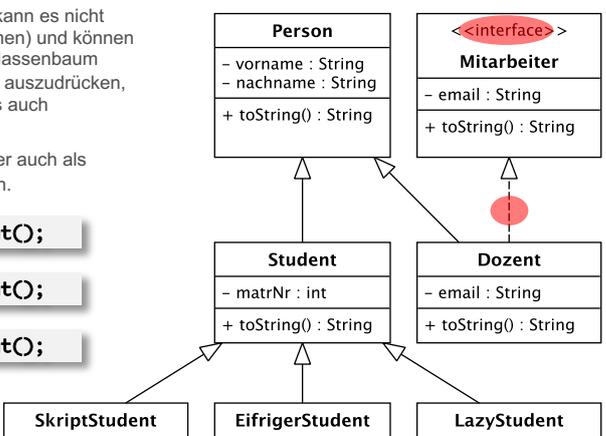
Schnittstellen sind sozusagen voll abstrakte Klassen (d.h. abstrakte Klassen haben keine implementierten Methoden, damit kann es nicht zum multiple inheritance Fall kommen) und können daher an beliebiger Stelle einem Klassenbaum „hinzugemischt“ werden, um bspw. auszudrücken, dass ein Dozent sowohl `Person` als auch `Mitarbeiter` ist.

Also als `Person`, als `Mitarbeiter` oder auch als `Dozent` angesprochen werden kann.

Mitarbeiter m = `new Dozent();`

Person p = `new Dozent();`

Dozent d = `new Dozent();`



Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme 120

120

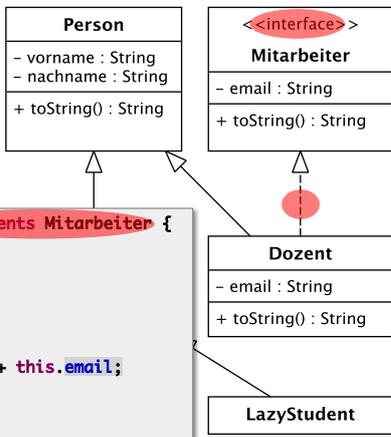
## Ein Dozent ist Person und Mitarbeiter



```

public interface Mitarbeiter {
    private String email;
    public String toString();
}

public class Dozent extends Person implements Mitarbeiter {
    private String email;
    public String toString() {
        return super.toString() + " EMail: " + this.email;
    }
}
    
```



**Hinweis:** Eine Klasse kann beliebige viele Schnittstellen implementieren (**implements**) aber nur eine Klasse erweitern (**extends**).

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

121

## Neu in Java 8: Default Implementierungen in Schnittstellen



```

public interface Person {
    public String getName();
    default String sayHello() {
        return "Hi, my name is " + getName();
    }
}

public class Dozent implements Person {
    private String name = "Max Mustermann";
    public String getName() {
        return this.name;
    }
}

Dozent d = new Dozent();
System.out.println(d.sayHello());

Hi, my name is Max Mustermann
    
```

Seit Java 8 können nun default Implementierungen in Schnittstellen vorgesehen werden. Werden diese nicht überschrieben, erben die eine Schnittstelle implementierenden Klassen diese. Default Methoden können allerdings nicht direkt auf Datenfelder eines Objekts zugreifen, sondern nur mittels der Schnittstelle bekannten Methoden. So werden viele Probleme der Mehrfachvererbung umgangen.

**Default Methoden sind immer automatisch public.**

**Warnung:** Default Methoden ermöglichen Mehrfachvererbung, und ziehen damit alle Probleme der Mehrfachvererbung mit sich.



