



**Hochschule  
Albstadt-Sigmaringen**  
University of Applied Sciences

Fakultät Informatik

# Programmierung 2

Teil 1 – TI/TIB/ITS/WIN

---

# Inhaltsverzeichnis

Vorwort

Einstieg in Java

Einstieg in die OOP: Klassen und Objekte/Instanzen

Die Klasse String

Graphische Darstellung von Klassen mit der UML – Teil 1

Vererbungsbeziehungen

Exceptions

Einstieg Streams

Fakultät Informatik

# Programmierung 2

Vorwort



## Vorwort

Einstieg in Java

Einstieg in die OOP: Klassen und Objekte/Instanzen

Die Klasse String

Graphische Darstellung von Klassen mit der UML – Teil 1

Vererbungsbeziehungen

Exceptions



---

# Einstieg Streams

---

# Historie des Foliensatzes

- Version 1: Sommersemester 2018, Prof. Dr. Ute Matecki
- Version 2: Sommersemester 2019, lektoriert und überarbeitet von
  - M.Comp.Sc. Peter Stumfol
  - Prof.Dr. Ute Matecki
- Version 3: Sommersemester 2020, nochmals lektoriert von
  - M.Comp.Sc. Peter Stumfol
  - Prof.Dr. Ute Matecki

---

## Arbeitshinweise zum Foliensatz

- Rasch eine eigene Java-Umgebung daheim installieren (siehe Folge-Kapitel).
- Vorlesungsbeispiele selbst (!!!) eingeben und zum Laufen bringen.
- Vorlesungsbispiel modifizieren und zum Laufen bringen.

---

## Arbeitshinweise zum Foliensatz

- Regelmäßig (!!!) die Praktikumsaufgaben bearbeiten.
- Falls zusätzliches Übungsmaterial in der Vorlesung besprochen wird  
wird: Selbst noch mal mit eigener Lösung programmieren!
- Programmieren lernen Sie nur, indem Sie programmieren!
- In diesem Sinne: Viel Spaß und viel Erfolg beim Erarbeiten der  
Programmiersprache Java!

Fakultät Informatik

# Programmierung 2

Einstieg in Java

## Vorwort

# Einstieg in Java

Installationsvoraussetzungen

Programmstruktur – Python vs. Java

Programm-Erzeugung und -Start mit Java

Programm-Erzeugung und -Start auf der Kommandozeile

Programm-Erzeugung und -Start mit der Entwicklungsumgebung Eclipse

Java-API

Variablen, Datentypen und Operatoren

Grundlagen zu Variablenvereinbarungen

Elementare Datentypen in Java

Konstanten bei elementaren Datentypen – **final**

Aggregierte Datentypen in Java – Referenzdatentypen allgemein

Aggregierte Datentypen in Java – einfache Arrays

Aggregierte Datentypen in Java – Mehrdimensionale Arrays

Aggregierte Datentypen in Java – Strings

---

Aggregierte Datentypen in Java – Klassen

Die null-Referenz

Arithmetische Operatoren und Zuweisung – Python vs. Java

Vergleichsoperatoren und Logische Operatoren: Python vs. Java

Bitweise Operatoren: Python vs. Java

Typkonvertierung in Java

Sichtbarkeit von Variablen – Teil 1

## Kontrollstrukturen

Kontrollstrukturen in Java

Fallunterscheidung mit if – else if – else

Fallunterscheidung mit switch-case

Kopfgesteuerte Schleifen – while

Kopfgesteuerte Schleifen – for

Fußgesteuerte Schleifen – do – while

## Methoden in Java

Methoden in Java vs. Funktionen in Python

Signatur einer Methode

Aufruf von Methoden und Parameterübergabe



## Die Klasse String

Graphische Darstellung von Klassen mit der UML – Teil 1

Vererbungsbeziehungen

Exceptions

Einstieg Streams

# Java Development Kit (JDK)

## Bestandteile von Java und Bezug von Java

Das aktuelle **JDK** enthält unter anderem

- den Java-Compiler `javac.exe` (unter Windows) bzw. `javac` (unter Linux) für die Erstellung eines lauffähigen Java-Programmes aus Ihrem Quellcode.
- das Java-Runtime-Environment `java.exe` (unter Windows) bzw. `java` (unter Linux) zum Laufenlassen Ihrer compileden Java-Programme.
- Sie finden das aktuelle **JDK 13** (freie Version) unter  
<https://jdk.java.net/>
- Achtung: Ab Version 11 ist nur noch das OpenJDK offen verfügbar. Bei den niedrigeren Versionsnummern können Sie noch den alten Pfad bei Oracle verwenden!

# Java Development Kit (JDK)

## Installation von Java

- Unter Windows: Entpacken Sie die heruntergeladene .zip-Datei in einem Verzeichnis Ihrer Wahl (bei mir z.B. C:\Programme\Java – Achtung: Geht in diesem Ordner nur mit Admin-Rechten!)
- Unter Linux: Entpacken Sie die heruntergeladene tar.gz-Datei in einem Verzeichnis Ihrer Wahl mit  
`tar xzf namederdatei.tar.gz`
- Für beide Betriebssysteme: Fügen Sie den Pfad der ausführbaren Dateien (bei mir unter Windows z. B. C:\ProgramFiles\Java\jdk-13.0.2\bin) Ihrer PATH-Variablen hinzu.

---

# Entwicklungsumgebung Eclipse

Die Entwicklungsumgebung **Eclipse** enthält

- einen intelligenten Editor mit Eingabe-Unterstützung (Code-Completion),
- automatisierte Programm-Compilation während der Eingabe
- komfortable Möglichkeiten, Ihre Programme zu testen, uvm.
- Sie finden die aktuelle Version unter [www.eclipse.org](http://www.eclipse.org)

## Programmstruktur Python

```
1 #!/usr/bin/python
2 # Datei: ProgrammStruktur1.py
3
4 # Eine Variable vereinbaren
5 intWert = 10000
6
7 # Diese Variable ausgeben
8 print "Wert von intWert: ",intWert
```

### Im einfachsten Fall:

- Imperatives Programmier-Paradigma
- Abfolge von Statements
- Diese werden nacheinander abgearbeitet

## Programmstruktur Java

```
1 // Programmdatei: ProgrammStruktur.java
2 public class ProgrammStruktur {
3
4     // main() ist Start-Methode
5     public static void main(String[] args) {
6         // Anlegen einer Variablen
7         int x = 10;
8
9         // Ausgabe der Variablen
10        System.out.println("Wert x=" + x);
11    } // end method main()
12
13 } // end class
```

# Programmstruktur Java

## Programmstruktur Java – Klassen

```
1 // Programdatei: ProgrammStruktur.java
2 public class ProgrammStruktur {
3
4     // ... methoden wie zB main()
5 }
6 // end class
```

■ Objektorientiertes Programmier - Paradigma

■ Programm-Datei = Klasse!

■ Programmdatei muss so heißen wie die Klasse

■ Programmdatei muss auf **.java** enden.

## Programmstruktur Python – Blockbildung durch Einrückung

```
1 # Deklaration einer Funktion
2 def myfunction():
3     alter=input("Wie alt sind Sie? ")
4     print "Sie sind",alter,"Jahre alt!"
5 # Aufruf der Funktion
6 myfunction()
```

- Kopf eines Blockes wird mit : beendet.
- Rumpf eines Blockes wird durch Einrückung kenntlich gemacht.
- Diese Art der Einrückung wird von Python **erzwungen**

## Programmstruktur Java – Blockbildung durch {}-Klammern

```
1 //      ProgrammStruktur.java
2 public class ProgrammStruktur {
3
4     // method main()
5     public static void main(String[] args) {
6         // Anlegen einer Variablen
7         int x = 10;
8
9         // Ausgabe der Variablen
10        System.out.println("Wert x=" + x);
11    } // end method main()
12
13 } // end class
```

---

## Programmstruktur Java – Blockbildung durch {}

- Blöcke werden durch geschweifte Klammern {} begrenzt.
- Einrückung wird von Java **nicht** erzwungen.
- **Aber:** Es gibt Code-Styleguides, die diese Einrückungen vorschreiben.
- → **Nicht Java tritt Ihnen bei Formatierungsfehlern auf die Füße, sondern Ihr Chef!!**

## Programmstruktur Java – main()-Methode

```
1 // Programdatei: ProgrammStruktur.java
2 public class ProgrammStruktur {
3     // main() ist Start-Methode
4     public static void main(String[] args) {
5         // Anlegen einer Variablen
6         int x = 10;
7
8         // Ausgabe der Variablen
9         System.out.println("Wert x=" + x);
10    } // end method main()
11 } // end class
```

# Programmstruktur Java – main()-Methode

Java-Programm muss enthalten:

- **Mindestens eine Klasse mit einer `main()`-Methode**
- Von dieser Methode aus werden **alle** weiteren Statements des Programms aufgerufen.
- Hierbei kann es sich um einfache arithmetische Anweisungen handeln, aber auch ...
- ... um Aufrufe von Methoden aus weiteren Java-Klassen.

# Programmstruktur Java – main()-Methode

## Aufbau der main()-Methode

```
1 public static void main(String[] args) {  
2     // ... weitere Statements  
3 }
```

- **main()** bekommt **immer** ein Array aus Strings übergeben.
- Dieses kann eventuelle Aufrufparameter des Programms enthalten.
- (Später mehr dazu ...)

# Programm-Erzeugung auf der Kommandozeile

## Eine einzelne Quelldatei

### Beispielklasse *Hello.java*

```
1 public class Hello {  
2     public static void main(String [] args) {  
3         System.out.println("Hello World!");  
4     } // end method main()  
5 } // end class Hello
```

Compileraufruf:

javac Hello.java

Erzeugt aus

Hello.java → Hello.class

# Programm-Erzeugung auf der Kommandozeile

## Mehrere Quelldateien

## Mehrere Klassen – jede in einer Quelldatei

- Klasse **MainClass** → Datei MainClass.java
  - Klasse **Eingabe** → Datei Eingabe.java

Compileraufruf: Erzeugt aus

`javac *.java`

# Erzeugt aus

MainClass.java → MainClass.class  
Eingabe.java → Eingabe.class

# Zusammenfassung Programm-Erzeugung und -Start (Konsole)

Merke: Manuelle Erzeugung und Start von Java-Programmen

- Jede Java-Klasse (Datei mit Endung **.java**) muss mit dem Compiler **javac** compiliert werden:
- **javac Dateiname.java**
  - führt einen Syntax-Check der Quelldatei durch.
  - liefert im Erfolgsfall:  
**Dateiname.java → Dateiname.class**
  - liefert im Fehlerfall (bei Syntaxfehlern) eine Fehlermeldung mit Zeilenangabe.
- Start eines Java-Programms mit der Java-Runtime (Java-VM) **java**:
- **java NameDerMainKlasse**  
startet das Programm von der Konsole aus

# Programmerzeugung in der Entwicklungsumgebung

Was sind Entwicklungsumgebungen?

Programme mit einer schönen Oberfläche zum

- Editieren,
- Compilieren und
- Starten von Programmen in **einer einzigen** grafischen Benutzeroberfläche.
- **Keine** mühsamen Befehlseingaben mehr – die Oberfläche tut dies für uns im Hintergrund.

# Programmerzeugung in der Entwicklungsumgebung

## Was sind Entwicklungsumgebungen?

- Häufig auch als **Integrierte Entwicklungsumgebung** bezeichnet:
- Die Schritte
  - Editieren,
  - Compilieren und
  - Starten
- sind in **einer einzigen** Oberfläche **integriert**.
- Integrierte Entwicklungsumgebung = Integrated Development Environment = **IDE**

# Programmerzeugung in der Entwicklungsumgebung

## Die IDE Eclipse

- Eclipse unterstützt viele Programmiersprachen und -paradigmen.
- Jede Programmiersprache wird in einer sog. **Perspektive** dargestellt.
- Perspektive = Eigene Oberfläche mit etwas anderem Look-and-Feel

# Programmerzeugung in der Entwicklungsumgebung

## Die IDE Eclipse

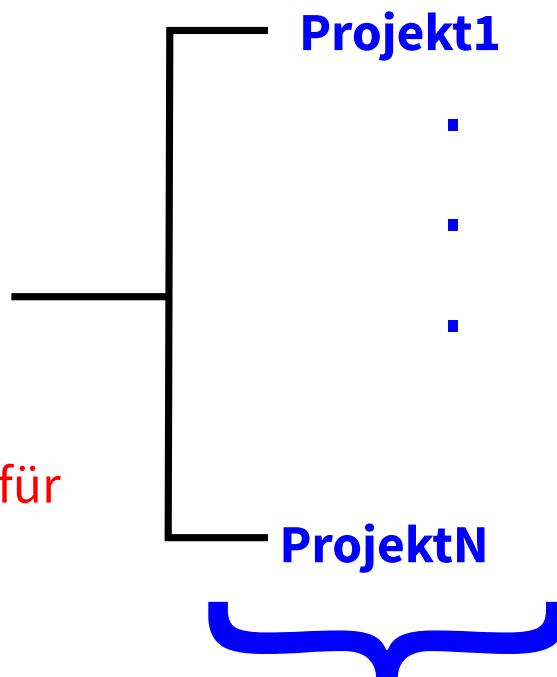
- In der IDE **Eclipse** werden Java-Programme in Form von **Projekten** organisiert.
- Die Projekte werden in einem **Workspace** angelegt. Hierbei handelt es sich einfach um ein Arbeitsverzeichnis, welches der Nutzer beim Start der IDE angibt.
- Ein **Projekt** innerhalb des **Workspaces** ist wiederum in einem eigenen Verzeichnisbaum angelegt.

# Programmerzeugung in der Entwicklungsumgebung

## Workspace und Projekte

C:\erna\myworkspace\

**Workspace** = Arbeitsverzeichnis für  
alle Projekte



Projektverzeichnisse liegen **innerhalb**  
Ihres Workspace. Sie enthalten meist  
alle Quelldateien eines Programms

# Programmerzeugung in der Entwicklungsumgebung

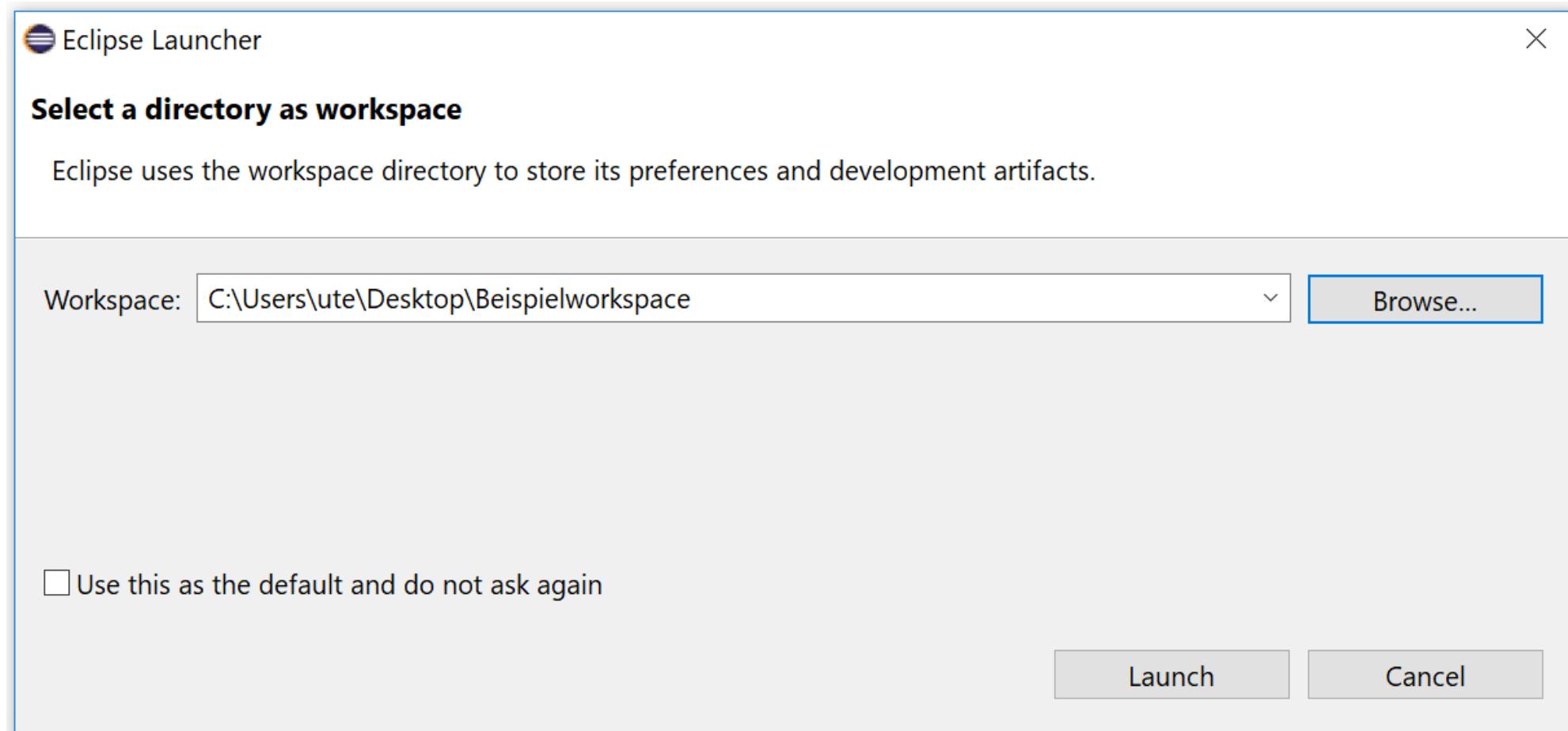
## Start der Entwicklungsumgebung ...



... durch Doppelklick auf's Eclipse-Icon

# Programmerzeugung in der Entwicklungsumgebung

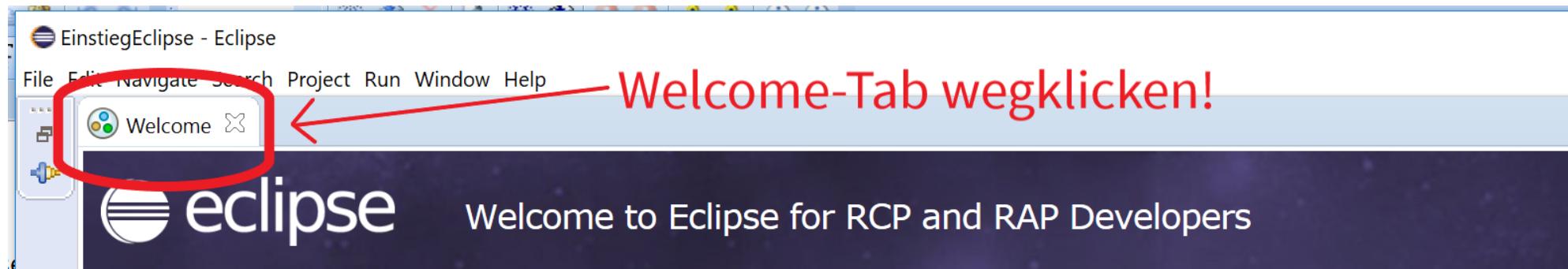
## Eingeben des Verzeichnisses für den Workspace ...



... und weiter mit **Launch**.