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

122

### Das war viel Theorie ...

Jetzt noch ein kleines Beispiel zur Modellbildung mit Klassen



Quelle: Pixabay

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

123

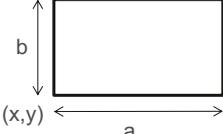
123

### Veranschaulichung an einem Beispiel

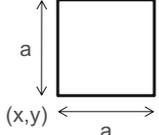
#### Fächenberechnung von Figuren



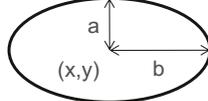
Rechteck



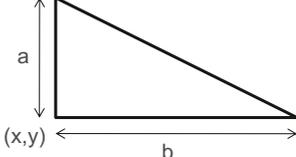
Quadrat



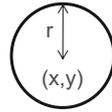
Ellipse



Rechtwinkliges Dreieck



Kreis



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

124

124

### Flächenberechnung von Figuren

**Was haben alle Figuren gemeinsam?**

Rechteck

Quadrat

Ellipse

Rechtwinkliges Dreieck

Kreis

**Einen Bezugspunkt**

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

125

125

### Flächenberechnung von Figuren

**Was haben viele Figuren gemeinsam?**

Rechteck

Quadrat

Ellipse

Rechtwinkliges Dreieck

Kreis

**Zwei Längenangaben**

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

126

126

## Flächenberechnung von Figuren

Welche Figuren sind Spezialfälle anderer Figuren

Rechteck:  $(x,y)$ ,  $a$ ,  $b$

Quadrat:  $(x,y)$ ,  $a$

Ellipse:  $(x,y)$ ,  $a$ ,  $b$

Kreis:  $(x,y)$ ,  $r$

Rechtwinkliges Dreieck:  $(x,y)$ ,  $a$ ,  $b$

Ein Quadrat ist ein spezielles Rechteck  
 Ein Kreis ist eine spezielle Ellipse

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

127

## Flächenberechnung von Figuren

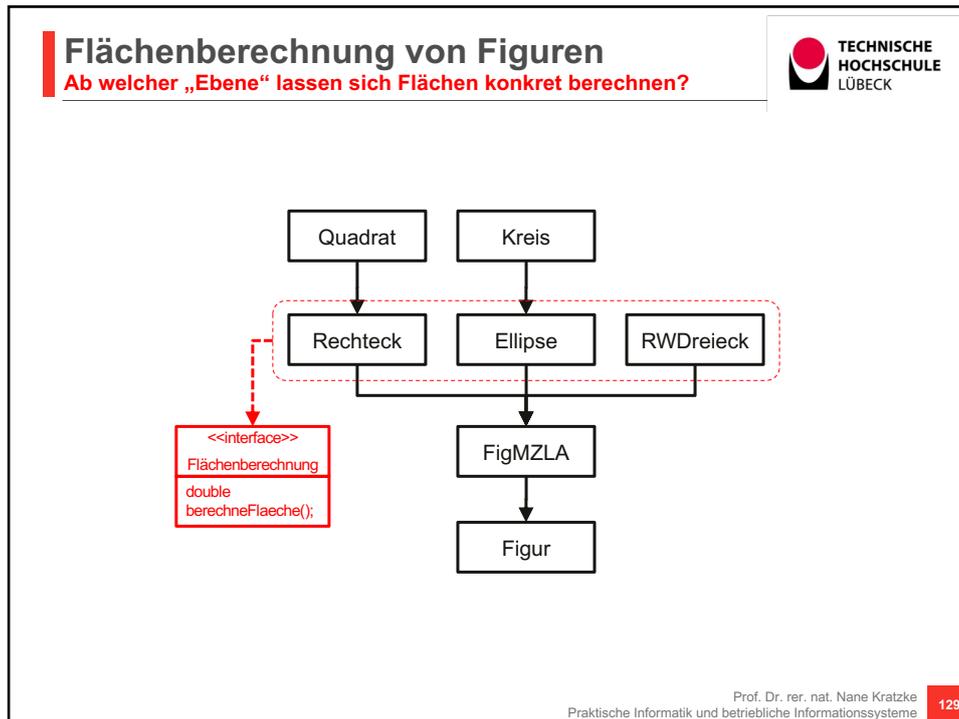
- Es gibt Figuren.
- Figuren mit zwei Längenangaben sind Figuren.
- Rechteck, Ellipse und rechtwinkliges Dreieck sind Figuren mit zwei Längenangaben.
- Ein Quadrat ist ein Rechteck.
- Ein Kreis ist eine Ellipse.

```

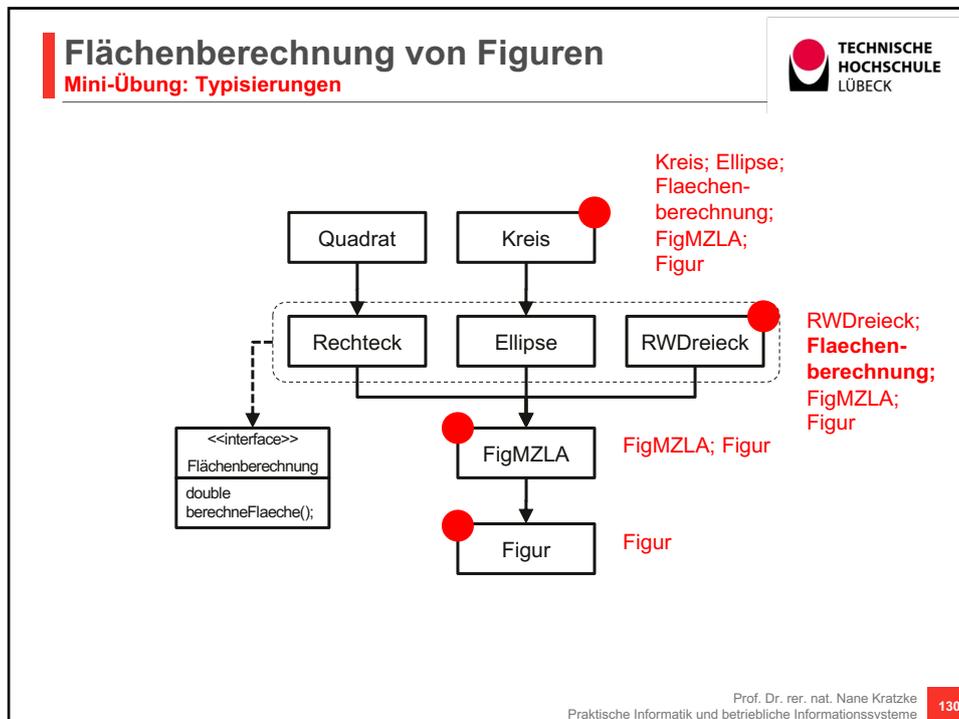
    graph TD
        Quadrat --> Rechteck
        Kreis --> Ellipse
        Rechteck --> FigMZLA
        Ellipse --> FigMZLA
        RWDreieck --> FigMZLA
        FigMZLA --> Figur
    
```

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

128



129



130

## Flächenberechnung von Figuren

### Einführung von Figuren

TECHNISCHE HOCHSCHULE LÜBECK

```

public class Figur {
    protected int X = 0;
    protected int Y = 0;
    public Figur(int x, int y)
    {
        this.X = x;
        this.Y = y;
    }
}
    
```

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

131

131

## Flächenberechnung von Figuren

### Einführung von Figuren mit zwei Längenangaben

TECHNISCHE HOCHSCHULE LÜBECK

```

public class FigMZLA
    extends Figur
{
    protected int A = 0;
    protected int B = 0;
    public FigMZLA(int x,
        int y,
        int a,
        int b)
    {
        super(x,y);
        this.A = a;
        this.B = b;
    }
}
    
```

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

132

132

## Flächenberechnung von Figuren

### Einführung einer Flächenberechnungsschnittstelle

TECHNISCHE HOCHSCHULE LÜBECK

```
public interface
Flaechenberechnung {
    double berechneFlaeche();
}
```

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

133

## Flächenberechnung von Figuren

### Implementierungen von Rechteck

TECHNISCHE HOCHSCHULE LÜBECK

Berechnung der Fläche eines Rechtecks mit  
 Seitenlängen a und b?

```
public class Rechteck
extends FigMZLA
implements Flaechenberechnung
{
    public Rechteck(int x,
                    int y,
                    int a,
                    int b)
    { super(x, y, a, b); }

    public double
    berechneFlaeche()
    { return Math.abs(A * B); }
}
```