# Programmerzeugung in der Entwicklungsumgebung

## Beim ersten Start sehen Sie den Welcome-Tab ...

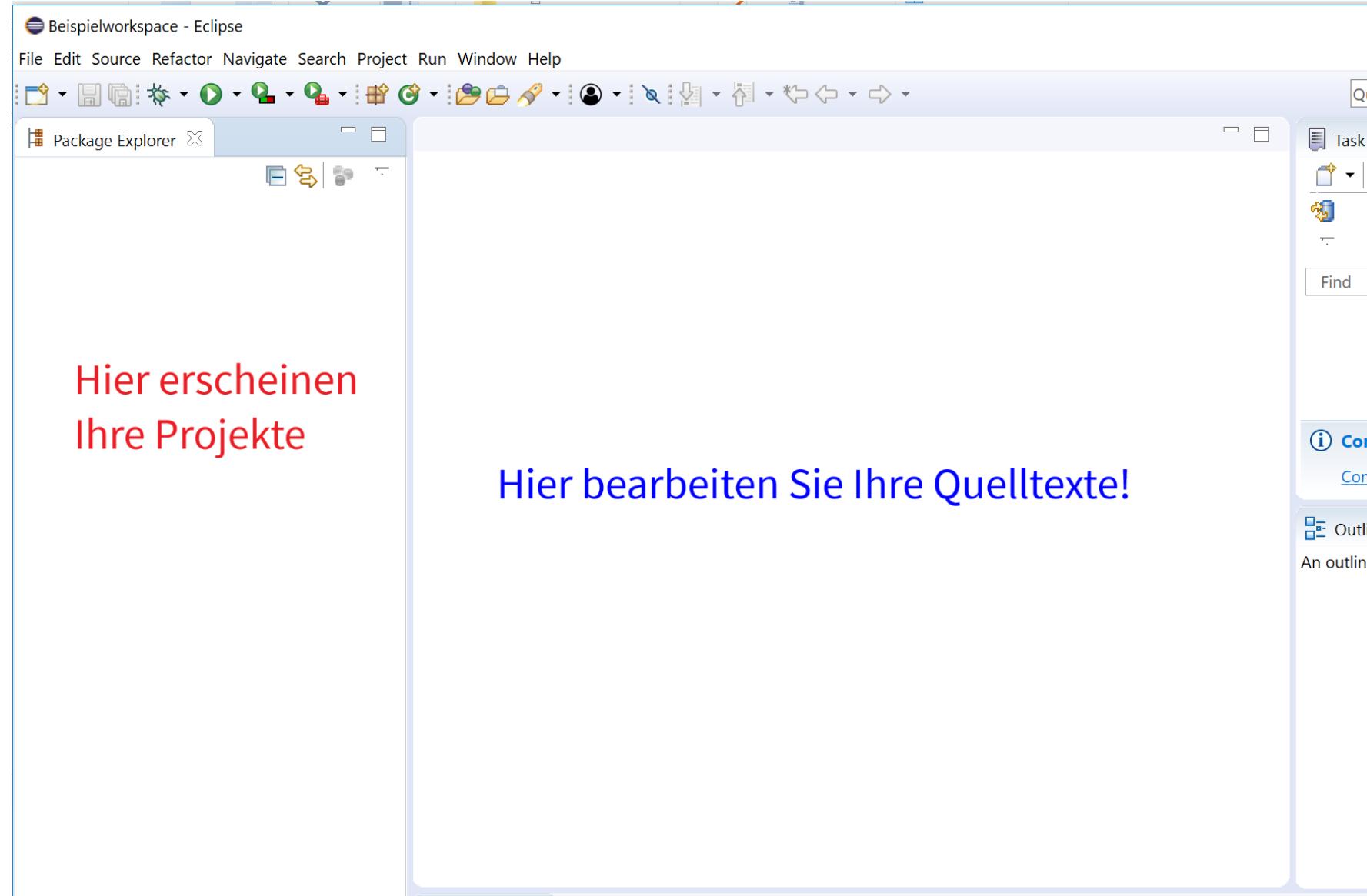


- |   |   |
|---|---|
|  <a href="#">Review IDE configuration settings</a><br>Review the IDE's most fiercely contested preferences   |  <a href="#">Overview</a><br>Get an overview of the features |
|  <a href="#">Tutorial: Create a Rich Client Application</a><br>A guided walk-through for creating an Eclipse RCP application                         |  <a href="#">Tutorials</a><br>Go through tutorials         |
|  <a href="#">Tutorial: Get started with the Remote Application Platform</a><br>A guided walk-through to deploy an Eclipse RCP application to the web |  <a href="#">Samples</a><br>Try out the samples            |

... den Sie weglassen.

# Programmerzeugung in der Entwicklungsumgebung

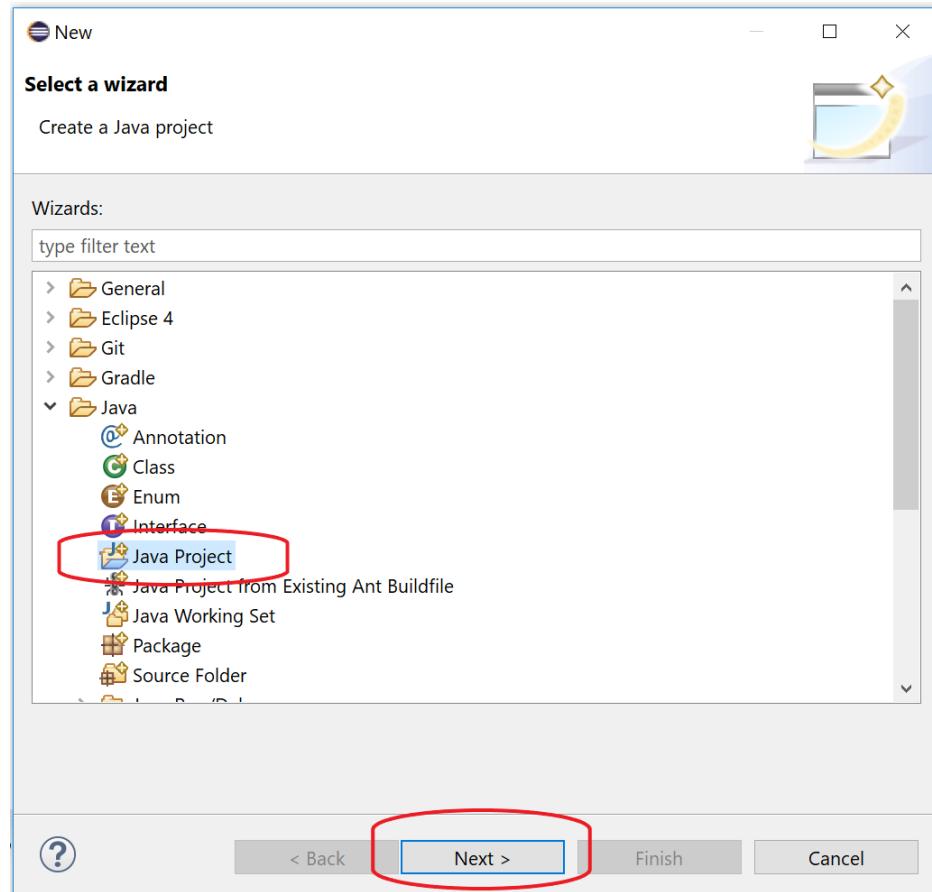
## Danach sehen Sie Ihren noch leeren Workspace ...



# Programmerzeugung in der Entwicklungsumgebung

## Erzeugung eines neuen Projektes mit dem Menüpunkt ...

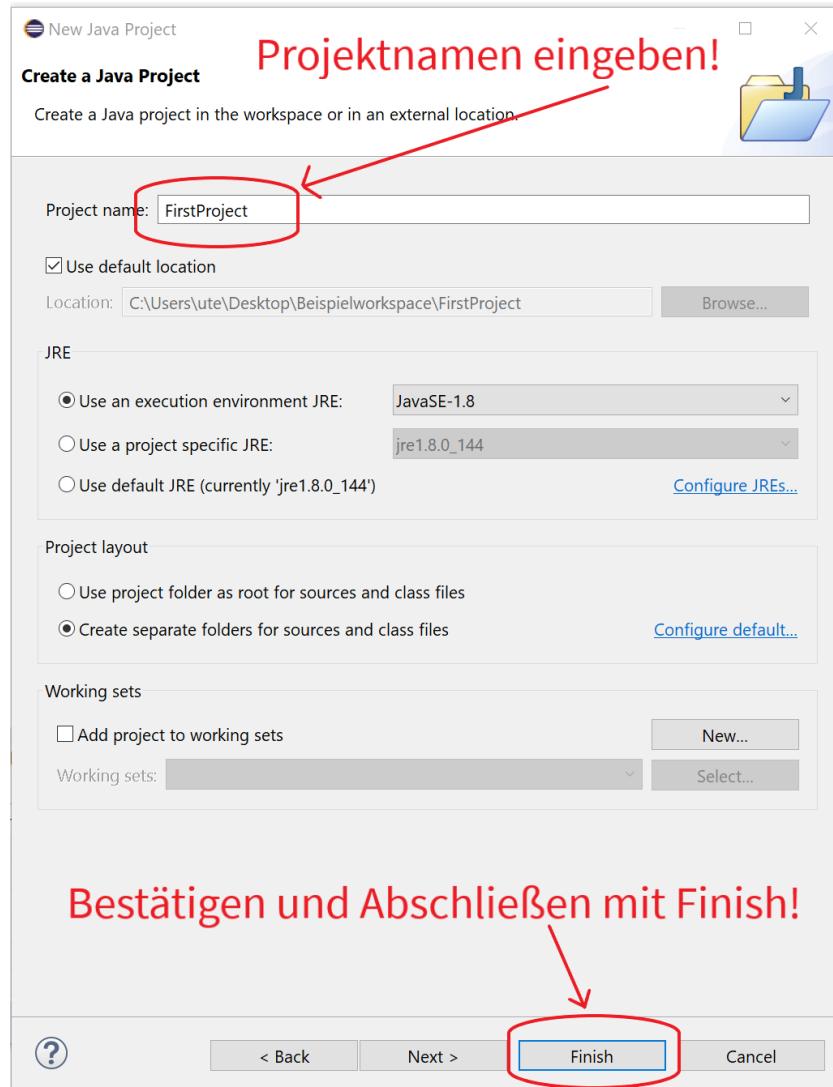
**File → New → Other** liefert folgenden Wizard:



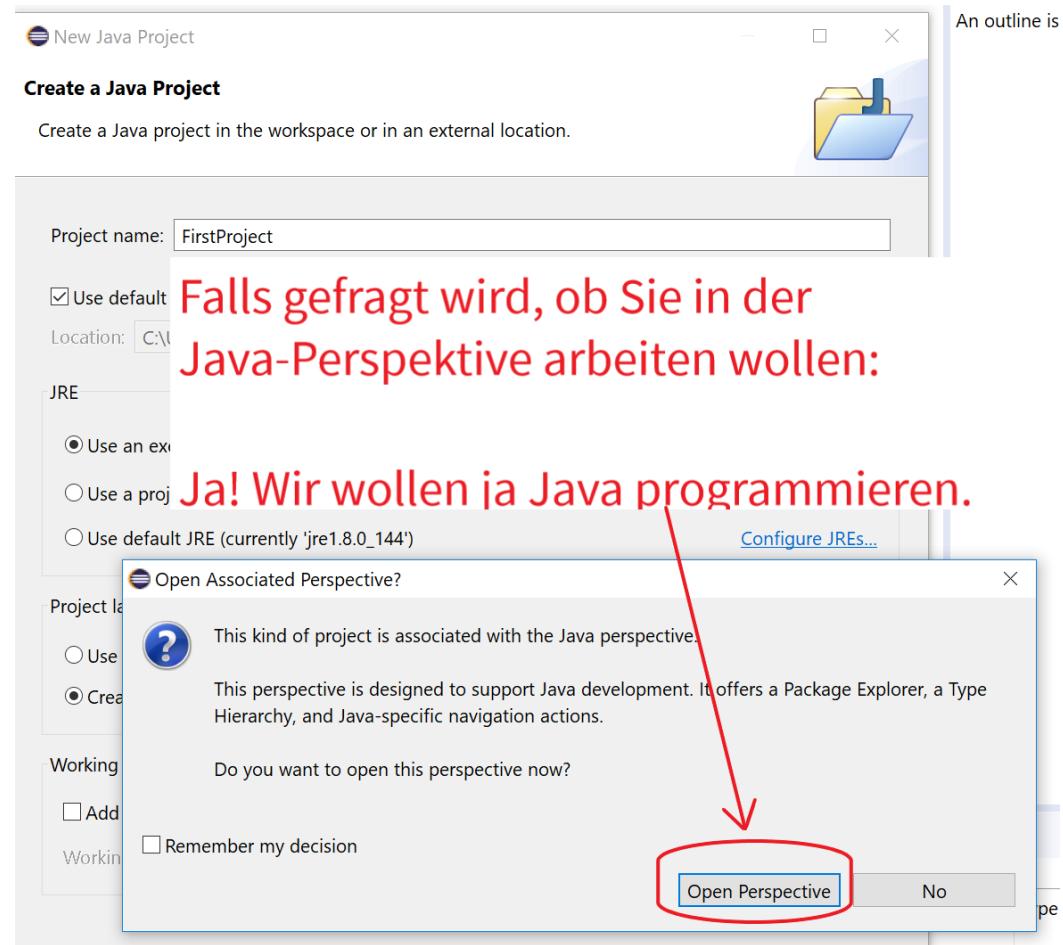
Wählen Sie **Java → Java Project** und gehen Sie weiter mit **Next**.

# Programmerzeugung in der Entwicklungsumgebung

## Vergabe eines Projektnamens ...

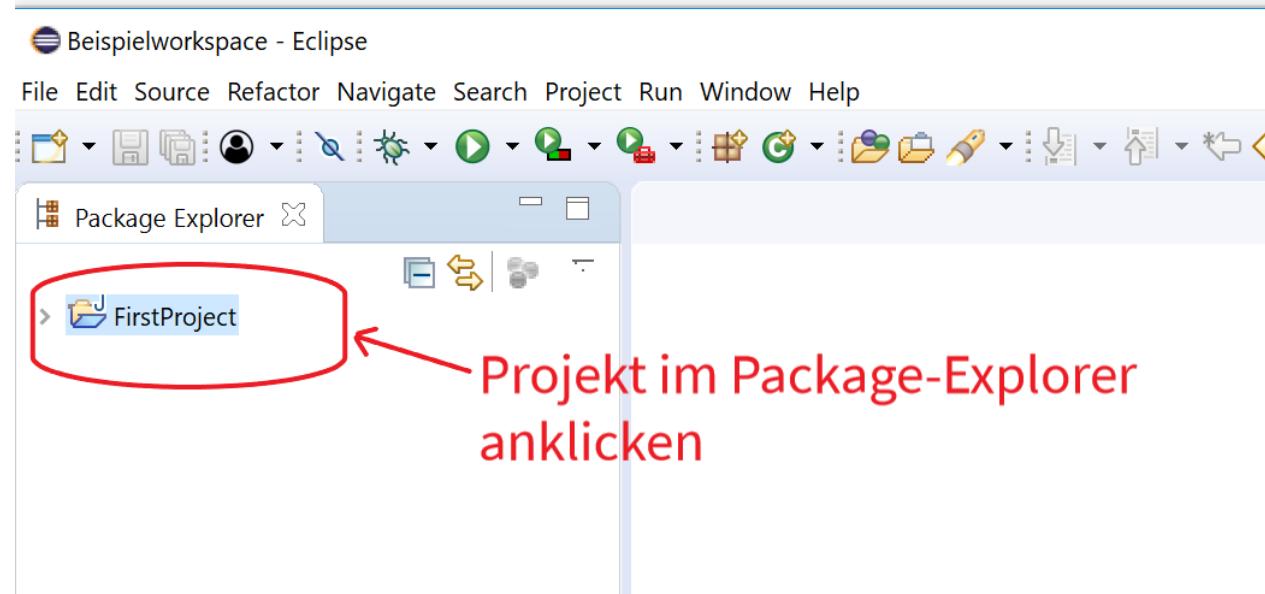


# Programmerzeugung in der Entwicklungsumgebung Java-Perspektive bestätigen, falls gefragt wird ...



# Programmerzeugung in der Entwicklungsumgebung

## Eine Java-Klasse innerhalb des Projektes anlegen ...

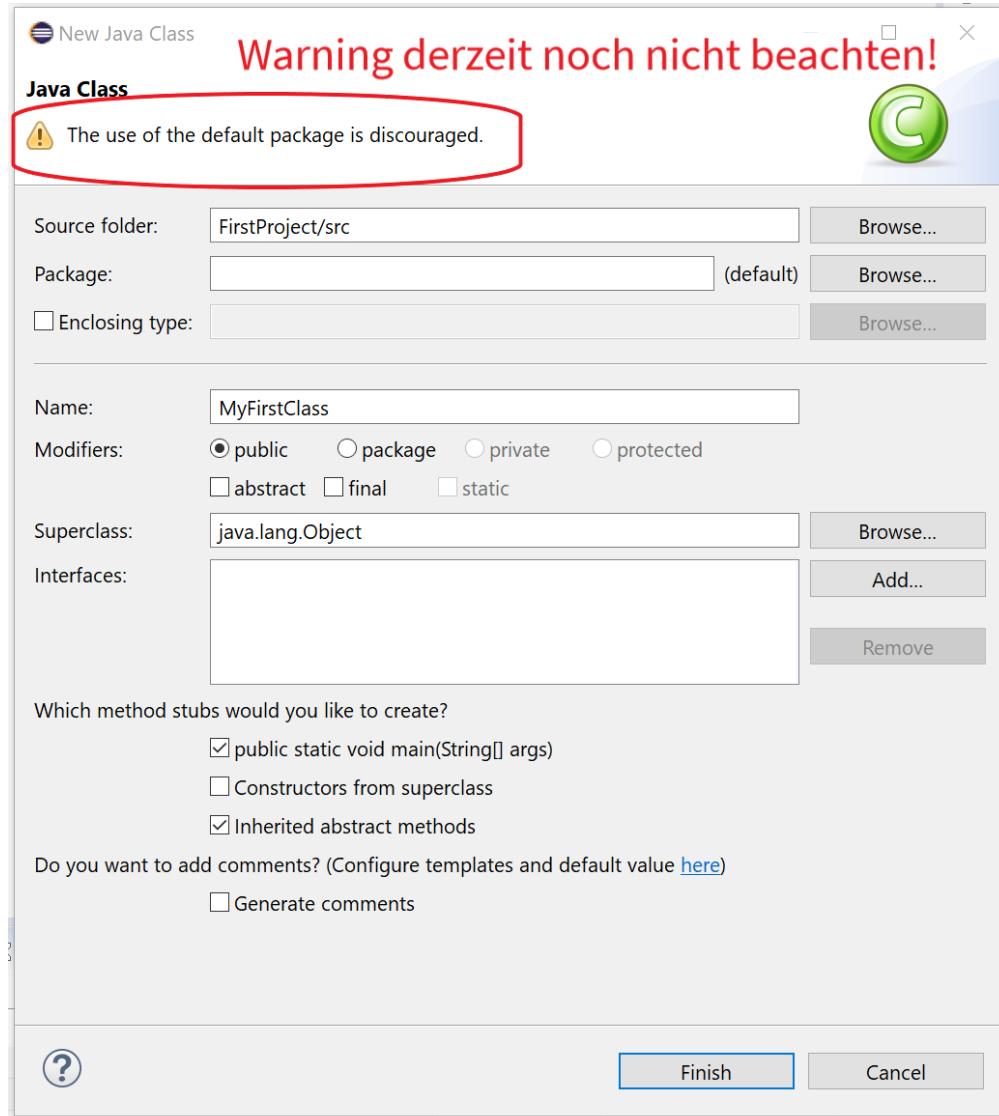


... und nach Anwählen des Projektes über den Menüpunkt **File → New → Other**

... und danach **Java → Class** den Wizard zur Erstellung einer Klasse wählen.

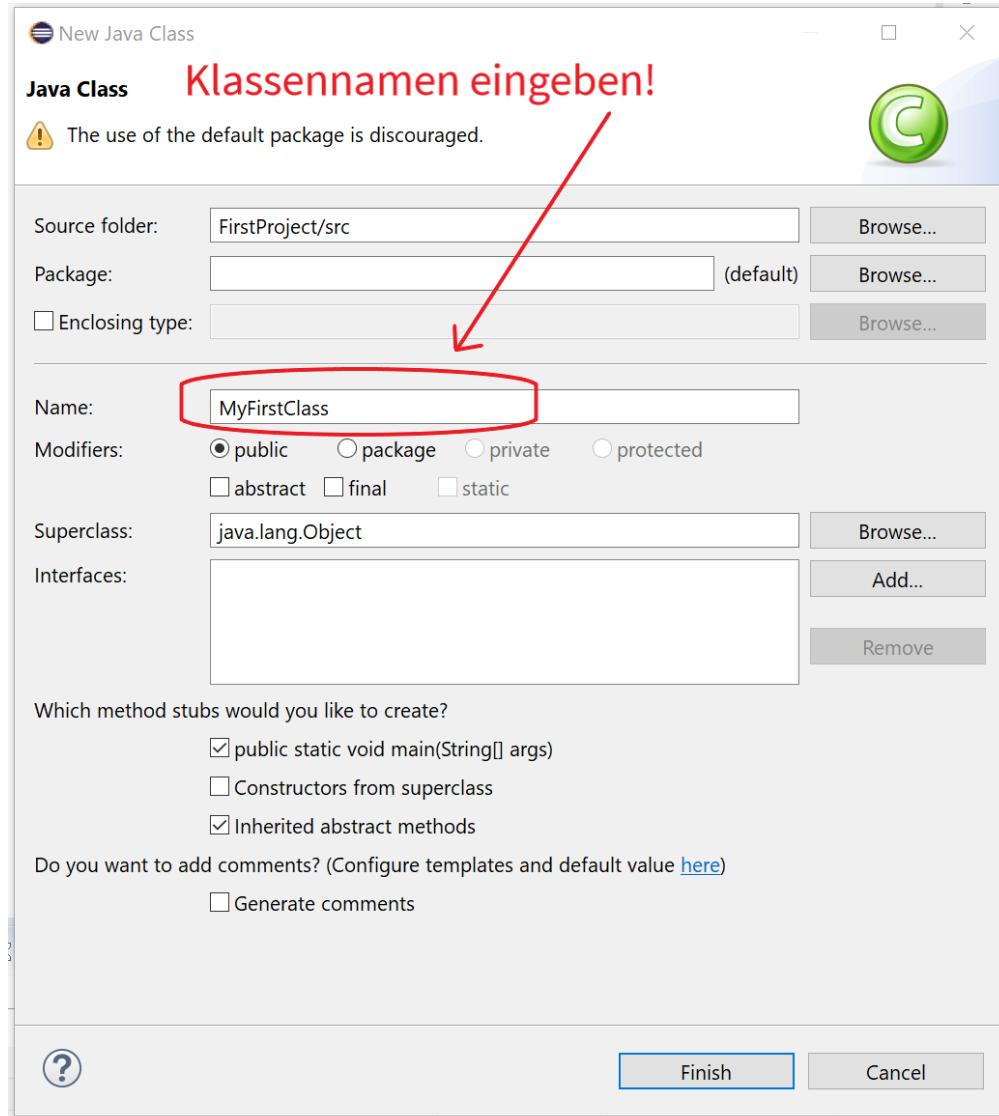
# Programmerzeugung in der Entwicklungsumgebung

## Eine Java-Klasse innerhalb des Projektes anlegen ...



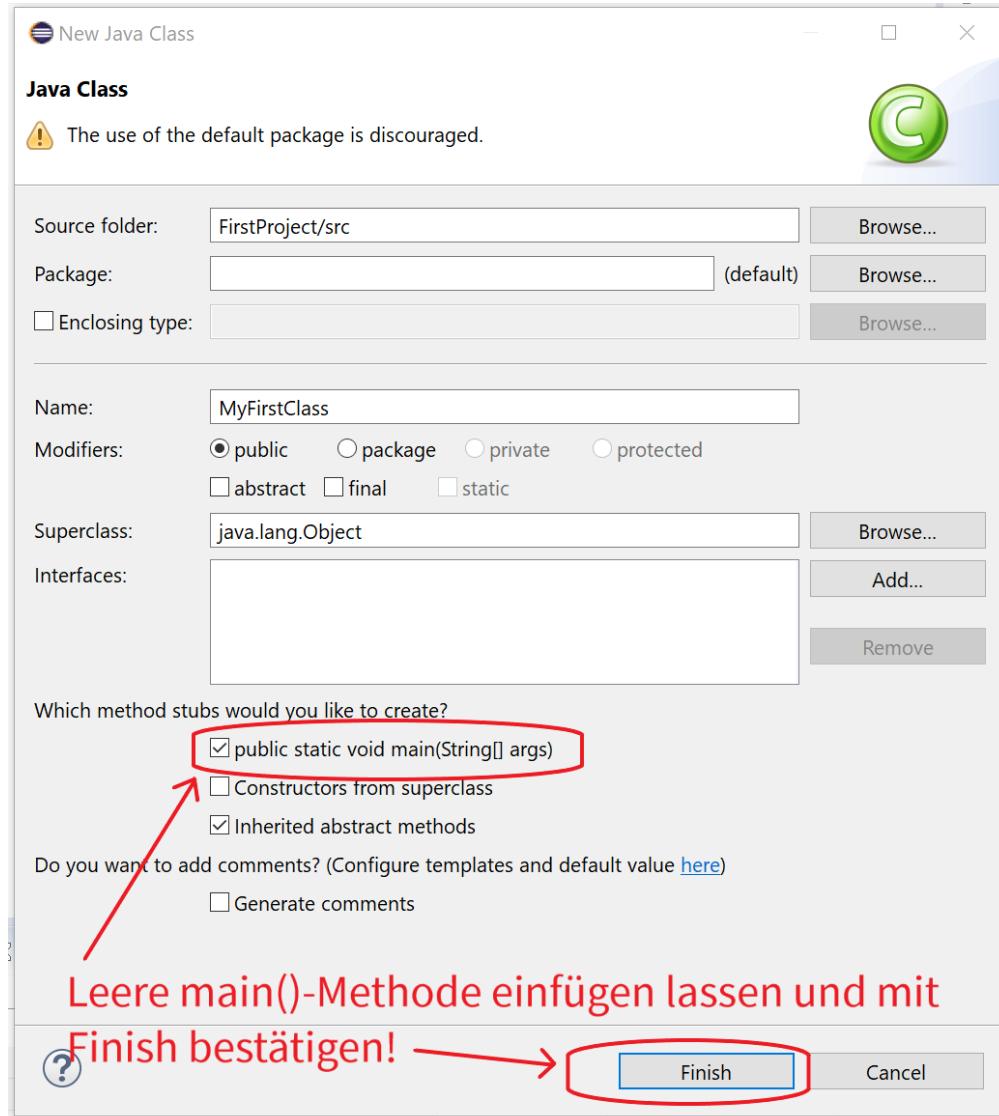
# Programmerzeugung in der Entwicklungsumgebung

## Eine Java-Klasse innerhalb des Projektes anlegen ...



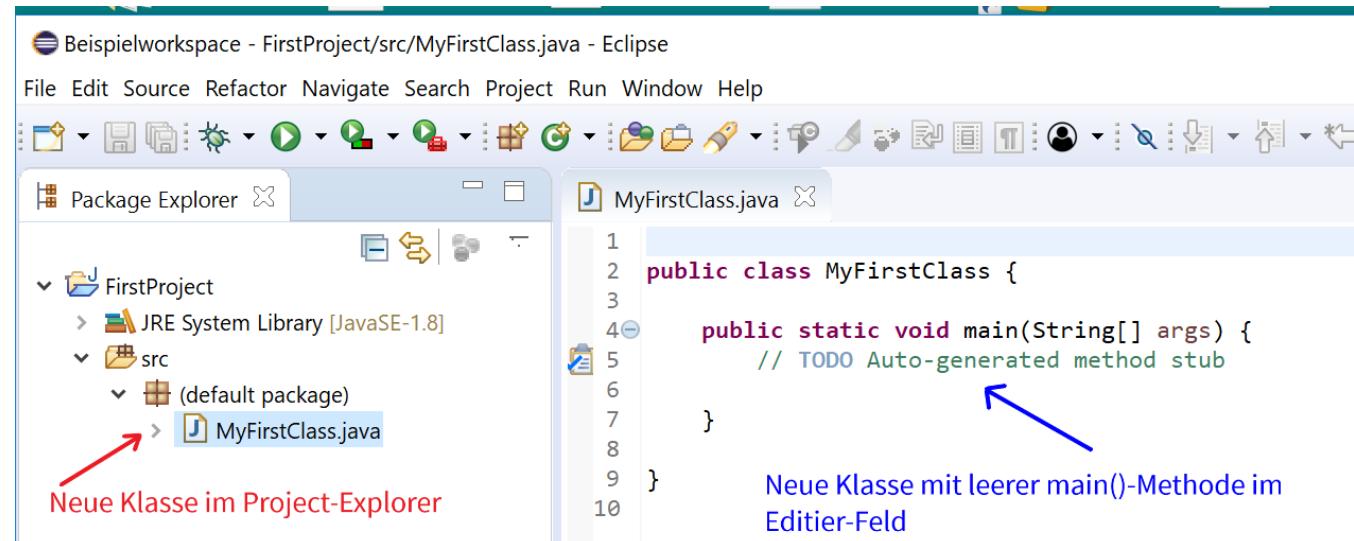
# Programmerzeugung in der Entwicklungsumgebung

## Eine Java-Klasse innerhalb des Projektes anlegen ...



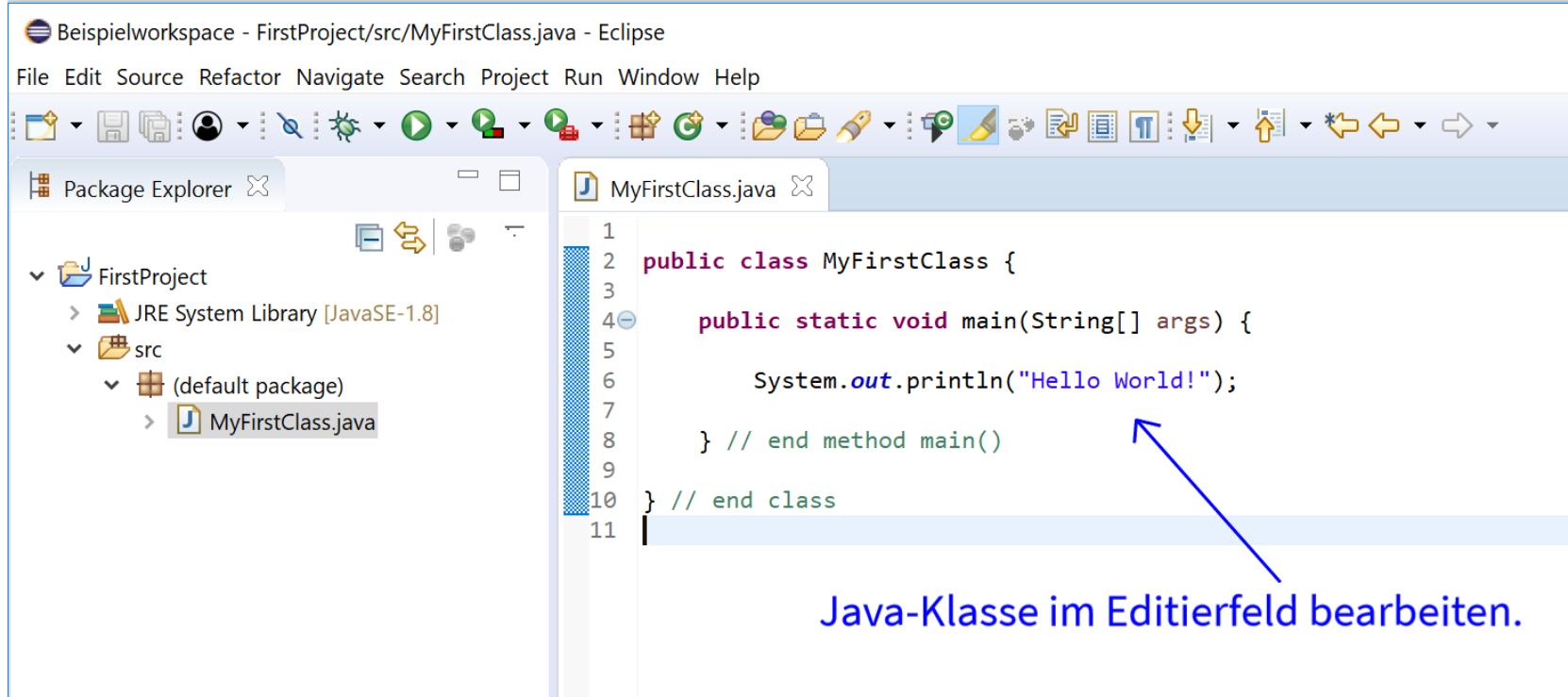
# Programmerzeugung in der Entwicklungsumgebung

## Java-Klasse erscheint im Project-Explorer und im Editier-Feld ...



# Programmerzeugung in der Entwicklungsumgebung

## Java-Klasse bearbeiten und abspeichern ...



The screenshot shows the Eclipse IDE interface. The title bar reads "Beispielworkspace - FirstProject/src/MyFirstClass.java - Eclipse". The menu bar includes File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help. The toolbar has various icons for file operations. The left sidebar is the "Package Explorer" showing a project named "FirstProject" with a "src" folder containing a "(default package)" folder and a file "MyFirstClass.java". The main area is the "MyFirstClass.java" editor window, displaying the following Java code:

```
1 public class MyFirstClass {  
2     public static void main(String[] args) {  
3         System.out.println("Hello World!");  
4     } // end method main()  
5 } // end class
```

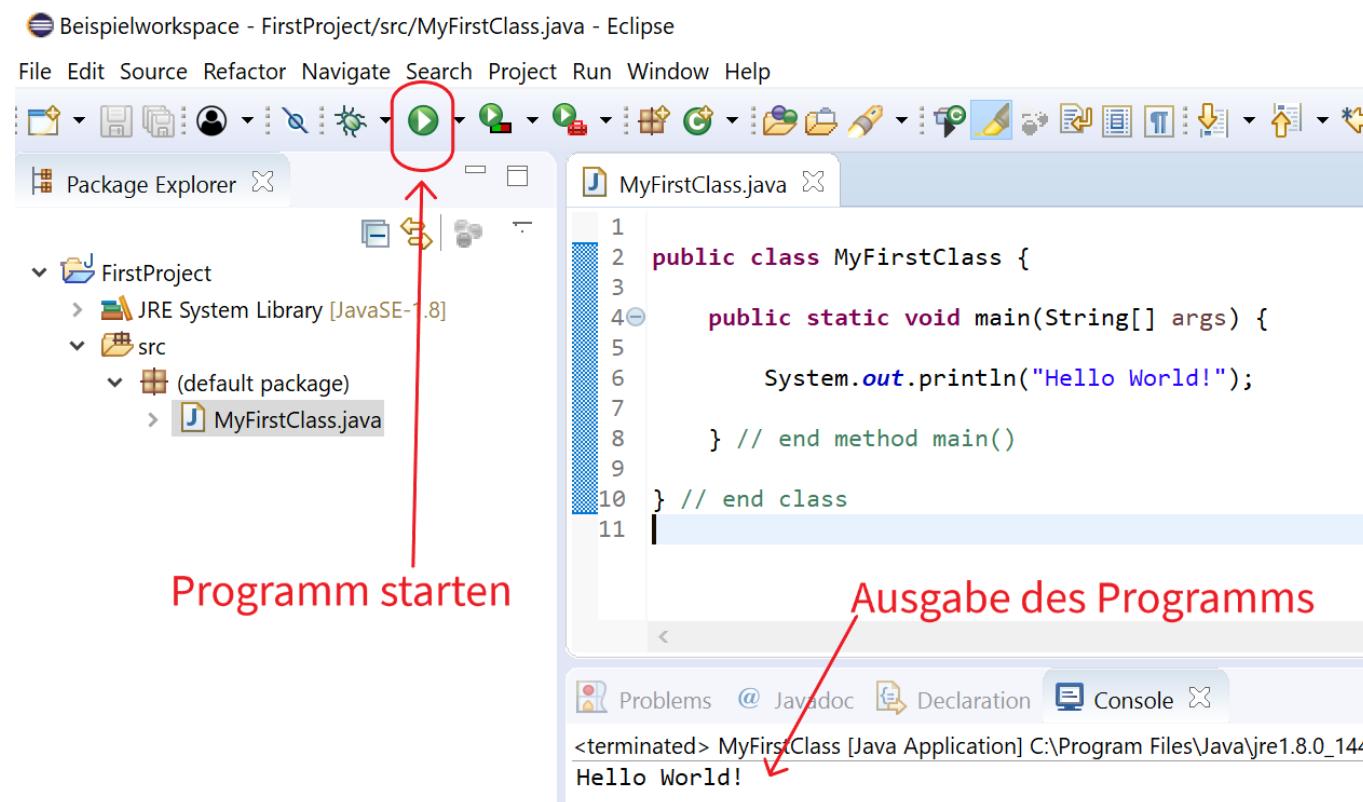
A blue arrow points from the text "Java-Klasse im Editierfeld bearbeiten." to the word "main" in the code editor.

Java-Klasse im Editierfeld bearbeiten.

→ Beim Abspeichern wird der Quelltext bereits automatisch compiliert!

# Programmerzeugung in der Entwicklungsumgebung

## Programm starten ...



→ Projekt oder main()-Klasse muss angewählt sein!

→ Innerhalb des Projektes sucht die IDE automatisch nach der Klasse mit der main()-Methode!

# Zusammenfassung Programm-Erzeugung und -Start (Eclipse-IDE)

Merke: Erzeugung und Start von Java-Programmen in Eclipse

- Java-Programme sind in Eclipse in **Projekten** organisiert.
- Beim Abspeichern jeder Java-Klasse innerhalb eines Java-Projektes wird diese **automatisch compiliert**.
- **Syntaxfehler** werden direkt beim Editieren im Editor gezeigt. Erst nach Entfernung dieser Fehler wird beim Abspeichern wieder compiliert.
- Der **Start** eines Java-Programms innerhalb von Eclipse erfolgt mit dem grünen Start-Pfeil 

# Die Java-API

## Schauen Sie in die API!

### ■ Damit ist gemeint:

- Schauen Sie in der Dokumentation von Java nach einer bestimmten Java- Klasse / Java-Methode.
- Nicht alles muss von Hand selbst programmiert werden.
- Für viele Problemstellungen gibt es Klassen und Algorithmen, die von den Entwicklern von Java „fix und fertig“ zur Verfügung gestellt werden.

### ■ Einstiegslink für viele nützliche Java-Klassen:

<https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/package-summary.html>

# Die Java-API

Was bedeutet das Akronym „API“?

- Application Programming Interface
- Deutsch (genaue Übersetzung): Anwendungs-Programmierschnittstelle.
- Meist abgekürzt: Programmierschnittstelle.
- Vorgefertigte Klassen/Funktionen.
- Diese können von Entwicklern/Entwicklerinnen verwendet werden, um Anwendungsprogramme zu schreiben.

# Die Java-API

## Was bedeutet das Akronym „API“?

- Etwas allgemeiner:
- Vorgefertigte Programmpakete (Bibliotheken), die von Entwicklern(Entwicklerinnen) benutzt werden,
  - um eigene Programme zu schreiben, ohne jeden jemals erfundenen Algorithmus noch mal neu entwickeln zu müssen.
  - um eigene Programmteile/Programmpakete an ein bestehendes System anzubinden.
- Diese vorgefertigten Bibliotheken sind häufig standardisiert und haben eine ausführliche HTML-Dokumentation.

## Die Java-API

Und was bedeutet nun „Schauen Sie in die API“??

- In unserem Fall: Schauen Sie in die offizielle HTML-Doku von Java nach bestimmten Klassen oder Funktionen!
- Auch für selbstdefinierte Java-Klassen kann eine solche Doku mit dem Programm `javadoc` erzeugt werden!
- Für Sprachen wie C und C++ gibt es ähnliche Doku-Generatoren (z.B. `doxygen`)

# Variablen und Datentypen

## Variablenvereinbarung allgemein

Generell können Variablen bei ihrer Vereinbarung

- **implizit** oder

- **explizit**

typisiert werden.

Bei der **impliziten** Typisierung wird der Datentyp einer Variablen durch die Wertzuweisung festgelegt (vgl. Python).

Bei der **expliziten** Typisierung wird der Datentyp einer Variablen direkt bei der Vereinbarung angegeben – Java!

# Variablenvereinbarung in Java

Eine Variablenvereinbarung erfolgt in Java immer **explizit**:

Explizite Variablenvereinbarung in Java

*datentyp variablename ;*

# Grobaufteilung von Datentypen

In Java unterscheiden wir

- **Elementare (Primitive)** Datentypen (beispielsweise Ganzzahl- oder Gleitkommazahlen)
- **Aggregierte (Zusammengesetzte)** Datentypen (z.B. Klassen, Arrays, Aufzählungen, etc.). Sie werden auch als **Referenzdatentypen** bezeichnet, da ihre Variablen immer eine Referenz (Speicheradresse) auf das eigentliche Datenelement enthalten.
- Variablen, deren Datentyp ein Referenzdatentyp ist, werden auch als **Referenzvariablen** bezeichnet.

# Elementare Datentypen in Java

Datentyp	Beschreibung	Vereinbarung mit Literal
byte	<ul style="list-style-type: none"><li>■ 8 Bit-Ganzzahl</li><li>■ Wertebereich -128 ... 127</li><li>■ Negative Zahlen im Zweierkomplement</li></ul>	<i>byte b=2;</i>
short	<ul style="list-style-type: none"><li>■ 16 Bit-Ganzzahl</li><li>■ Wertebereich -32768 ... 32767</li><li>■ Negative Zahlen im Zweierkomplement</li></ul>	<i>short b=45;</i>

# Elementare Datentypen in Java

Datentyp	Beschreibung	Vereinbarung mit Literal
int	<ul style="list-style-type: none"><li>■ 32 Bit-Ganzzahl</li><li>■ Wertebereich -2147483648 ... 2147483647</li><li>■ Negative Zahlen im Zweierkomplement</li></ul>	<i>int b=5;</i>
long	<ul style="list-style-type: none"><li>■ 64 Bit-Ganzzahl</li><li>■ Wertebereich <math>-2^{63}</math> ... <math>2^{63} - 1</math></li><li>■ Ab Java 8 sind auch vorzeichenlose Ganzzahlen von <math>0 \dots 2^{64} - 1</math> möglich.</li><li>■ Negative Zahlen im Zweierkomplement</li></ul>	<i>long b=1000L;</i>

# Elementare Datentypen in Java

Datentyp	Beschreibung	Vereinbarung mit Literal
char	<ul style="list-style-type: none"><li>■ Länge 16 Bit (Unicode)</li><li>■ Wertebereich ganzzahlig 0 ... 65535</li><li>■ Das Zeichen <b>A</b> hat hier beispielsweise – wie in der ASCII-Codierung den Wert 65.</li></ul>	<code>char x='A';</code>
boolean	<ul style="list-style-type: none"><li>■ Wahrheitswert</li><li>■ Kann die Werte true oder false annehmen</li></ul>	<code>boolean b=true;</code>

# Elementare Datentypen in Java

Datentyp	Beschreibung	Vereinbarung mit Literal
float	<ul style="list-style-type: none"><li>■ 32 Bit-Gleitkommazahl (IEEE 754)</li><li>■ Wertebereich <math>\pm 1.4 \cdot 10^{-45} \dots \pm 3.4 \cdot 10^{+38}</math></li></ul>	<i>float</i> b=48.7f;
double	<ul style="list-style-type: none"><li>■ 64 Bit-Gleitkommazahl (IEEE 754)</li><li>■ Wertebereich <math>\pm 4.9 \cdot 10^{-324} \dots \pm 1.7 \cdot 10^{+308}</math></li></ul>	<i>double</i> b=55.7;

# Konstanten (konstante Variablen) in Java

## Modifier `final` für konstante Variablen

- wird vor die Variablenvereinbarung geschrieben.
- Wirkung: Nach der ersten Wertzuweisung kann kein neuer Wert mehr an diese Variable zugewiesen werden.