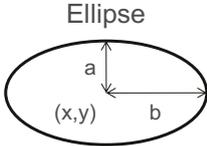
Prof. Dr. rer. nat. Nane Kratzke  
 Praktische Informatik und betriebliche Informationssysteme

134

## Flächenberechnung von Figuren

### Implementierung von Ellipse





Ellipse

Berechnung der Fläche einer Ellipse mit Radien a und b?

$$\pi * a * b$$

```

public class Ellipse
    extends FigMZLA
    implements Flaechenberechnung {
    public Ellipse(int x,
                  int y,
                  int a,
                  int b)
    { super(x, y, a, b);
    }

    public double berechneFlaeche()
    { return Math.abs(Math.PI * A * B);
    }
}
    
```

Quadrat

Rechteck

Kreis

Ellipse

RWDreieck

FigMZLA

<<interface>>  
 Flaechenberechnung  
 double berechneFlaeche();

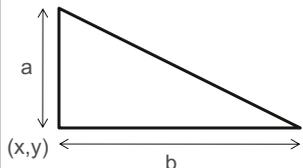
Figur

135

## Flächenberechnung von Figuren

### Implementierung eines rechtwinkligen Dreiecks





Rechtwinkliges Dreieck

Berechnung eines rechtwinkligen Dreiecks mit den Seitenlängen a und b?

$$a * b / 2$$

```

public class RWDreieck
    extends FigMZLA
    implements Flaechenberechnung {
    public RWDreieck(int x,
                    int y,
                    int a,
                    int b)
    { super(x, y, a, b);
    }

    public double berechneFlaeche()
    { return Math.abs(
        A * B / 2.0);
    }
}
    
```

Quadrat

Rechteck

Kreis

Ellipse

RWDreieck

FigMZLA

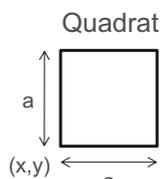
<<interface>>  
 Flaechenberechnung  
 double berechneFlaeche();

Figur

136

## Flächenberechnung von Figuren

### Implementierung von Quadrat

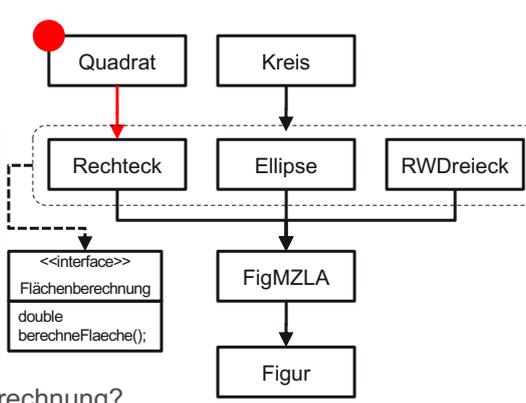


Berechnung der Fläche eines Quadrats mit der Seitenlänge  $a$ ?

$$a^2$$

```

public class Quadrat
    extends Rechteck
    {
        public Quadrat(int x,
                       int y,
                       int a)
        { super(x, y, a, a);
        }
    }
    
```



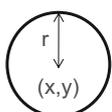
Und wo erfolgt die Flächenberechnung?  
 In der Klasse Rechteck

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

137

## Flächenberechnung von Figuren

### Implementierung von Kreis

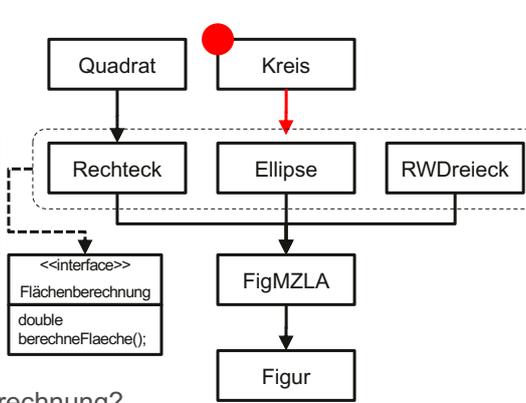


Berechnung der Fläche eines Kreises mit dem Radius  $r$ ?