## Beispiel für eine `final` -Variable

### Beispielquelltext

```
1 final double PI=3.141592653589793;  
2 System.out.println("PI: " + PI);  
3  
4 // Versuch: Änderung von PI  
5 PI=9.0;
```

Modifier `final`: Variable kann nach ~~der ersten Zuweisung~~ nicht mehr geändert werden! Compilerfehler!

# Aggregierte Datentypen in Java

Hier sind unter anderem zu nennen

- Arrays
- Klassen (kommt später – in diesem Kapitel nur ein kurzes Beispiel)
- Eine besondere Klasse ist hier die Klasse **String**
- Interfaces (kommt später)
- Collections (z.B. Listen) (kommt später), und einige mehr ...

# Aggregierte Datentypen in Java

Aggregierte Datentypen in Java = Referenzdatentypen!

- Eine Variable eines aggregierten Datentyps enthält in Java nicht die tatsächlichen Werte des Datentyps, sondern nur eine **Referenz** auf die Speicherstelle, an der die echten Werte stehen.
- Eine Referenz ist eine speziell codierter Wert der **Speicheradresse** der tatsächlichen Werte.

# Aggregierte Datentypen in Java – einfache Arrays

## Arrays

- Ein Array ist eine Datenstruktur, die in der Lage ist, eine feste Anzahl von Variablen gleichen Typs aufzunehmen.
- Die Größe eines Arrays bleibt, nachdem es erzeugt wurde, gleich!

# Aggregierte Datentypen in Java – einfache Arrays

## Vereinbarung von Array-Variablen

// Möglichkeit 1:

```
datentyp [] array1 = new datentyp [anzahl];
```

// Möglichkeit 2:

```
datentyp [] array2 = {element1, element2, ... , elementN};
```

**Die beiden Variablen array1 und array2 enthalten jeweils nur die Referenz auf die Speicheradresse, an der die Werte des Arrays beginnen!**

# Aggregierte Datentypen in Java – einfache Arrays

## Beispiel für ein Array vom Typ int – Möglichkeit 1

```
1 // Vereinbarung eines int-Arrays mit Platz fuer 4 Elemente
2 int [] myArray = new int [4];
3 // Belegung der Array-Elemente mit Werten
4 myArray[0] = 7;
5 myArray[1] = -99;
6 myArray[2] = 102;
7 myArray[3] = 8;
```

0	1	2	3
7	-99	102	8

# Aggregierte Datentypen in Java – einfache Arrays

## Beispiel für ein Array vom Typ int – Möglichkeit 2

```
1 // Vereinbarung eines int-Arrays mit Platz fuer 4 Elemente
2 int [] myArray = {7,-99,102,8};
```

0	1	2	3
7	-99	102	8

# Aggregierte Datentypen in Java – einfache Arrays

Laenge eines Arrays feststellen: `length`-Variable

```
1 int [] myArray = {7,-99,102,8};  
2 System.out.println(  
3     "Laenge des Arrays: "+myArray.length);
```

Array hat 4 Elemente → Ausgabeanweisung gibt aus:

Laenge des Arrays: 4

# Aggregierte Datentypen in Java – mehrdimensionale Arrays

Vereinbarung mehrdimensionaler Arrays: Allgemeine Form

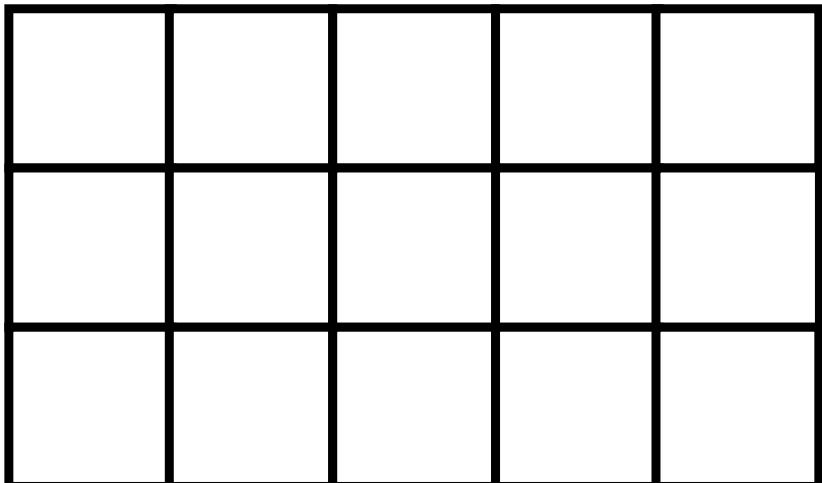
```
1 typ [] [] ... arrname = new typ [zeilen] [spalten] [...] ... ;
```

Für jede Dimension wird bei **Vereinbarung** und **Erzeugung** ein **[]**-Klammerpaar verwendet!

# Aggregierte Datentypen in Java – mehrdimensionale Arrays

## Vereinbarung mehrdimensionaler Arrays: Eine Matrix

```
1 int rows=3;  
2 int columns=5;  
3 int [][] matrix1 = new int [rows] [columns];
```



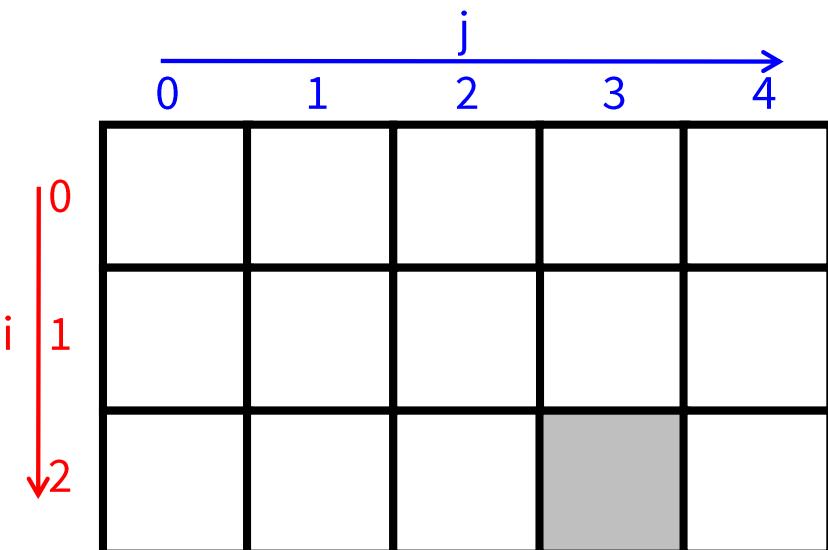
**Beispiel:**

rows = 3  
columns = 5

# Aggregierte Datentypen in Java – mehrdimensionale Arrays

## Zugriff auf mehrdimensionale Arrays

```
1 int i=2;  
2 int j=3;  
3 matrix1[i][j]=255;  
4 System.out.println("Position [2][3]: "+matrix1[i][j]);
```



**Indizierung:**

*i* = 2

*j* = 3

# Aggregierte Datentypen in Java – Strings

## Zeichenketten – Strings

- Zeichenfolgen / Zeichenketten werden in Java meist über die Klasse **String** vereinbart.
- Hierbei handelt es sich um eine Standard-Java-Klasse, die jedoch etwas einfacher funktioniert, als andere Klassen. Sie wird im Kapitel **Die Klasse String** ausführlich behandelt.
- In einigen Fällen wird auch die Klasse **StringBuffer** verwendet.
- Falls diese in späteren Phasen der Lehrveranstaltung von Ihnen benötigt wird, so werden Ihnen Ihre Dozenten die Handhabung zeigen.

# Aggregierte Datentypen in Java – Strings

## Einführungsbeispiel zur Vereinbarung von Strings

```
// Vereinbarung einer Variablen  
// vom Typ String  
String s1 = "Erna";
```

Vereinbarung und Initialisierung  
einer Variablen vom Typ String.

```
// Ausgabe eines mit + verketteten  
// Strings  
System.out.println("Hallo " + s1);
```

Ein konstanter  
String (Literal)

Verkettungsoperator

Unser vorher vereinbarter String

# Aggregierte Datentypen in Java – Klassen

## Vereinbarung von Instanzen einer Klasse

```
// Wenn der Instanz bei der Erzeugung
// keine Parameter uebergeben werden ...
KlassenName nameRefVariable = new KlassenName();

// Erzeugung einer Instanz mit Parameteruebergabe
KlassenName nameRefVariable = new KlassenName(parameter ...);
```

# Aggregierte Datentypen in Java – Klassen

Einfaches Beispiel einer Klasse: HelloMessages/Teil 1

```
1 public class HelloMessages {  
2  
3     String nachricht; // eine Objektvariable  
4  
5     // Standard-Konstruktor (immer parameterlos)  
6     HelloMessages(){  
7         nachricht="Moin";  
8     }  
9  
10    // voll qualifizierter Konstruktor: setzt Instanzvariablen  
11    HelloMessages(String nachricht) {  
12        this.nachricht = nachricht;  
13    } // ... weiter naechste Seite
```

# Aggregierte Datentypen in Java – Klassen

## Einfaches Beispiel einer Klasse: HelloMessages/Teil 2

```
14 // ... Fortsetzung von der letzten Seite
15 // einfache Methode
16 void hello() {
17     System.out.println(nachricht);
18 } // end method
19
20 } // end class
```

# Aggregierte Datentypen in Java – Klassen

## Einfaches Beispiel einer Klasse: HelloMain/Teil 1

```
1 public class HelloMain {  
2  
3     public static void main(String[] args) {  
4  
5         // Variable vom Referenztyp HelloMessages  
6         // Erzeugung Arbeitsobjekt mit Standardkonstruktor  
7         HelloMessages worker1 = new HelloMessages();  
8  
9         // Variable vom Referenztyp HelloMessages  
10        // Erzeugung Arbeitsobjekt mit voll qual. Konstruktor  
11        HelloMessages worker2 = new HelloMessages("Guten Tag!");  
12        // ... Fortsetzung auf der naechsten Seite
```

# Aggregierte Datentypen in Java – Klassen

## Einfaches Beispiel einer Klasse: HelloMain/Teil 2

```
13 // ... Fortsetzung von der vorherigen Seite
14 // Methodenaufruf bei den Arbeitsobjekten
15 worker1.hello(); // Gibt Standardnachricht aus
16 worker2.hello(); // Gibt uebergebene Nachricht aus
17
18 } // end method main()
19 } // end class
```

Ausgabe:

moin

Guten Tag!

# Aggregierte Datentypen in Java – null-Referenz

Das Literal `null` bedeutet:

- Variable ist noch nicht mit einer gültigen Adresse initialisiert!
- Wird immer dort zur Initialisierung verwendet, wo erst später entschieden wird, was genau zugewiesen wird.
- `null` ist also ein „Default-Wert“ für einen Referenz-Wert.

# Aggregierte Datentypen in Java – null-Referenz

## Beispiel zur null-Referenz: Versuch 1

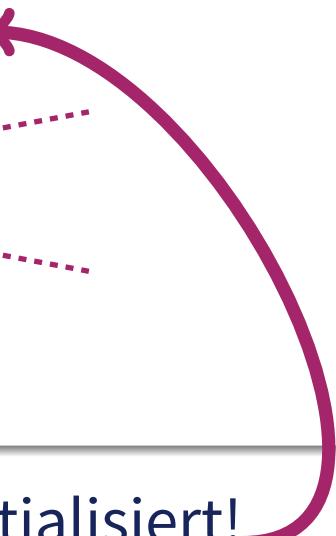
```
1 public class NullReferenzVersuch1 {  
2  
3     public static void main(String[] args) {  
4         int [] array;  
5         array[0] = 5; // Compilerfehler  
6     } // end method  
7 } // end class
```

- nicht initialisiert!
- Zuweisung funktioniert nicht – Compilerfehler!

# Aggregierte Datentypen in Java – null-Referenz

## Beispiel zur null-Referenz: Versuch 2

```
1 public class NullReferenzVersuch2 {  
2  
3     public static void main(String[] args) {  
4         int [] array = null; // Reference initialized to null  
5         array[0] = 5; // Accessing elements of null array  
6     } // end method  
7 } // end class
```



- Referenz ist jetzt mit null initialisiert!
- Compiler sagt zwar: OK!
- Es entsteht jedoch ein Laufzeitfehler beim Start des Programms, da der Variablen immer noch keine gültige Speicheradresse zugewiesen wurde.

# Aggregierte Datentypen in Java – null-Referenz

## Beispiel zur null-Referenz: Versuch 3 / Teil 1

```
1 import java.util.Scanner;  
2  
3 public class NullReferenzVersuch3 {  
4  
5     public static void main(String[] args) {  
6         int [] array = null; // Line 6  
7         Scanner eingabe = new Scanner(System.in);  
8  
9         System.out.print("Anzahl Elemente? ");  
10        int anzahl = eingabe.nextInt();
```

- Referenz ist jetzt mit null initialisiert!
- Rest auf der nächsten Folie ...

# Aggregierte Datentypen in Java – null-Referenz

## Beispiel zur null-Referenz: Versuch 3 / Teil 1

```
11 // Achtung: anzahl <= 0 wurde nicht abgefangen!
12 array = new int [anzahl];
13
14 if (array != null) {
15     array [0] = -99;
16 } // end if
```

→Referenzvariable array wird auf null getestet, bevor der Speicherplatz an Position 0 belegt wird!

## Zusammenfassung zur null-Referenz

Die null-Referenz wird verwendet ...

- ... um Referenzvariablen (z.B. Arrays, Referenzen auf Objekte einer Klasse, etc.) zu initialisieren, sofern die echte Initialisierung später stattfinden soll.
  
- null wird auch häufig von Methoden zurückgegeben, deren Rückgabetyp ein Referenzdatentyp ist: Dann, wenn ihnen eine Erzeugung der Referenzvariablen mit new fehlgeschlagen ist. Dazu später mehr!

# Bereits bekannte arithmetische Operatoren aus Python ... ... sehen in Java genau gleich aus!

Python-Operator	Java-Operator	Erläuterung
+	+	Addition
-	-	Subtraktion
*	*	Multiplikation
/	/	Division
%	%	Division mit Rest
+ - * / %  += -= *= /= %= 	+ - * / %  += -= *= /= %= 	Zuweisungsoperatoren für arithmetische Operationen

# Neu eingeführte arithmetische Operatoren in Java

## Inkrement- und Dekrement-Operatoren

Operator	Erläuterung
<code>++</code>	Inkrement-Operator: Zählt eine Variable um 1 hoch
<code>--</code>	Dekrement-Operator: Zählt eine Variable um 1 herunter

# Neu eingeführte arithmetische Operatoren in Java

## Inkrement- und Dekrement-Operatoren

### Grundform Inkrement/Dekrement

```
1 int i=22;  
2 int j=100;  
3  
4 i++; // i hat nach dieser Anweisung den Wert 23  
5 j--; // j hat nach dieser Anweisung den Wert 99
```

# Neu eingeführte arithmetische Operatoren in Java

## Inkrement- und Dekrement-Operatoren

### Pre-Inkrement und -Dekrement

→ **Erst** die Inkrement-/Dekrement-Anweisung ausführen, **danach** die Zuweisung:

```
1 int i=22;
2 int j=100;
3
4 // Nach diesem Statement: m hat den Wert 23;
5 // i hat den Wert 23
6 int m = ++i;
7
8 // Nach diesem Statement: n hat den Wert 99;
9 // j hat den Wert 99
10 int n = --j;
```

# Neu eingeführte arithmetische Operatoren in Java

## Inkrement- und Dekrement-Operatoren

### Post-Inkrement und -Dekrement

→ **Erst** die Zuweisung ausführen, **danach** die Inkrement-/Dekrement-Anweisung:

```
1 int i=22;
2 int j=100;
3
4 // Nach diesem Statement: m hat den Wert 22;
5 // i hat den Wert 23
6 int m = i++;
7
8 // Nach diesem Statement: n hat den Wert 100;
9 // j hat den Wert 99
10 int n = j--;
```

# Operatoren in Kontrollstrukturen

Für die Abfragen der Bedingungen innerhalb von Fallunterscheidungen und Schleifen benötigen wir:

- Vergleichsoperatoren
- logische Operatoren

# Bereits bekannte Vergleichsoperatoren aus Python ... ... sehen in Java genau gleich aus!

Python-Vergleichsoperator	Java-Vergleichsoperator	Erläuterung
<code>a &lt; b</code>	<code>a &lt; b</code>	kleiner
<code>a &lt;= b</code>	<code>a &lt;= b</code>	kleiner gleich
<code>a == b</code>	<code>a == b</code>	gleich
<code>a != b</code>	<code>a != b</code>	ungleich
<code>a &gt;= b</code>	<code>a &gt;= b</code>	grösser gleich
<code>a &gt; b</code>	<code>a &gt; b</code>	grösser

# Bereits bekannte, logische Operatoren aus Python ... ... sehen in Java etwas anders aus:

Python-Operator	Java-Operator	Erläuterung
a or b	a    b	Oder (In Java: Kurzschlussoperator; b wird nur ausgewertet, wenn a false ist.)
a or b	a   b	Oder
a and b	a && b	Und (In Java: Kurzschlussoperator; b wird nur ausgewertet, wenn a true ist.)
a and b	a & b	Und
not a	!a	Negation

## Bereits bekannte Bitoperatoren aus Python ... ... sehen in Java fast gleich aus!

Python-Bitoperation	Java-Bitoperation	Erläuterung
$a \& b$	$a \& b$	Bitweises Und von zwei Ganzzahl-Variablen
$a   b$	$a   b$	Bitweises Oder von zwei Ganzzahl-Variablen
$a ^ b$	$a ^ b$	Bitweises Exklusiv-Oder zweier Ganzzahl-Variablen
$\sim a$	$\sim a$	Bitweise Negation (Einerkomplement) einer Ganzzahl-Variablen

## Bereits bekannte Bitoperatoren aus Python ... ... sehen in Java fast gleich aus!

Python-Bitoperation	Java-Bitoperation	Erläuterung
$a >> n$	$a >> n$	Vorzeichenerhaltender Rechtsshift von a um n Bits
$a << n$	$a << n$	Linksshift von a um n Bits
nicht vorhanden	$a >>> n$	Vorzeichenloser Rechtsshift von a um n Bits (es werden immer 0en nachgeschoben) →gibt es nur in Java, nicht in Python

# Typkonvertierung in Java

## Begriff Typkonvertierung / Typecast

- Wir haben gelernt: In Java erfolgt die Typisierung von Variablen **explizit**.
- Bisher: Verwendung von Operatoren nur bei **gleichtypigen** Operanden.
- Frage: Was, wenn wir beispielsweise einen double-Operanden mit einem int-Operanden multiplizieren müssen?

# Typkonvertierung in Java

## Zwei Arten der Typkonvertierung

- **Implizite** Typkonvertierung: Wird von Java automatisch durchgeführt.  
Nur möglich, wenn wir von **klein → groß** konvertieren.
- **Explizite** Typkonvertierung: Hier geben wir den gewünschten Datentyp einer Variablen bei der Operation oder Zuweisung an.

# Typkonvertierung in Java

## Beispiel 1: erweiternder impliziter Typecast

```
1 short var1 = 20000;  
2 short var2 = 5;  
3  
4 int var3 = var1 * var2;
```

Beide Operanden werden **vor** der Multiplikation nach int konvertiert!

→ Produkt übersteigt Wertebereich von short!

→ Hier: Kein Problem, da var3 vom Typ int.

→ Ergebnis wird vor der Zuweisung automatisch nach int konvertiert.

# Typkonvertierung in Java

## Beispiel 2: erweiternder impliziter Typecast

```
1 int var1 = 20000;  
2 short var2 = 5;  
3  
4 int var3 = var1 * var2;
```

→ var2 wird vor der Multiplikation automatisch nach int konvertiert.

→ Ergebnis wird vor der Zuweisung automatisch nach int konvertiert – OK!

# Typkonvertierung in Java

## Beispiel 3: fehlerhafter impliziter Typecast

```
1 short var1 = 20000;  
2 short var2 = 5;  
3  
4 // Klappt nicht: var1 * var2 liefert int-Ergebnis!  
5 short var3 = var1 * var2;
```

→ Compiler bricht mit Fehlermeldung ab!

→ Wir haben hier versucht, ein int - Ergebnis an einen short zuzuweisen!

# Wichtigste erweiternde, implizite Konvertierungsregeln

Ergebnisse von Operationen mit

- arithmetischen Operatoren,
- Vorzeichenoperatoren,
- bitweisen logischen Operatoren
- Bitmanipulationsoperatoren

werden implizit in *int* konvertiert, sofern der oder die Operanden *char*, *byte*, *short* oder *int* sind.

# Wichtigste erweiternde, implizite Konvertierungsregeln

Datentyp	Kann auch zugewiesen werden an ...
byte	short, int, long, float, double
short	int, long, float, double
int	long, float, double
long	float, double
char	int, long, float, double
float	double

# Explizite Typecasts

## Syntax expliziter Typecasts

Wir schreiben **explizit vor** den zu konvertierenden Ausdruck oder **vor** die zu konvertierende Variable, welchen Datentyp wir wünschen:

```
variable = (GewuenschterTyp) variable2;
```

oder

```
oder variable = (GewuenschterTyp) ausdruck;
```

# Explizite Typecasts

## Beispiel für explizite Typecasts

```
1   short a=50;
2   short b=70;
3   short c=1000;
4
5   // Expliziter Typecast des Ergebnisses
6   // a * b von int --> short
7   // ergebnis1 = 3500 --> OK!
8   short ergebnis1 = (short)(a * b);
9
10  // Expliziter Typecast des Ergebnisses
11  // b * c von int --> short; b * c = 70000
12  // ABER: ergebnis2 = 4464!!!! -- Warum??
13  short ergebnis2 = (short)(b*c);
```

# Sichtbarkeit von Variablen in Java – Teil 1

Variablen sind ...

... innerhalb ihres **Gültigkeitsbereiches** sichtbar.

# Sichtbarkeit von Variablen – Teil 1

Was ist ein Gültigkeitsbereich?

Gültigkeitsbereich einer Variablen

=

Codebereich, in dem die Variable verwendbar ist.

# Gültigkeitsbereiche von Variablen in Java

In Java unterscheiden wir

## ■ Instanzvariablen (Objektvariablen)

- „global“ bekannt innerhalb aller Methoden eines Objektes
- Wird für mit **new** erzeugte Objekt dieser Klasse neu angelegt.
- **n Objekte → n Variablen-Instanzen**

## ■ Klassenvariablen

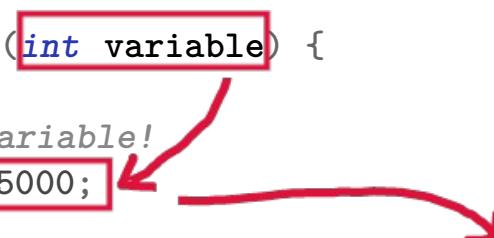
- Modifizierer **static** (später!)
- „global“ bekannt innerhalb aller Methoden einer Klasse
- Existiert nur **einmal gemeinsam** für alle mit **new** erzeugten Objekte dieser Klasse.
- **n Objekte → 1 Variablen-Instanz**

## ■ lokale Variablen

- nur innerhalb der Methode bekannt, in der sie definiert wurden.
- → außerhalb der Methode **nicht** verwendbar!

# Beispiel: Objektvariablen vs. lokale Variablen

```
public class Variablen2 {  
  
    // Objektvariable: wirkt innerhalb der Methoden dieser Klasse "global"  
    int variable=100;  
  
    void methode1(int variable) {  
  
        // lokale Variable!  
        variable = 5000;  
  
        System.out.println("variable: "+variable);  
    }  
  
    void methode2() {  
        // lokale Variable!  
        int variable = -2;  
        System.out.println("variable: "+variable);  
    }  
  
    void methode3() {  
        // Zugriff auf Objektvariable oben!  
        System.out.println("variable: "+variable);  
    }  
} // end class
```

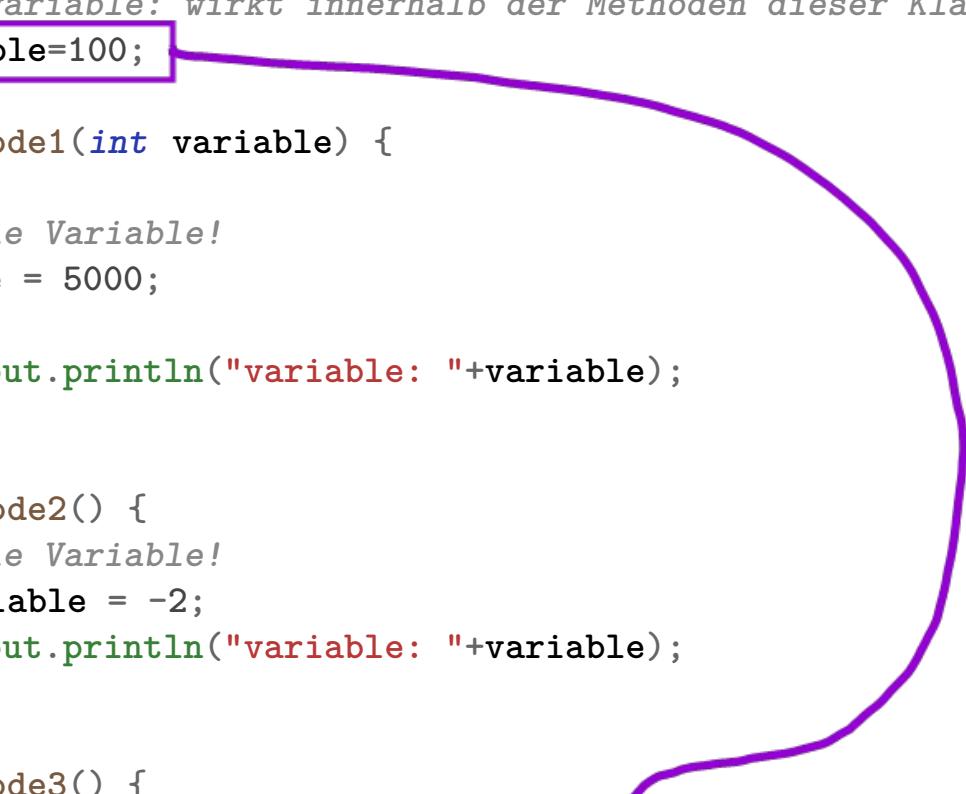


# Beispiel: Objektvariablen vs. lokale Variablen

```
public class Variablen2 {  
  
    // Objektvariable: wirkt innerhalb der Methoden dieser Klasse "global"  
    int variable=100;  
  
    void methode1(int variable) {  
  
        // lokale Variable!  
        variable = 5000;  
  
        System.out.println("variable: "+variable);  
    }  
  
    void methode2() {  
        // lokale Variable!  
        int variable = -2;   
        System.out.println("variable: "+variable);  
    }  
  
    void methode3() {  
        // Zugriff auf Objektvariable oben!  
        System.out.println("variable: "+variable);  
    }  
} // end class
```

# Beispiel: Objektvariablen vs. lokale Variablen

```
public class Variablen2 {  
  
    // Objektvariable: wirkt innerhalb der Methoden dieser Klasse "global"  
    int variable=100;  
  
    void methode1(int variable) {  
  
        // lokale Variable!  
        variable = 5000;  
  
        System.out.println("variable: "+variable);  
    }  
  
    void methode2() {  
        // lokale Variable!  
        int variable = -2;  
        System.out.println("variable: "+variable);  
    }  
  
    void methode3() {  
        // Zugriff auf Objektvariable oben!  
        System.out.println("variable: "+variable);  
    }  
} // end class
```



# Beispiel: Objektvariablen vs. lokale Variablen

```
public class Variablen2Main {  
    public static void main(String [] args) {  
  
        Variablen2 worker = new Variablen2();  
        worker.methode1(42);  
        worker.methode2();  
        worker.methode3();  
    } // end method main()  
} // end class
```

Zum selber Ausprobieren: Was wird hier ausgegeben? Warum?

# Bereits bekannte Kontrollstrukturen aus Python

Wir kennen bereits

- Fallunterscheidung mit if – elif – else
- Kopfgesteuerte Schleife mit while
- Kopfgesteuerte Schleife mit for

# Verfügbare Kontrollstrukturen in Java

Java stellt bereit

- Fallunterscheidung mit if – else if – else
- Fallunterscheidung mit switch – case
- Kopfgesteuerte Schleife mit while
- Kopfgesteuerte Schleife mit for
- Fußgesteuerte Schleife mit do – while

## Fallunterscheidung mit if - else if - else

Wird für kompliziertere Abfragen verwendet, beispielsweise:

- Größenvergleiche
- Wertebereichsabfragen
- Bereichsüberprüfungen bei Strings
- ...

# Fallunterscheidung mit if - else if - else

## Syntax if - else if - else

```
1 if (bedingung1) {  
2     anweisung1;  
3     // ...  
4 }  
5 else if (bedingung2) {  
6     anweisungN;  
7     // ...  
8 }  
9 // ... weitere else if()-Bedingungen  
10 else {  
11     anweisungM;  
12     // ...  
13 }
```

# Fallunterscheidung mit if - else if - else

## Beispiel if - else if - else if - Teil 1

```
1 import java.util.Scanner;  
2  
3 public class Fallunterscheidung1 {  
4  
5     public static void main(String[] args) {  
6  
7         // "Werkzeug" fuer Tastatureingaben  
8         Scanner eingabe = new Scanner(System.in);  
9  
10        // Eingabe vom Nutzer anfordern  
11        System.out.print("Bitte eine Ganzzahl eingeben: ");  
12        int wert = eingabe.nextInt(); // naechste Seite ..
```

# Fallunterscheidung mit if - else if - else

## Beispiel if - else if - else if - Teil 2

```
13 // Fallunterscheidung: Wertebereiche der Eingabe
14 if (wert >= 0 && wert <= 5) {
15     System.out.println("Wert liegt zwischen 0 und 5");
16 } // end if
17 else if (wert >= 6 && wert <= 10) {
18     System.out.println("Wert liegt zwischen 6 und 10");
19 } // end else if
20 else {
21     System.out.println("Ausserhalb der beiden Bereiche!");
22 } // end else
23 } // end method main()
24 } // end class
```

## Fallunterscheidung mit if - else if - else

Beispiel if - else if - else - Eingabeszenario

Bitte eine Ganzzahl eingeben: 4

Wert liegt zwischen 0 und 5

# Fallunterscheidung mit switch-case

Wird für einfachere Abfragen verwendet:

- Vergleich mit konstanten Werten
- Wertebereiche können durch „Fall-through“ (Weglassen der **break**-Anweisung) abgefragt werden.
- Wertebereiche werden jedoch oft einfacher durch eine **if**-Konstruktion abgefragt.

# Fallunterscheidung mit switch-case

## Syntax switch-case – Grundform

```
1 switch(wert) {  
2     case konstantWert1: anweisungen1; break;  
3     case konstantWert2: anweisungen2; break;  
4     // .. weitere case-Anweisungen  
5     case konstantWertN: anweisungenN; break;  
6     default: anweisungAndereWerte; break;  
7 }
```

# Fallunterscheidung mit switch-case

## Beispiel switch-case - Teil 1

```
1 import java.util.Scanner;  
2  
3 public class Fallunterscheidung2 {  
4  
5     public static void main(String[] args) {  
6  
7         // "Werkzeug" fuer Tastatureingaben  
8         Scanner eingabe = new Scanner(System.in);  
9  
10        // Eingabe vom Nutzer anfordern  
11        System.out.print("Bitte eine Ganzzahl eingeben: ");  
12        int wert = eingabe.nextInt();
```

# Fallunterscheidung mit switch-case

## Beispiel switch-case - Teil 2

```
13 // switch-case ueberprueft auf einzelne, konstante
14 // Werte
15 switch (wert) {
16     case 1: System.out.println("Eingabe 1!"); break;
17     case 2: System.out.println("Eingabe 2!"); break;
18     case 3: System.out.println("Eingabe 3!"); break;
19     case 4: System.out.println("Eingabe 4!"); break;
20     default: System.out.println("Was anderes!"); break;
21 } // end switch
22
23 } // end method main()
24 } // end class
```

# Fallunterscheidung mit switch-case

## Besonderheit fall-through

```
//  
// ...  
// Eingabe vom Nutzer anfordern  
System.out.print("Bitte eine Ganzzahl eingeben: "); } }  
int wert = eingabe.nextInt();  
  
// Fallunterscheidung  
switch (wert) {  
    case 1: System.out.println("Erste Ausgabe!");  
    case 2: System.out.println("Zweite Ausgabe!");  
    case 3: System.out.println("Dritte Ausgabe!"); break; // break be-  
    case 4: System.out.println("Vierte Ausgabe"); break;  
    default: System.out.println("Was anderes!"); break;  
} // end switch
```

z.B.  
wert = 1

break beendet den Fall

# Fallunterscheidung mit switch-case

Besonderheit fall-through – Eingabeszenario 1

Bitte eine Ganzzahl eingeben: 1

Erste Ausgabe!

Zweite Ausgabe!

Dritte Ausgabe!

- **case 1** wird durchlaufen – trifft zu. Er wird aber nicht mit **break** beendet.  
Daher:
- Danach werden **cases 2 und 3** durchlaufen, ehe die **break**-Anweisung in **case 3** die **switch**-Anweisung abbricht.

## Fallunterscheidung mit switch-case

Besonderheit fall-through – Eingabeszenario 2

Bitte eine Ganzzahl eingeben: 2

Zweite Ausgabe!

Dritte Ausgabe!

- **case 1** wird nicht durchlaufen – trifft nicht zu.
- **case 2** wird durchlaufen – trifft zu und wird aber nicht mit **break** beendet.
- Daher wird auch **case 3** durchlaufen, ehe **break** die **switch**-Anweisung abbricht.

# Kopfgesteuerte Schleife mit while

Für kompliziertere Wiederholungsanweisungen mit z.B.

- Größenvergleichen
- Wertebereichsabfragen
- Bereichsüberprüfungen bei Strings
- Stringvergleichen
- ...

# Kopfgesteuerte Schleife mit while

## Syntax von while

```
1 while (bedingung) {  
2     zuWiederholendeAnweisung1;  
3     zuWiederholdendeAnweisung2;  
4     // ...  
5     zuWiederholendeAnweisungN;  
6 }
```

# Kopfgesteuerte Schleife mit while

## Beispiel while - Teil 1

```
1 import java.util.Scanner;  
2  
3 public class SchleifeWhile1 {  
4  
5     public static void main(String [] args) {  
6  
7         // "Werkzeug" fuer Tastatureingaben  
8         Scanner eingabe = new Scanner(System.in);  
9  
10        // Eingabe vom Nutzer anfordern  
11        System.out.print(  
12            "Bitte eine Zahl zwischen 1 und 10 eingeben: ");  
13        int zahl = eingabe.nextInt();
```

# Kopfgesteuerte Schleife mit while

## Beispiel while - Teil 2

```
14 // Schleife laeuft, so lange die eingetippte Zahl
15 // zwischen 1 und 10 liegt
16 while (zahl >= 1 && zahl <= 10) {
17     System.out.print("Noch eine Zahl zwischen 1 und 10: ");
18     zahl = eingabe.nextInt();
19 } // end while
20
21 System.out.println(
22     "Letzte Zahl war nicht mehr zwischen 1 und 10!");
23
24 } // end method main()
25 } // end class
```

# Kopfgesteuerte Schleife mit while

## Eingabeszenario while

Bitte eine Zahl zwischen 1 und 10 eingeben: 5

Noch eine Zahl zwischen 1 und 10: 3

Noch eine Zahl zwischen 1 und 10: 66

Letzte Zahl war nicht mehr zwischen 1 und 10!

- Sofern der erste eingegebene Wert zwischen 1 und 10 liegt: Einstieg in den **Schifenrumpf**
- Schleife wird durchlaufen, so lange der letzte eingegebene Wert zwischen 1 und 10 liegt.
- Schleifenrumpf fordert eine weitere Nutzereingabe an.
- Sobald ein Wert außerhalb des geforderten Intervalls eingegeben wurde: Schleife ist **beendet**.

# Kopfgesteuerte Schleife mit for

Wird verwendet für

- Zählschleifen (z.B. Durchlauf von 1 bis n)
- Iterationsschleifen über Arrays oder iterierbare Listen (foreach-Schleife)

# Kopfgesteuerte Schleife mit for

## Syntax von for als Zählschleife

```
1 for (startInitialisierung; bedingung; iterationsAnweisung) {  
2     zuWiederholendeAnweisung1;  
3     zuWiederholdendeAnweisung2;  
4     // ...  
5     zuWiederholendeAnweisungN;  
6 }
```

# Kopfgesteuerte Schleife mit for

## Beispiel for als Zählschleife – Teil 1

```
1 import java.util.Scanner;  
2  
3 public class SchleifeFor1 {  
4  
5     public static void main(String [] args) {  
6  
7         // "Werkzeug" fuer Tastatureingaben  
8         Scanner eingabe = new Scanner(System.in);  
9  
10        // Eingabe vom Nutzer anfordern  
11        System.out.print("Bitte einen Endwert eingeben: ");  
12        int zahl = eingabe.nextInt();
```

# Kopfgesteuerte Schleife mit for

## Beispiel for als Zählschleife – Teil 2

```
13 // Schleife laeuft, bis i den Wert zahl erreicht
14 for (int i=1; i<= zahl; i++) {
15
16     System.out.println("i: " + i);
17
18 } // end for
19
20 System.out.println("Schleife zu Ende!");
21
22 eingabe.close(); // Scanner schliessen
23
24 } // end method main()
25 } // end class
```

# Kopfgesteuerte Schleife mit for

## Syntax von for als Iterationsschleife

```
1 // Statt eines Arrays kann auch eine Liste (zB. ArrayList)  
2 // durchlaufen werden!  
3 for (datentyp laufvariable : array) {  
    zuWiederholendeAnweisung1;  
    zuWiederholdendeAnweisung2;  
    // ...  
    zuWiederholendeAnweisungN;  
}
```

# Kopfgesteuerte Schleife mit for

## Beispiel for als Iterationsschleife

```
1 public class SchleifeFor2 {  
2     public static void main(String [] args) {  
3           
4         int [] zahlen = {32, 11, -12, -10};  
5           
6         // Variable wert lauft ueber alle Elemente im Array  
7         for (int wert : zahlen) {  
8             System.out.println("wert ist jetzt: "+wert);  
9         } // end for  
10     } // end method main()  
11 } // end class
```

# for-Schleife: Ein Matrixalgorithmus

## Problemstellung

- Ganzzahlige Matrix einer bestimmten Größe erzeugen.
- Diese Matrix mit einem „Schachbrettmuster“ befüllen:
  - Wert 0: schwarz
  - Wert 1: weiss
- Die Matrix zurückgeben.
- Die Werte „matrixförmig“ ausgeben.

## for-Schleife: Ein Matrixalgorithmus

0	255	0	255	0
255	0	255	0	255
0	255	0	255	0

### Beispiel 1:

rows = 3

columns = 5

0	255	0	255
255	0	255	0

### Beispiel 2:

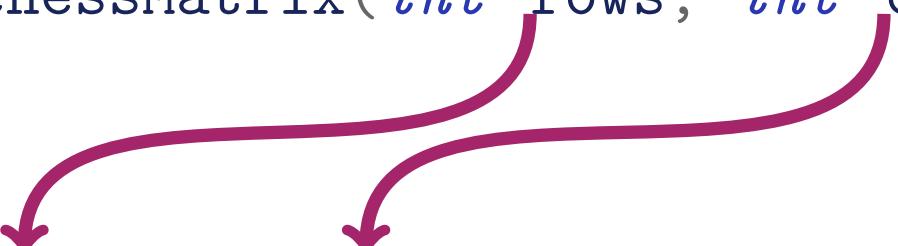
rows = 2

columns = 4

# for-Schleife: Ein Matrixalgorithmus

## Methode zur Erzeugung der Schachbrettmatrix – Teil 1

```
1 public class MatrixExample1 {  
2  
3     // creates a matrix filled with chessboard pattern  
4     public static int [][] chessMatrix(int rows, int cols){  
5         final int black=0;  
6         final int white=255;  
7         // create matrix  
8         int [][] mat = new int [rows] [cols];
```



# for-Schleife: Ein Matrixalgorithmus

## Methode zur Erzeugung der Schachbrettmatrix – Teil 2

```
9   for (int i=0; i<rows; i++) {  
10     for (int j=0; j<cols; j++) {  
11       if ((i%2==0 && j%2==0) || (i%2==1 && j%2==1)) {  
12         mat[i][j]=black; // black cells  
13       }  
14       else {  
15         mat[i][j] = white; // white cells  
16       }// end else  
17     } // end for j (columns)  
18   } // end for i (columns)  
19   return mat;  
20 } // end method chessMatrix()
```

# for-Schleife: Ein Matrixalgorithmus

## Methode zur Ausgabe der Matrix

```
21  public static void printMatrix(int [] [] mat) {  
22      // mat.length: number of rows  
23      for (int i=0; i<mat.length; i++) {  
24          for(int j=0; j<mat[i].length; j++) {  
25              // TAB after each number  
26              System.out.print(mat[i][j]+\t");  
27          } // end for j (columns)  
28          System.out.println(); // line break  
29      } // end for i (rows)  
30  } // end method printMatrix  
31 } // end class
```

# for-Schleife: Ein Matrixalgorithmus

Aufruf der beiden Methoden in einer main()-Klasse

```
0 public class MatrixMain {  
1  
2     public static void main(String[] args) {  
3         int rows = 3, cols=5;  
4         // Erzeugung der Matrix  
5         int [] [] matr = MatrixExample.chessMatrix(rows, cols);  
6  
7         // Ausgabe der Matrix  
8         MatrixExample.printMatrix(matr);  
9  
10    } // end main()  
11 } // end class
```

## Fußgesteuerte Schleifen – wozu?

Verwendung immer dann, wenn ...