$$\pi r^2$$

```

public class Kreis
    extends Ellipse
    {
        public Kreis(int x,
                    int y,
                    int r)
        { super(x, y, r, r);
        }
    }
    
```



Und wo erfolgt die Flächenberechnung?  
 In der Klasse Ellipse

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

138

## Erzeugung von Objekten

### Verfolgen von Konstruktoraufrufen

```

            public class Kreis extends Ellipse
            {
                public Kreis(int x, int y, int r) {
                    super(x, y, r, r);
                }
            }

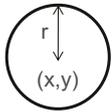
            public class Ellipse extends FigMZLA
            {
                public Ellipse(int x, int y, int a, int b) {
                    super(x, y, a, b);
                }
            }

            public class FigMZLA extends Figur
            {
                public FigMZLA(int x, int y, int a, int b) {
                    super(x, y); this.A = a; this.B = b;
                }
            }

            public class Figur
            {
                public Figur(int x, int y) {
                    this.X = x; this.Y = y;
                }
            }
        
```



Kreis



Kreis

↓

Ellipse

↓

FigMZLA

↓

Figur

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

139

## Arbeiten mit Objekten

### Verfolgen von Methodenaufrufen

```

            Kreis k = new Kreis(5, 5, 10);
            double flaeche = k.berechneFlaeche();

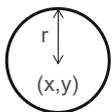
            public class Kreis extends Ellipse
            {
                public Kreis(int x, int y, int r) {
                    super(x, y, r, r);
                }
            }

            public class Ellipse extends FigMZLA
                implements Flaechenberechnung
            {
                public Ellipse(int x, int y, int a, int b) {
                    super(x, y, a, b);
                }

                public double berechneFlaeche() {
                    return Math.abs(Math.PI * this.A * this.B);
                }
            }
        
```



Kreis



Kreis

↓

Ellipse

↓

FigMZLA

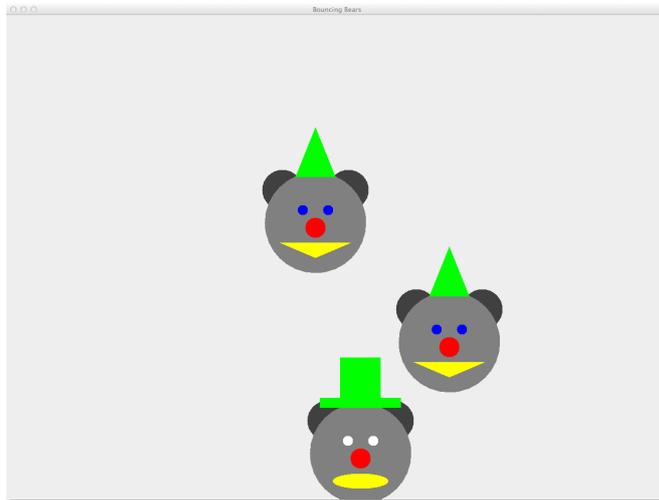
↓

Figur

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

140

## Das alles erweitern wir noch zu ...



141

## Zusammenfassung

A+

- Grundsatz der Objektorientierung: Denken in Objekten
  - Klassen sind Baupläne
  - Objekte sind konkrete Ausprägungen dieser Baupläne
  - Objekte kommunizieren miteinander (Methoden) um ein Problem zu lösen
- Objekte haben ein **Verhalten** (Methoden)
- Objekte haben einen (gekapselten) **Zustand** (Datenfelder)
- Objekte können **kommunizieren** (Methodenaufrufe entlang ihrer Assoziationen)
  - Assoziationen
  - Part-of-Hierarchien
- Objekte sind vielgestaltig (**polymorph**)
  - Abstraktion entlang von
  - Vererbungshierarchien (is a-Hierarchien)



142