- ... mindestens **ein** Schleifendurchlauf stattfinden soll.
- Sprachen, die keine fußgesteuerten Schleifen anbieten, müssen für derartige Algorithmen immer „doppelten“ Code mitführen.

# Fußgesteuerte Schleifen – wenn es sie nicht gibt

## Beispielproblem – Teil 1:

```
import java.util.Scanner;

public class SchleifeWhile1 {

    public static void main(String [] args) {

        // "Werkzeug" fuer Tastatureingaben
        Scanner eingabe = new Scanner(System.in);
        int zahl; // Variable fuer Zahleneingabe

        // Eingabe vom Nutzer anfordern
        System.out.print("Bitte eine Zahl zwischen 1 und 10 eingeben: ");
        zahl = eingabe.nextInt();
    }
}
```

Anweisungsfolge erscheint  
einmal vor dem Schleifenkopf  
...



# Fußgesteuerte Schleifen – wenn es sie nicht gibt

## Beispielproblem – Teil 2:

```
// Schleife läuft, so lange die eingetippte
// Zahl zwischen 1 und 10 liegt
while (zahl >= 1 && zahl <= 10) {

    // Eingabe vom Nutzer anfordern
    System.out.print("Bitte eine Zahl zwischen 1 und 10 eingeben: ");
    zahl = eingabe.nextInt();

} // end while

System.out.println(
    "Letzte Zahl war nicht mehr zwischen 1 und 10!");

} // end method main()
} // end class
```

Anweisungsfolge erscheint ein zweites Mal im Schleifenrumpf. Fehleranfällig!!!



# Fußgesteuerte Schleife mit do – while

## Syntax von do – while

```
1 // Mindestens EIN Durchlauf, bevor die Durchfuehrungs-
2 // bedingung am FUSS der Schleife geprueft wird!
3 do {
4     zuWiederholendeAnweisung1;
5     zuWiederholdendeAnweisung2;
6     // ...
7     zuWiederholendeAnweisungN;
8 } while (wiederholungsBedingung);
```

# Fußgesteuerte Schleife mit do – while

## Beispielproblem mit do – while – Teil 1

```
import java.util.Scanner;  
  
public class SchleifeDoWhile1 {  
  
    public static void main(String [] args) {  
  
        // "Werkzeug" fuer Tastatureingaben  
        Scanner eingabe = new Scanner(System.in);  
        int zahl; // Variable fuer Zahleneingabe
```

Beginn der Klasse: Wie vorher mit while ...



# Fußgesteuerte Schleife mit do – while

## Beispielproblem mit do – while – Teil 2

```
do {  
  
    // Eingabe vom Nutzer anfordern  
    System.out.print("Bitte eine Zahl zwischen 1 und 10 eingeben: ");  
    zahl = eingabe.nextInt();  
  
} while (zahl >= 1 && zahl <= 10);  
  
System.out.println(  
    "Letzte Zahl war nicht mehr zwischen 1 und 10!");  
  
} // end method main()  
} // end class
```

Anweisungsfolge erscheint nur noch  
1x im Schleifenrumpf!

# Fußgesteuerte Schleife mit do – while

## Beispielproblem mit do – while – Teil 2

```
do {  
    // Eingabe vom Nutzer anfordern  
    System.out.print("Bitte eine Zahl zwischen 1 und 10 eingeben: ");  
    zahl = eingabe.nextInt();  
}  
    while (zahl >= 1 && zahl <= 10);  
  
System.out.println(  
    "Letzte Zahl war nicht mehr zwischen 1 und 10!");  
  
} // end method main()  
} // end class
```

Bedingungsprüfung erst nach 1-maliger Durchführung der Anweisungsfolge!

# Zusammenfassung Kontrollstrukturen

## Wann welche Fallunterscheidung?

- Fallunterscheidung mit if – else if immer dann, wenn **kompliziertere logische Abfragen** notwendig sind (Wertebereiche, etc.)
- Fallunterscheidung mit switch – case immer dann, wenn **konstante Werte** unterschieden werden.

# Zusammenfassung Kontrollstrukturen

## Wann welche Schleife?

- **Kopfgesteuerte Schleife** mit `while` immer dann, wenn **von Beginn an** der Durchlauf **verhindert** werden soll, falls das Durchführungskriterium nicht erfüllt ist.
- **Kopfgesteuerte Zählschleife** mit `for` immer dann, wenn ein **Laufzähler** im Schleifenablauf inkrementiert oder dekrementiert werden soll.
- **Kopfgesteuerte Iterationsschleife** mit `for` immer dann, wenn die **Elemente eines Arrays oder einer Liste** durchlaufen werden sollen.
- **Fußgesteuerte Schleife** mit `do - while` immer dann, wenn der Schleifenrumpf **mindestens einmal** durchlaufen werden soll.

## Im vergangenen Semester: Funktionen in Python

Funktionen in Python: wir erinnern uns ...

- In Python konnten Funktionen **innerhalb** und **ausserhalb** von Klassen vereinbart werden.
- Funktionen **innerhalb** von Klassen werden auch als **Methoden** bezeichnet.

## Im vergangenen Semester: Funktionen in Python

Wir erinnern uns: Syntax von Funktionsdefinitionen in Python

```
def funcName(param1, param2, ...):  
    Anweisung1  
    Anweisung2  
    # ...  
    AnweisungN
```

→ Typisierung von Parametern und Rückgabewert erfolgte **implizit!**

# In diesem Semester: Methoden in Java

## Methoden in Java

- Methoden beinhalten in sich abgeschlossene Unteraufgaben innerhalb einer Klasse.
- In Java können Funktionen **nur innerhalb** von Klassen vereinbart werden.
- Daher heißen in Java unsere Funktionen **immer Methoden**.

→ Typisierung von Parametern und Rückgabewert erfolgt **explizit!**

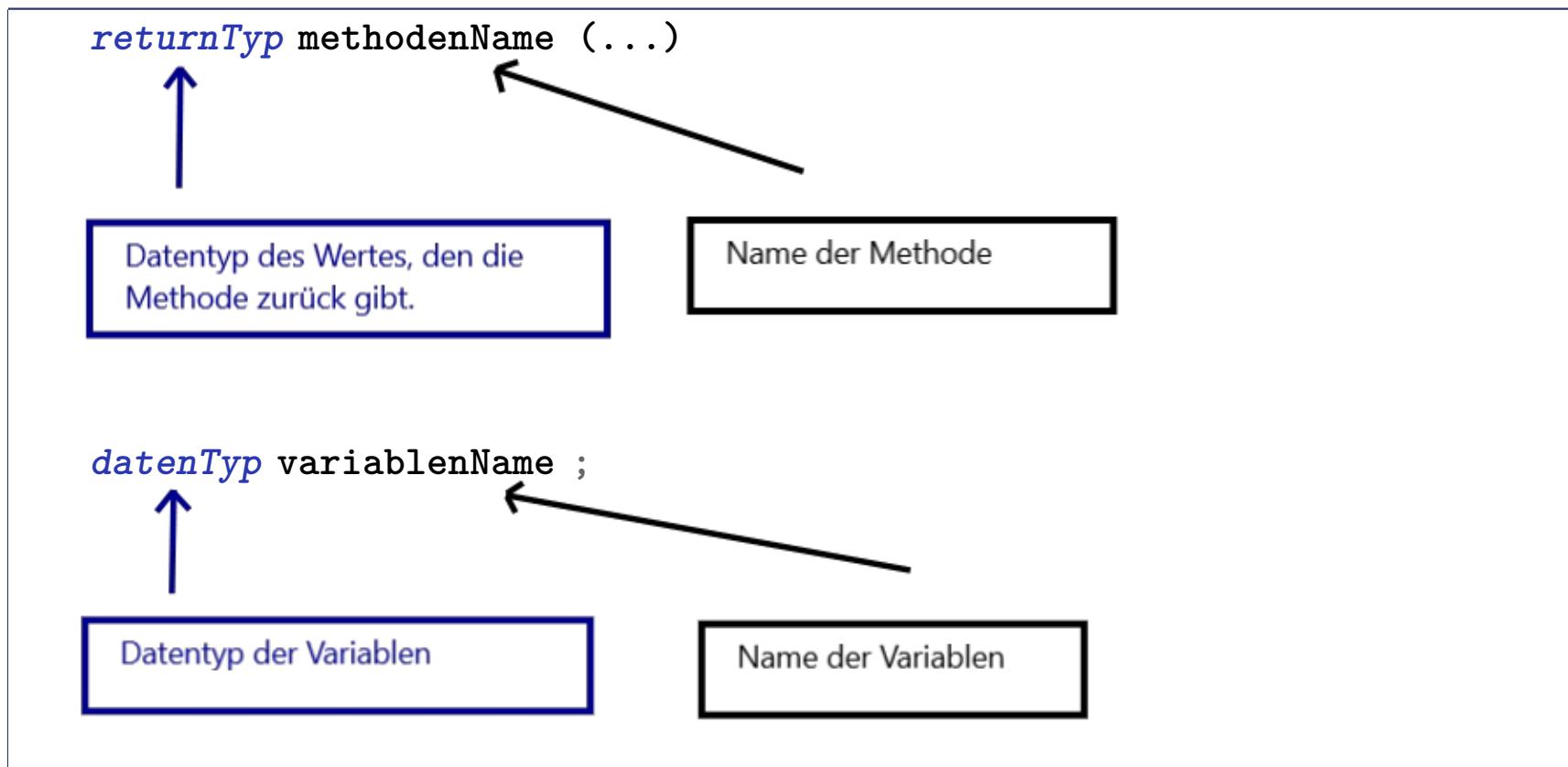
# Methoden in Java: Vereinbarungssyntax

## Erste Variante: vereinfachte Vereinbarungssyntax

```
returnTyp methodName(typ1 param1, typ2 param2, ...) {  
    Anweisung1;  
    Anweisung2;  
    // ...  
    AnweisungN;  
}
```

# Methoden in Java: Vereinbarungssyntax

## Erste Variante: vereinfachte Vereinbarungssyntax



→ Syntax für die Vereinbarung von Methodenköpfen ist sehr ähnlich der Syntax zur Vereinbarung von Variablen!

# Methoden ohne Rückgabe

Methode gibt nichts zurück: Rückgabetyp void

```
1 void sagHallo(String name) {  
2     System.out.println("Hallo " + name);  
3 }
```

→ void-Methoden haben am Ende der Methode **keine** return-Anweisung!

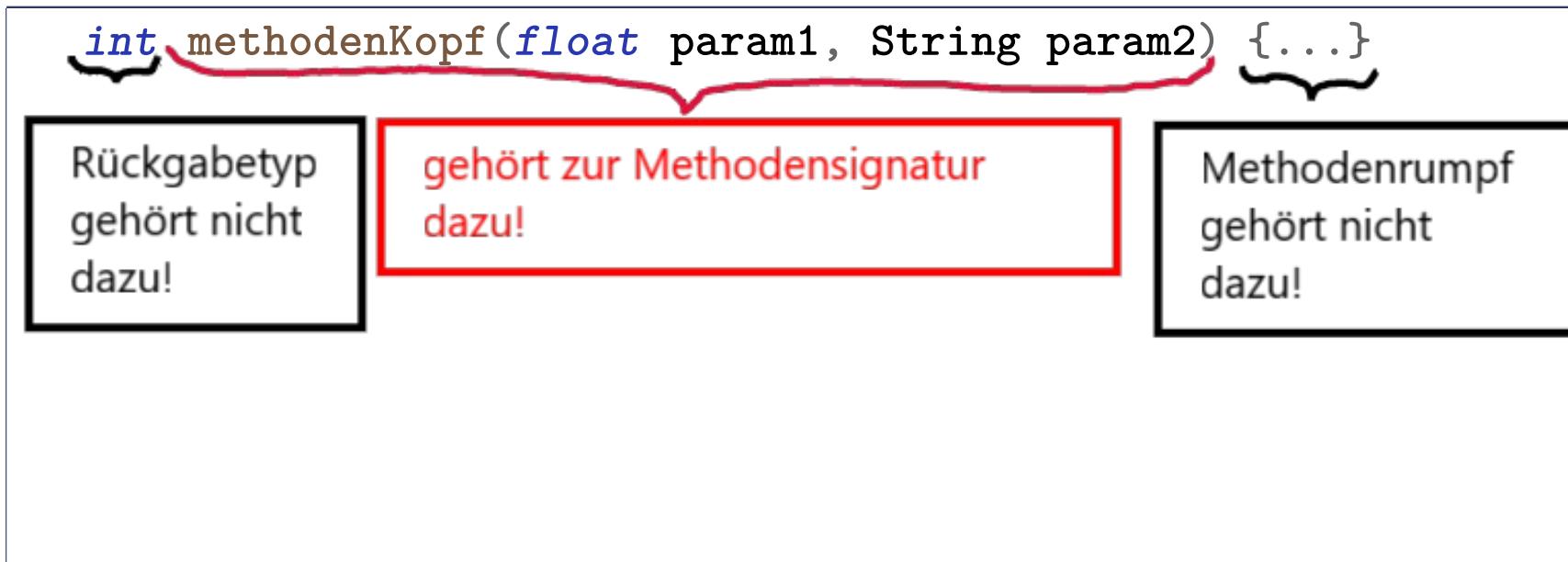
# Methoden mit Rückgabe

Methode gibt ein Ergebnis zurück: Rückgabetyp und return-Statement

```
1 int altersBerechnung(int alter) {  
2     int ergebnis = alter - 18;  
3     return ergebnis;  
4 }
```

- → Methoden **mit Rückgabetyp** haben am Ende der Methode **zwingend** eine return-Anweisung!
- Der Typ des zurückgegebenen Wertes muss dem im Kopf angegebenen Rückgabetyp (oben: returnType) entsprechen!

# Methoden in Java: Signatur einer Methode ... ... ist der Methodenkopf ohne Angabe des Rückgabetyps



→ Dieser Begriff wird später in der OOP beim Überladen von Methoden benötigt!

# Parameterübergabe: Formale und aktuelle Parameter

```
public class Methoden1 {  
    void hello1(String name) {  
        System.out.println("Hallo " + name);  
    } // end hello1()
```

Formaler Parameter: "Schablone" bei  
der Definition der Methode!

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

```
// Arbeitsobjekt DIESER Klasse anlegen
```

```
Methoden1 test = new Methoden1();
```

```
test.hello1("Obelix");
```

Aktueller Parameter: Konkreter Wert,  
der beim Aufruf der Methode  
übergeben wird!

```
} // end method main()  
} // end class
```

# Parameterübergabe: Formale und aktuelle Parameter

## Merke: Methodendefinition vs. Methodenaufruf

- Bei der Methodendefinition sind **formale Parameter** angegeben. Sie spezifizieren, welche Übergaben die Methode überhaupt erwartet. Ohne deren Namen könnte der Methodenkörper nicht auf übergebene Daten zugreifen.
- Beim Methodenaufruf werden die tatsächlich im Programm entstandenen Daten durchgereicht. Dies sind die **aktuellen Parameter**.

# Parameterübergabe: Call-by-value vs. Call-by-reference

```
public class Methoden2 {  
    void testMethode1(Person pers) {  
        pers.ausgabe();  
    } // end testMethode0()  
  
    double testMethode2(double m, double x, double b) {  
        double y = m*x + b;  
  
        return y;  
    } // end testMethode2()
```

referenz

Kopierte Werte auf  
dem lokalen Speicher-  
bereich der zwei  
Methoden

2.0
1.5
0.5

# Parameterübergabe: Call-by-value vs. Call-by-reference

```
public static void main(String [] args) {  
    Person p1 = new Person("Erna Etepete");  
    double mNeu = 2.0; 2.0  
    double xNeu = 1.5; 1.5  
    double bNeu = 0.5; 0.5  
  
    // Arbeitsobjekt DIESER Klasse anlegen  
    Methoden2 test = new Methoden2();  
  
    test.testMethode1(p1);  
    test.testMethode2(mNeu, xNeu, bNeu);  
  
} // end method main()  
} // end class Methoden2
```

*referenz*

*Erna Etepete*

*Werte auf dem lokalen Stack von main()*

*Bei der Übergabe werden Kopien der Werte angelegt.*

*Objekt auf dem Frei-Speicher existiert nur 1x*

# Parameterübergabe: Call-by-value vs. Call-by-reference

## Genaue Betrachtung der Parameterübergabe beim Aufruf

- Die Variablen **elementaren Datentyps** liegen in *main()* auf dem dortigen lokalen Speicherbereich (auf dem Stack von *main()*).
- Die **Referenz** auf das Personen-Objekt liegt ebenfalls auf dem Stack von *main()*.
- Das mit *new* angelegte **Objekt** vom Typ *Person* liegt dagegen auf dem **Freispeicherbereich** oder **Heap** des Programms.
- Bei der Übergabe der Variablen **elementaren** Typs wird eine **Kopie der Werte** auf dem **Stack von testMethode2()** abgelegt. Mit diesen Kopien arbeitet unsere Methode bei ihrer Berechnung.
- Diese Art Methodenaufruf heisst **Call-by-value**.

# Parameterübergabe: Call-by-value vs. Call-by-reference

Genaue Betrachtung der Parameterübergabe beim Aufruf

- Bei der Übergabe der **Referenzvariablen** wird nur die **Referenz** – also die Adresse der Variablen kopiert.
- Das Personen-Objekt existiert weiterhin **nur einmal**.
- Diese Art Methodenaufruf wird oft ungenauer Weise als **Call-by-reference** bezeichnet.
- Dies ist nicht ganz richtig: In Java wird hier ebenfalls ein **Wert** übergeben – nämlich der Adreßwert des Objektes!
- Andere Programmiersprachen kennen eine speziell gekennzeichnete Referenz-Übergabe: Sie wird wirklich als **call-by-reference** bezeichnet.

Fakultät Informatik

# Programmierung 2

Einstieg in die OOP: Klassen und  
Objekte/Instanzen

## Vorwort

### Einstieg in Java

#### Einstieg in die OOP: Klassen und Objekte/Instanzen

Objekte in der OOP

Klassen in Java

Vereinbarung von Klassen in Java

Erzeugung und Verwendung von Objekten/Instanzen

Sichtbarkeit von Variablen und Methoden – private, public, protected und Package (kein Modifier)

Instanzvariablen/Instanzmethoden vs.

Klassenvariablen/Klassenmethoden

Arrays aus Objekten/Instanzen

Listen aus Objekten/Instanzen



## Die Klasse String

Graphische Darstellung von Klassen mit der UML – Teil 1

Vererbungsbeziehungen

Exceptions

Einstieg Streams

# Objekte in der realen Welt – abgebildet in der Informatik

## Beispiele für Objekte

- Ein grün lackierter Mini, der Herrn Müller-Lüdenscheid gehört.
- Ein Buch mit dem Titel „Grauen der OOP“ von Erna Etepete,
- Ein Besatzungsmitglied eines Raumschiffs mit dem Namen Spock,
- ...

# Klassen in Java

## Was sind Klassen?

Klassen sind **Baupläne für Objekte**, die versuchen, gemeinsame Eigenschaften bestimmter Objektarten zu modellieren: Sie enthalten unter anderem:

- Klassename,
- Attribute (Felder) → Zustände eines Objektes dieser Klasse
- Methoden → Verhalten eines Objektes dieser Klasse
- Konstruktoren (besondere Methoden zur Initialisierung der Attribute)

# Klassen und Objekte/Instanzen in Java

## Was ist ein Objekt einer Klasse?

Ein **Objekt einer Klasse XY** ist eine **Ausprägung** dieser Klasse, welche anschließend über einen Variablenamen ansprechbar ist. Objekte werden auch als **Instanzen einer Klasse** bezeichnet.

- Objekte werden immer mit `new` erzeugt.
- Der Aufruf von `new` bewirkt einen **Konstruktoraufruf** für das Objekt.
- Mit diesem Konstruktoraufruf wird das Objekt im Freispeicherbereich (Heap) der Java-Virtual-Machine angelegt und initialisiert.

# Vereinbarung von Klassen in Java

## Vereinbarungssyntax – ohne Vererbungsmechanismen

```
1 [modifizierer] class KlassenName {  
2  
3     // Variablen, welche in allen Methoden INNERHALB  
4     // der Klassenvereinbarung sichtbar sind  
5     [Vereinbarung von Attributen]  
6  
7     // Konstruktoren sind spezielle Methoden, welche  
8     // die Attribute mit Werten initialisieren.  
9     [Vereinbarung von Konstruktoren]  
10  
11    [Vereinbarung von Methoden]  
12 }
```

# Vereinbarung von Klassen in Java

## Vereinbarungsbeispiel – ohne Vererbungsmechanismen – Teil 1

```
1 public class Buch1 {  
2     String autor; // Attribut 1  
3     String titel; // Attribut 2  
4  
5     // Standardkonstruktor  
6     // ACHTUNG: Konstruktoren haben KEINEN Rueckgabetyp!  
7     Buch1() {  
8         autor="Hugo Helmchen";  
9         titel="Allein im Weltraum";  
10    } // end constructor  
11    // .. weiter auf der naechsten Seite
```

# Vereinbarung von Klassen in Java

## Vereinbarungsbeispiel – ohne Vererbungsmechanismen – Teil 1

- Hier wird eine Klasse mit 2 Attributen vereinbart
- Der parameterlose Konstruktor wird auch als **Standardkonstruktor** bezeichnet.
- Er setzt die Attribute auf Standard-Werte, die immer gleich sind.
- **Wichtig: Konstruktoren geben nichts zurück – sie haben also keinen Rückgabetyp!**

# Vereinbarung von Klassen in Java

## Vereinbarungsbeispiel – ohne Vererbungsmechanismen – Teil 2

```
12 // ... weiter von Teil 1 ...
13 // Voll qualifizierter Konstruktor
14 Buch1(String autor, String titel) {
15     this.autor = autor;
16     this.titel = titel;
17 } // end constructor
18
19 // Ausgabemethode -- sie gibt die Attributwerte aus!
20 void print() {
21     System.out.println("Titel: " + titel);
22     System.out.println("Autor: " + autor);
23 } // end method
24 } // end class
```

# Vereinbarung von Klassen in Java

## Vereinbarungsbeispiel – ohne Vererbungsmechanismen – Teil 2

- Der Konstruktor **mit Parametern, die alle Attribute belegen** wird auch als **voll qualifizierter Konstruktor** bezeichnet.
- Er setzt die Attribute auf die Werte, die als Parameter übergeben wurden.
- Die Variable **this** ist eine Referenz auf das Objekt, in dem wir uns gerade befinden, wenn der Konstruktor abläuft. Also **dieses** Objekt. (vgl. **self** in Python).
- Würden wir **this** im voll qualifizierten Konstruktor **nicht** verwenden:  
`autor = autor;`  
In diesem Fall würde der Übergabeparameter einfach mit sich selbst belegt werden! (Gültigkeitsbereiche: lokaler Parameter überlagert global!)

# Begriffswiederholung: Klassen und Objekte/Instanzen in Java

Wir wiederholen: Was ist ein Objekt einer Klasse?

Ein **Objekt einer Klasse XY** ist eine **Ausprägung** dieser Klasse, welche anschließend über einen Variablenamen ansprechbar ist. Objekte werden auch als **Instanzen einer Klasse** bezeichnet.

- Objekte werden immer mit `new` erzeugt.
- Der Aufruf von `new` bewirkt einen **Konstruktoraufruf** für das Objekt.
- Mit diesem Konstruktoraufruf wird das Objekt im Freispeicherbereich (Heap) der Java-Virtual-Machine angelegt und initialisiert.

# Erzeugung von Objekten/Instanzen

## Syntax der Objekterzeugung

*// Erzeugung eines Objektes mit Standardkonstruktor*

```
KlassenName variablenName1 = new KlassenName();
```

*// Erzeugung eines Objektes mit einem parametrisierten*

*// Konstruktor*

*// Dies kann beispielsweise ein voll qualifizierter*

*// Konstruktor sein*

```
KlassenName variablenName2 =
```

```
new KlassenName(param1, ..., paramN);
```

# Attribute und Methoden von erzeugten Objekten/Instanzen

## Zugriff auf Attribute und Methoden von Objekten

```
// Zugriff auf ein Attribut von Objekt obj1:  
obj1.attribut1 = wert;
```

```
// Aufruf einer Methode von Objekt obj1:  
obj1.methode();
```

# Beispiel zu Objekterzeugung und Methodenaufruf

## Unsere Buch-Klasse von vorhin – Teil 1

```
public class Buch1 {  
    String autor; // Attribut 1  
    String titel; // Attribut 2  
  
    // Standardkonstruktor  
    // ACHTUNG: Konstruktoren haben KEINEN Rueckgabetyp!  
    Buch1() {  
        autor="Hugo Helmchen";  
        titel="Allein im Weltraum";  
    } // end constructor  
    // .. weiter auf der naechsten Seite
```

# Beispiel zu Objekterzeugung und Methodenaufruf

## Unsere Buch-Klasse von vorhin – Teil 2

```
12 // ... weiter von Teil 1 ...
13 // Voll qualifizierter Konstruktor
14 Buch1(String autor, String titel) {
15     this.autor = autor;
16     this.titel = titel;
17 } // end constructor
18
19 // Ausgabemethode -- sie gibt die Attributwerte aus!
20 void print() {
21     System.out.println("Titel: " + titel);
22     System.out.println("Autor: " + autor);
23 } // end method
24 } // end class
```

# Beispiel zu Objekterzeugung und Methodenaufruf

## Eine main()-Klasse, die zwei Bücher erzeugt – Teil 1

```
1 public class BuchMain1 {  
2  
3     public static void main(String[] args) {  
4  
5         // Objekt b1: Aufruf des voll qualifizierten  
6         // Konstruktors  
7         Buch1 b1 = new Buch1("Gernot Grusel",  
8             "Das Grauen der OOP");  
9  
10        // Objekt b2: Aufruf des Standardkonstruktors  
11        Buch1 b2 = new Buch1();  
12        // ... weiter in Teil 2
```

# Beispiel zu Objekterzeugung und Methodenaufruf

## Eine main()-Klasse, die zwei Bücher erzeugt – Teil 2

```
13 // ... weiter von Teil 1
14 // Methodenaufruf bei b1:
15 b1.print();
16
17 // Methodenaufruf bei b2:
18 b2.print();
19 } // end method main()
20 } // end class
```

# Beispiel zu Objekterzeugung und Methodenaufruf

Compilierung und Start des Buch-Programms mit ...

```
javac Buch1.java
```

```
javac BuchMain1.java
```

```
java BuchMain1
```

... oder innerhalb von Eclipse ...

# Beispiel zu Objekterzeugung und Methodenaufruf

Start des Programms liefert die Ausgabe

Titel: Das Grauen der OOP

Autor: Gernot Grusel

Titel: Allein im Weltraum

Autor: Hugo Helmchen

# Zusammenfassung Klassen und Objekte

## Was sind Klassen?

Klassen sind **Baupläne für Objekte**, die versuchen, gemeinsame Eigenschaften bestimmter Objektarten zu modellieren: Sie enthalten unter anderem:

- Klassename,
- Attribute (Felder) → Zustände eines Objektes dieser Klasse
- Methoden → Verhalten eines Objektes dieser Klasse
- Konstruktoren (besondere Methoden zur Initialisierung der Attribute)

# Zusammenfassung Klassen und Objekte

## Was ist ein Objekt einer Klasse?

Ein **Objekt einer Klasse XY** ist eine **Ausprägung** dieser Klasse, welche anschließend über einen Variablenamen ansprechbar ist. Objekte werden auch als **Instanzen einer Klasse** bezeichnet.

- Objekte werden immer mit `new` erzeugt.
- Der Aufruf von `new` bewirkt einen **Konstruktoraufruf** für das Objekt.
- Mit diesem Konstruktoraufruf wird das Objekt im Freispeicherbereich (Heap) der Java-Virtual-Machine angelegt und initialisiert.

# Zusammenfassung Klassen und Objekte

## Was ist ein Objekt einer Klasse?

- Attribute und Methoden von Objekten/Instanzen werden mit dem „.“-Operator angesprochen:  
`obj.attribut=wert;`  
`obj.methodenaufruf();`
- Objekte existieren nach der Erzeugung auf dem Freispeicher nur **einmal**.
- Die Variable, über die ein Objekt/eine Instanz angesprochen wird, enthält nur eine **Referenz** auf das Objekt – also seine **Adresse**.

# Zusammenfassung Klassen und Objekte

## Was ist ein Objekt einer Klasse?

- Mehrere Referenzen können auf das gleiche Objekt verweisen – beispielsweise mit:

```
Klassenname obj2 = obj;
```

- Hier wird **kein neues Objekt** erzeugt – die Referenz obj2 verweist **auf die gleiche Adresse** wie obj!

## Bisherige Klassenvereinbarungen ...

... mit Attributen und Methoden ohne zusätzliche Angaben

```
public class Buch1 {  
    String autor; // Attribute ohne weitere Angaben ...  
    String titel;
```

```
Buch1() { // Konstruktor ohne weitere Angaben ...  
// ...  
}
```

```
void print() { // Methode ohne weitere Angaben ...  
// ...  
}  
} // end class
```

## Bisherige Klassenvereinbarungen ...

... mit Attributen und Methoden ohne zusätzliche Angaben

Konsequenz aus dem Weglassen zusätzlicher Angaben (**Modifizierer**)

- Attribute eines sind einfach von **jeder Methode jeder Klasse** aus sichtbar, die sich in der gleichen Organisationseinheit (**Package**) befindet, wie das Objekt, dem die Attribute gehören.
- Das gleiche gilt für die Methoden eines Objektes.

# Erzeugung und Verwendung von Objekten der bisherigen Klassen

Zunächst Erzeugung und Ausgabe ...

```
1 public class BuchMain2 {  
2  
3     public static void main(String[] args) {  
4  
5         // Objekt b1: Aufruf des voll qualifizierten  
6         // Konstruktors  
7         Buch1 b1 = new Buch1("Gernot Grusel",  
8             "Das Grauen der OOP");  
9  
10        // Methodenaufruf bei b1: Ausgabe der  
11        // Attribute von b1  
12        b1.print();
```

# Ausgabe nach der Objekterzeugung ist ...

... ähnlich wie vorher:

Titel: Das Grauen der OOP

Autor: Gernot Grusel

# Erzeugung und Verwendung von Objekten der bisherigen Klassen

## ...danach Manipulation eines Attributes

```
13 // Wir ändern den Autor des Buches ...
14 b1.autor = "Hans Hinkel";
15
16 // ... und geben nochmals die Attribute aus:
17 b1.print();
18
19 } // end method main()
20 } // end class
```

## Ausgabe nach der Manipulation ...

... zeigt einen anderen Autor des gleichen Buches:

Titel: Das Grauen der OOP

Autor: Hans Hinkel

**Wir stellen uns derartige Manipulationen in 100000 lines of code vor ... .**

## Prinzip des Information Hiding

... wurde mit der bisherigen Klassenstruktur verletzt

Da die Attribute des Objektes mit der Referenz b1 von außenstehenden Klassen sichtbar sind, können sie von beliebigen Stellen im Code manipuliert werden. Dies kann zu beliebig unlesbaren Programmen und zu bösen Stolperfallen führen.

# Prinzip des Information Hiding

Unter Information Hiding in der OOP verstehen wir ...

- das Verbergen der Daten in den Objektattributen, sowie
- das Verbergen der Implementierung von Methoden.
- z. B. die Attribute eines Objektes sollen nur über den Konstruktor einmal initial gesetzt werden, und danach nicht mehr verändert werden.
- z. B. will ein Bauplan einer Klasse nur einige wenige Methoden nach außen hin sichtbar machen. Kleine Hilfsmethoden, die von diesen genutzt werden, sollen verborgen werden.

# Prinzip des Information Hiding

... in Java umgesetzt durch Modifier (dt. Modifizierer)

- **Modifier** sind Schlüsselworte, mit denen wir das Standardverhalten von Klassen, Attributen und Methoden verändern können.
- Diese Schlüsselworte werden **vor** die Klassen-, Methoden- oder Attributdefinition geschrieben.
- **Modifier für die Sichtbarkeit** verändern das Sichtbarkeitsverhalten von Klassen, Methoden und Attributen.
- Wir werden im Laufe der Vorlesung die drei dieser Modifier **public, private** und **protected** kennenlernen.
- Wir werden auch das Verhalten ohne diese drei Modifier beleuchten.

# Übersicht Sichtbarkeitsmodifizierer

Modifier	Beschreibung
public	Außenstehende Klassen und Objekte und Methoden können public gesetzte Attribute und Methoden von anderen Objekten sehen und voll darauf zugreifen.
private	Außenstehende Klassen, Objekte und Methoden können Attribute und Methoden mit diesem Zugriffsschutz nicht sehen.

# Übersicht Sichtbarkeitsmodifizierer

Modifier	Beschreibung
protected	Außenstehende Objekte können Attribute und Methoden mit diesem Zugriffsschutz nicht sehen. Nur Methoden von Kindklassen oder von Klassen im gleichen Package (kommt beides später) können auf Attribute und Methoden mit diesem Zugriffsschutz zugreifen.
<b>kein Sichtbarkeitsmodifizierer</b>	Nur Methoden von Klassen im gleichen Package (kommt später) können auf Attribute und Methoden ohne Sichtbarkeitsmodifizierer zugreifen.

# Beispiel zu den Modifizierern public und private

## Modifizierte Buch-Klasse – Teil 1

```
1 public class Buch2 {  
2     private String autor;  
3     private String titel;  
4     private int auflage;
```

- Ein weiteres Attribut (auflage) ist hinzugekommen.
- Alle drei Attribute sind private

# Beispiel zu den Modifizierern public und private

## Modifizierte Buch-Klasse – Teil 2

```
5   public Buch2() {  
6       this.autor="Hugo Helmchen";  
7       this.titel="Allein im Weltraum";  
8       this.auflage=1;  
9   } // end constructor  
10  
11  public Buch2(String autor, String titel, int auflage) {  
12      this.autor = autor;  
13      this.titel = titel;  
14      this.auflage = auflage;  
15  } // end constructor
```

- Beide Konstruktoren sind public
- Sie belegen nun alle drei Attribute mit Werten.

# Beispiel zu den Modifizierern public und private

## Modifizierte Buch-Klasse – Teil 3

```
16 // get-Methoden fuer Autor und Titel
17 public String getAutor() {
18     return autor;
19 } // end method
20
21 public String getTitle() {
22     return titel;
23 } // end method
```

- Es gibt nun zwei public-Methoden zum Abfragen von Autor und Titel
- Wir erinnern uns: Die Attribute, die hier mit return zurückgegeben werden, sind private!
- Wir können mit diesen Methoden nur Werte abfragen – **nicht ändern!**

## Beispiel zu den Modifizierern public und private

### Modifizierte Buch-Klasse – Teil 4

```
24   public int getAuflage() {  
25       return auflage;  
26   } // end method  
27  
28   public void setAuflage(int auflage) {  
29       this.auflage = auflage;  
30   } // end method  
31 } // end class
```

- Für das Attribut Auflage: public **getter()-Methode** und **setter()-Methode**
- Wir können mit der setter()-Methode das Attribut auflage **nach der Objekterzeugung verändern!**

# Beispiel zu den Modifizierern public und private

## Modifizierte main()-Klasse Teil 1

```
1 public class BuchMain2Modified {  
2  
3     public static void main(String[] args) {  
4  
5         Buch2 b = new Buch2("G. Grusel", "Grauen der OOP", 5);  
6  
7         // Attribute abfragen  
8         String aut = b.getAutor();  
9         String ti = b.getTitle();  
10        int aufl = b.getAuflage();
```

→getter() liefern Attribute zurück

# Beispiel zu den Modifizierern public und private

## Modifizierte main()-Klasse Teil 1

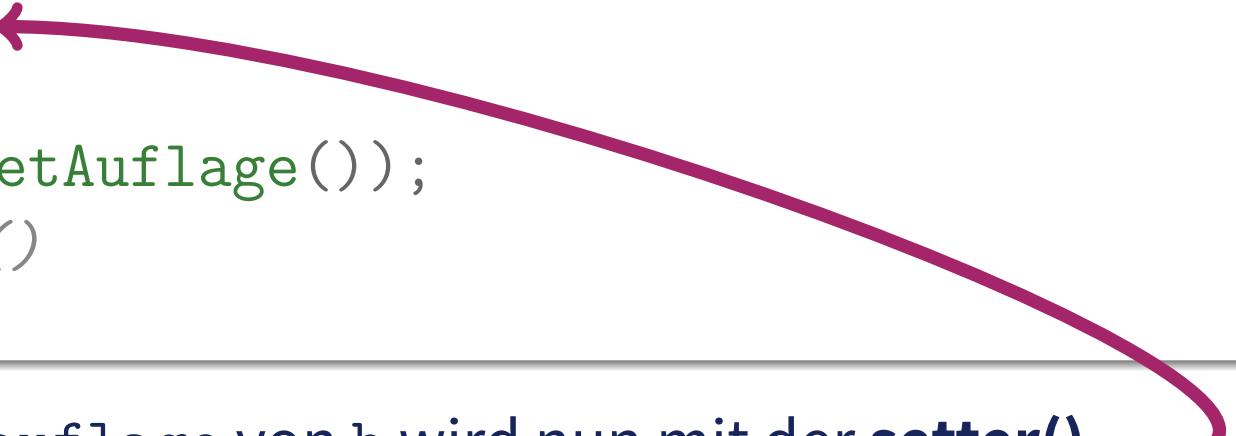
```
11 // Abgefragte Attribut-Werte mit ihren aktuellen  
12 // Werten ausgeben  
13 System.out.println("Autor: " + aut);  
14 System.out.println("Titel: " + ti);  
15 System.out.println("Auflage: " + aufl);
```

- Die mit den **getter()**-Methoden abgefragten Attribut-Werte werden in lokalen Variablen in `main()` abgelegt.
- Die Werte dieser lokalen Variablen geben wir auf die Konsole aus.

# Beispiel zu den Modifizierern public und private

## Modifizierte main()-Klasse Teil 1

```
17 // Wert von aufl aendern und neu einsetzen:  
18 // Buch liegt nun in der 6. Auflage vor  
19 aufl ++;  
20 b.setAuflage(aufl);  
21 System.out.println(  
22     "Auflage: " + b.getAuflage());  
23 } // end method main()  
24 } // end class
```



- Der private Attributwert `auflage` von `b` wird nun mit der **setter()**-Methode auf den Wert 6 gesetzt.
- Anschließend wird er mit der **getter()**-Methode für die Konsol-Ausgabe wieder abgefragt.

# Zusammenfassung zu Sichtbarkeitsmodifizierern

## Sichtbarkeitsmodifizierer und das Prinzip des Information Hiding

- Sichtbarkeitsmodifizierer können auf **Methoden** und **Attribute** innerhalb von Klassen angewendet werden:
  - **private** bedeutet: Das Attribut / Die Methode ist von Methoden anderer Klassen aus **nicht sichtbar**.
  - Es ist also nicht möglich, mit `obj.variable=wert`; oder mit `obj.methodenaufruf()`; auf sie zuzugreifen.
  - **public** bedeutet: Das Attribut / Die Methode ist von **überall aus** sichtbar!
- Daneben können Sichtbarkeitsmodifizierer auch auf Klassendefinitionen angewendet werden.
- So bedeutet z.B. `public class Foo { ... }`: Die Klasse Foo ist auch in **jedem anderen Package** außerhalb des eigenen Packages sichtbar.  
(Packages kommen später ...)

## Bisher behandelt: Instanzvariablen/Instanzmethoden

Für Instanzvariablen und Instanzmethoden gilt:

- Variablen werden bei jeder neu erzeugten Instanz einer Klasse separat aufgebaut.
- Methoden u. U. auch. Das bedeutet: Jedes Objekt einer Klasse XY hat seine **eigenen** Instanzvariablen und Instanzmethoden.
- Um Methoden aufrufen zu können, mussten wir ein **Arbeitsobjekt** bzw. eine **Arbeitsinstanz** einrichten.
- Das gleiche galt für den Variablenzugriff.

## Bisher behandelt: Instanzvariablen/Instanzmethoden

Für Instanzvariablen und Instanzmethoden gilt:

- Daher werden derartige Variablen und Methoden als **Instanzvariablen** bzw. **Instanzmethoden** bezeichnet.
- Als Synonym werden auch die Begriffe **Objektvariablen** bzw. **Objektmethoden** verwendet.

# Beispiel zu Instanzvariablen

## Klasse Person1

```
1 public class Person1 {  
2     private String name;  
3  
4     public Person1(String name) {  
5         this.name = name;  
6     } // end constructor  
7  
8     public String getName() {  
9         return name;  
10    } // end method  
11 } // end class
```



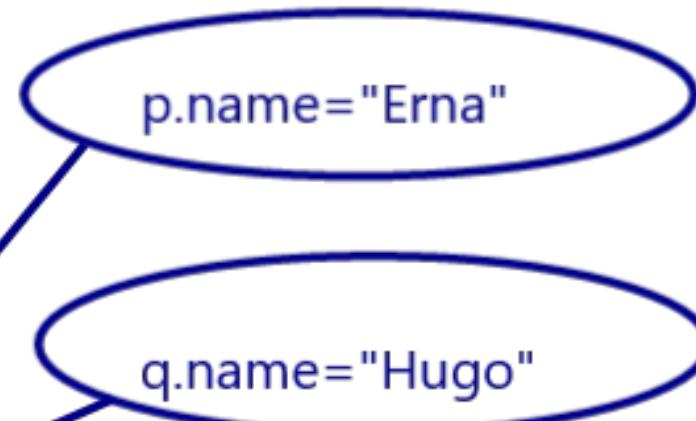
Instanzvariable

# Beispiel zu Instanzvariablen

Klasse PersonMain1 erzeugt zwei Instanzen

```
public class PersonMain1 {  
    public static void main() {  
  
        Person1 p = new Person1("Erna");  
  
        Person1 q = new Person1("Hugo");  
    }  
}
```

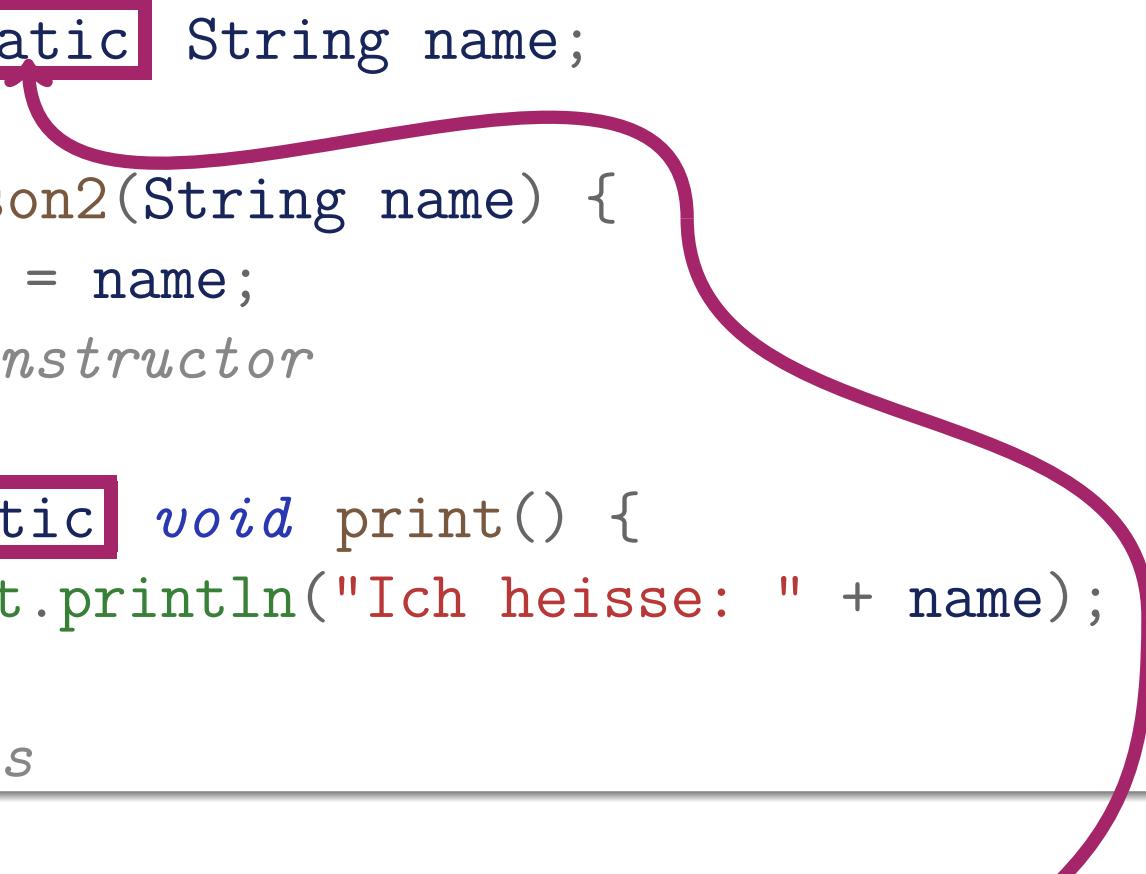
Jede der beiden Instanzen hat  
ihre eigene name-Variable!



# Einführung von Klassenvariablen und Klassenmethoden

## Klasse Person2 mit Modifizierer static

```
1 public class Person2 {  
2     private static String name;  
3  
4     public Person2(String name) {  
5         this.name = name;  
6     } // end constructor  
7  
8     public static void print() {  
9         System.out.println("Ich heisse: " + name);  
10    }  
11 } // end class
```

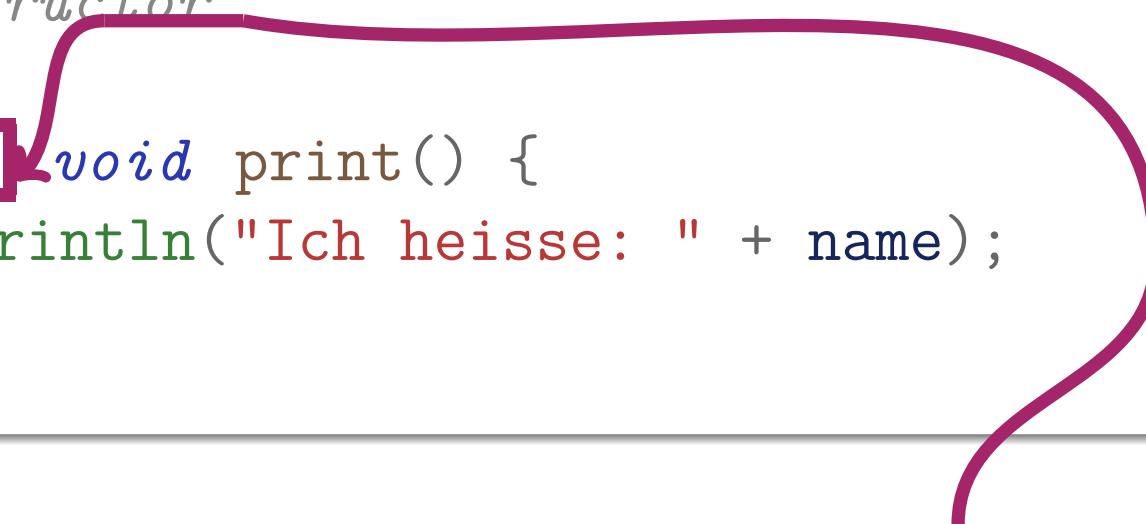


Klassenvariable

# Einführung von Klassenvariablen und Klassenmethoden

## Klasse Person2 mit Modifizierer static

```
1 public class Person2 {  
2     private static String name;  
3  
4     public Person2(String name) {  
5         this.name = name;  
6     } // end constructor  
7  
8     public static void print() {  
9         System.out.println("Ich heisse: " + name);  
10    }  
11 } // end class
```

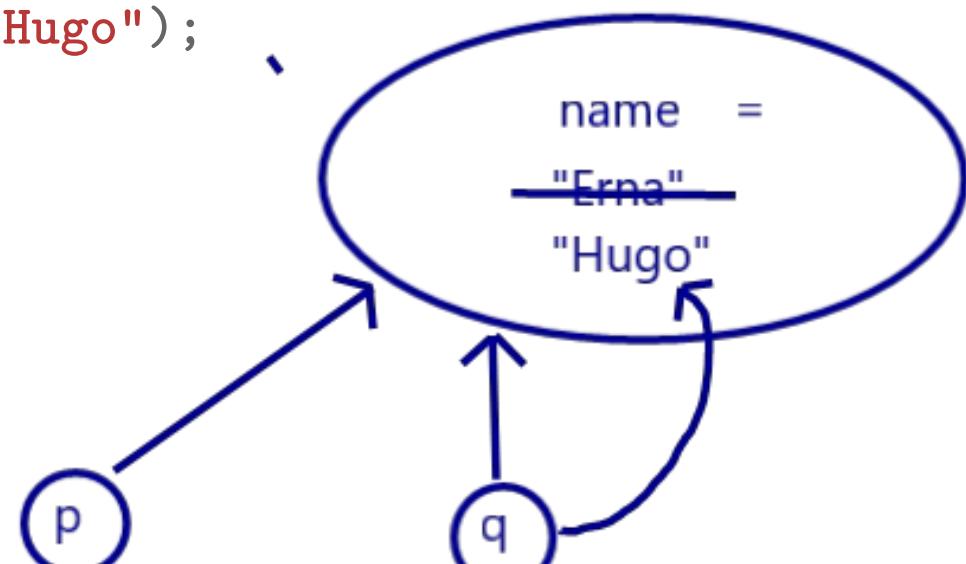


Klassenmethode

# Einführung von Klassenvariablen und Klassenmethoden

Klasse PersonMain2 erzeugt ebenfalls 2 Instanzen, aber:

```
public class PersonMain2 {  
    public static void main(String [] args) {  
  
        Person2 p = new Person2("Erna");  
        Person2 q = new Person2("Hugo");  
  
        p.print();  
        q.print();  
    } // end method main()  
} // end class
```



Beide Instanzen p und q teilen sich die Klassenvariable.  
name wird von Instanz q mit "Hugo" überschrieben

# Einführung von Klassenvariablen und Klassenmethoden

Klasse PersonMain2 erzeugt ebenfalls 2 Instanzen, aber:

- Variablen mit dem Modifizierer `static` existieren **nur einmal**, gleichgültig, wie viele Instanzen der Klasse erzeugt werden!
- Bei der Benutzung von **statischen** Variablen und Methoden kann auf die **Erstellung einer Arbeitsinstanz verzichtet werden!**
- Statt dessen können wir **statische** Variablen und Methoden direkt über den Klassennamen ansprechen →**Klassenvariablen** bzw. **Klassenmethoden!**

# Einführung von Klassenvariablen und Klassenmethoden

Klassenvariablen und Klassenmethoden können ohne Instanz angesprochen werden

```
1  public class Person3 {  
2      public static String name;  
3  
4      public Person3(String name) {  
5          this.name = name;  
6      } // end constructor  
7  
8      public static void print() {  
9          System.out.println("Ich heisse: " + name);  
10     }  
11  } // end class
```

Zu Testzwecken hier public

# Einführung von Klassenvariablen und Klassenmethoden

Klassenvariablen und Klassenmethoden können ohne Instanz angesprochen werden

```
public class PersonMain3 {  
    public static void main(String [] args) {  
  
        Person3.name = "Hugo";  
        Person3.print();  
    } // end method main()  
} // end class
```

Klassenvariable und ...  
Klassenmethode

} können direkt über den Namen der Klasse angesprochen werden!



Es müssen keine Arbeitsobjekte bzw. Arbeitsinstanzen mehr erstellt werden



# Einführung von Klassenvariablen und Klassenmethoden

Wo machen statische Variablen/Methoden denn nun Sinn?

- Für allgemeine Konstanten (final), die dann mit *Klasse.konstante* verwendet werden.
- Für Methoden, die sich zwischen ihren Aufrufen keine Zwischenzustände merken müssen.

# Einführung von Klassenvariablen und Klassenmethoden

## Sinnvolles Beispiel für static

```
1 // Klasse zur Umrechnung Kelvin --> Celsius
2 public class StaticOk {
3     // Konstante: Absoluter Nullpunkt
4     static final double kelvinNull=-273.15;
5
6     // Methode zur Umrechnung Kelvin --> Celsius
7     static double gradCelsius(double kelvin) {
8         double celsius = kelvin + kelvinNull;
9         return celsius;
10    } // end method
11 } // end class
```

**Statische Konstante**

# Einführung von Klassenvariablen und Klassenmethoden

## Sinnvolles Beispiel für static

```
1 // Klasse zur Umrechnung Kelvin --> Celsius
2 public class StaticOk {
3     // Konstante: Absoluter Nullpunkt
4     static final double kelvinNull=-273.15;
5
6     // Methode zur Umrechnung Kelvin --> Celsius
7     static double gradCelsius(double kelvin) {
8         double celsius = kelvin + kelvinNull;
9         return celsius;
10    } // end method
11 } // end class
```

**Statische Methode:** Sie muss sich keine „Objektzustände“ merken – wird einfach immer wieder bei Bedarf mit neuer °K-Zahl aufgerufen.

# Einführung von Klassenvariablen und Klassenmethoden

Sinnvolles Beispiel für static – Anwendung in *main()*

```
1 import java.util.Locale;  
2 // ... Ausschnitt aus main()-Methode ...  
3 double k=245.0;  
4 Locale.setDefault(Locale.ENGLISH); // Punkt statt Komma  
5 System.out.println(  
6   "Abs. Nullpunkt Kelvin: " + Static0k.kelvinNull);  
7  
8 double c = Static0k.gradCelsius(k);  
9 // %.2f heisst: Ausgabe als float oder double mit  
10 // 2 Nachkomma-Stellen  
11 System.out.printf(  
12   "%.2f Grad Kelvin entspricht %.2f Grad Celsius", k,c);
```

# Einführung von Klassenvariablen und Klassenmethoden

## Zusammenfassung Klassenvariablen und Klassenmethoden

- **Statische Variablen** werden auch als **Klassenvariablen** bezeichnet, da der Zugriff auf sie nur an den **Klassennamen**, nicht aber an ein Objekt gebunden ist.
- Dementsprechend werden **Statische Methoden** auch als **Klassenmethoden** bezeichnet.
- **Nicht-Statische Variablen** werden auch als **Instanzvariablen** bzw. **Objektvariablen** bezeichnet, da der Zugriff auf sie immer an **ein Objekt/eine Instanz** gebunden ist.
- Dementsprechend werden **Nicht-Statische Methoden** auch als **Instanzmethoden** bzw. **Objektmethoden** bezeichnet.

## Bereits kennengelernt: Arrays elementarer Datentypen

### Syntax zur Vereinbarung von Arrays

// Möglichkeit 1:

```
datentyp [] arrayname = new datentyp [anzahl];
```

// Möglichkeit 2:

```
datentyp [] arrayname2 = {element1, element2, ..., elementN};
```

→ Arrays haben, nachdem sie einmal angelegt wurden, eine feste Größe!

# Neu: Arrays aus Objekten/Instanzen

## Arbeitsschritte zur Erzeugung von Arrays aus Objekten

- Erzeugung der Referenz für das Array selbst
- Erzeugung der Instanzen
- Einhängen der Instanzen ins Array

# Beispiel zu Arrays aus Objekten/Instanzen

Unsere Personenklasse wird zur Objekterzeugung verwendet

```
1 public class Person1 {  
2     private String name;  
3  
4     public Person1(String name) {  
5         this.name = name;  
6     } // end constructor  
7  
8     public String getName() {  
9         return name;  
10    } // end method  
11 } // end class
```

# Beispiel zu Arrays aus Objekten/Instanzen

## main()-Klasse zur Objekterzeugung

```
1 import java.util.Scanner;  
2  
3 public class PersonMainArray1 {  
4  
5     public static void main(String[] args) {  
6  
7         // Scanner zum Eintippen der Personen-Namen  
8         Scanner tastatur = new Scanner(System.in);  
9  
10        // Schritt 1: Erzeugung der Array-Referenz  
11        // Wir wollen Platz fuer 3 Personen  
12        Person1 [] personen = new Person1[3];
```

# Beispiel zu Arrays aus Objekten/Instanzen

## main()-Klasse zur Objekterzeugung

```
13 // Schleife fuer die Belegung des Arrays
14 for (int i=0; i < personen.length; i++) {
15     System.out.print("Name der naechsten Person: ");
16     String name = tastatur.nextLine();
17
18     // Schritt 2: Erzeugung der Instanz
19     Person1 p = new Person1(name);
20
21     // Schritt 3: Einhaengen ins Array
22     personen[i] = p;
23 } // end for
```

# Beispiel zu Arrays aus Objekten/Instanzen

## main()-Klasse zur Objekterzeugung

```
24 // Testen des Arrays: Ausgabe aller Personennamen
25 System.out.println("----- Wir haben eingegeben: -----");
26 // Schleife fuer die Ausgabe des Arrays
27 for (int i=0; i < personen.length; i++) {
28     String name = personen[i].getName();
29     System.out.println(name);
30 } // end for
31 } // end method main()
32 } // end class
```

# Listen aus Objekten/Instanzen

## Beispiel *ArrayList*

- Unterschied zu Arrays: Listen können wachsen / schrumpfen
- Unterschied zu Arrays: Listen können nur Referenzvariablen aufnehmen, keine Variablen elementaren Typs!
- Listen können entweder selbst implementiert werden ...
- ... oder wir nutzen eine der vorgefertigten Listenklassen von Java: z. B. die Klasse *ArrayList*.

# Listen aus Objekten/Instanzen

## Syntax beim Anlegen einer *ArrayList*

```
1 ArrayList <Typ> variablenname = new ArrayList<Typ>();
```

Objekttyp der Liste

Konstruktorauftrag

# Listen aus Objekten/Instanzen

Syntax beim Anlegen einer *ArrayList* – Liste aus Personen

```
1 ArrayList <Person1> variablenname = new ArrayList<Person1>();
```

Objekttyp der Liste

Konstruktoraufruf

# Beispiel zu Listen aus Objekten/Instanzen

main()-Klasse zur Objekterzeugung

```
1 import java.util.Scanner;  
2 import java.util.ArrayList;  
3  
4 public class PersonMainListe1 {  
5  
6     public static void main(String[] args) {  
7  
8         // Wir wollen 3 Personen eintippen  
9         int anzahl = 3;  
10        // Scanner zum Eintippen der Personen-Namen  
11        Scanner tastatur = new Scanner(System.in);
```

# Beispiel zu Listen aus Objekten/Instanzen

## main()-Klasse zur Objekterzeugung

```
12 // Schritt 1: Erzeugung der Listen-Referenz  
13 ArrayList<Person1> personen = new ArrayList<Person1>(); S
```

# Beispiel zu Listen aus Objekten/Instanzen

## main()-Klasse zur Objekterzeugung

```
14 // Schleife fuer die Belegung der Liste
15 for (int i=0; i<anzahl; i++) {
16     System.out.print("Name der naechsten Person: ");
17     String name = tastatur.nextLine();
18
19     // Schritt 2: Erzeugung der Personen-Instanz
20     Person1 p = new Person1(name);
21
22     // Schritt 3: Einhaengen in die Liste
23     personen.add(p);
24 } // end for
```

# Beispiel zu Listen aus Objekten/Instanzen

## Ausgabe mit Zählschleife for

### main()-Klasse zur Objekterzeugung

```
25 // Testen der Liste: Ausgabe aller Personennamen
26 System.out.println("----- Wir haben eingegeben: -----");
27
28 // Schleife fuer die Ausgabe der Liste
29 for (int i=0; i < personen.size(); i++) {
30     Person1 p = personen.get(i);
31     String name = p.getName();
32     System.out.println(name);
33 } // end for
34 } // end method main()
35 } // end class
```

# Beispiel zu Listen aus Objekten/Instanzen

## Alternative Ausgabe mit Iteratorschleife for

### main()-Klasse zur Objekterzeugung

```
25 // Testen der Liste: Ausgabe aller Personennamen
26 System.out.println("----- Wir haben eingegeben: -----");
27
28 // foreach-Schleife
29 for (Person1 p : personen) {
30
31     String name = p.getName();
32     System.out.println(name);
33 } // end for
34 } // end method main()
35 } // end class
```

Es wird **kein expliziter get()-Aufruf der Listenvariable personen mehr benötigt!**

Fakultät Informatik

# Programmierung 2

Die Klasse String

Vorwort

Einstieg in Java

Einstieg in die OOP: Klassen und Objekte/Instanzen

Die Klasse String

Grundlagen Strings

Stringoperationen

Wichtigste Methoden der Klasse String

Eigenschaften von Strings abfragen

Graphische Darstellung von Klassen mit der UML – Teil 1



# Vererbungsbeziehungen

## Exceptions

## Einstieg Streams

# Grundlagen Zeichen und Strings

## Der Datentyp **char**

- Der Datentyp **char** ist in Java ein 16-Bit-Datentyp.
- In **String**-Objekten werden Zeichenfolgen, bestehend aus Unicode-Zeichen gekapselt.
- Die Klasse **String** stellt eine Reihe von String-Verarbeitungs- Methoden zur Verfügung, z.B.
  - Ausschneiden von Teilstrings,
  - Suchen nach Teil-Zeichenketten,
  - Splitten nach „Trenner“, uvm.
  - Siehe hierzu auch die offizielle API-Seite  
<https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>

# Grundlagen Zeichen und Strings

## Variablen vom Typ **char** und vom Typ **String**

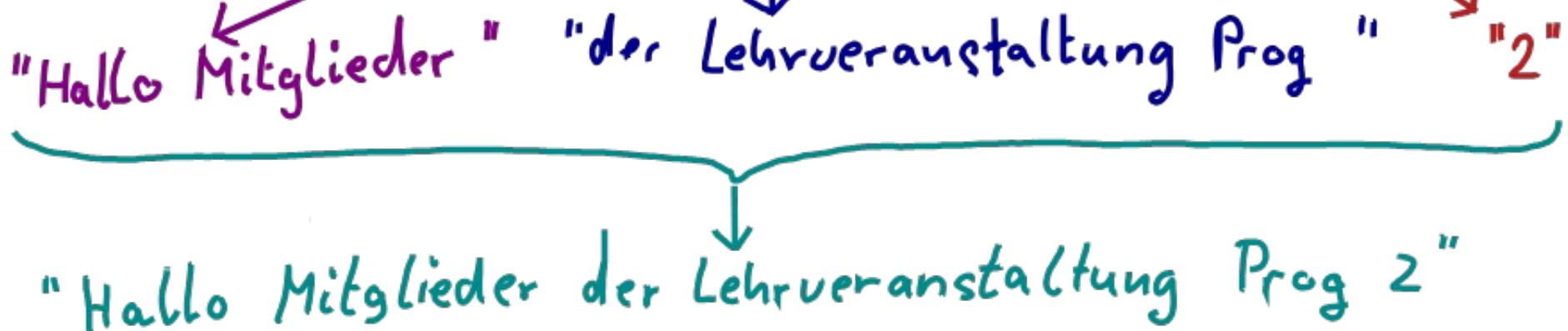
```
1 char x = 'M'; // einfaches Zeichen
2
3 // String mit Zuweisung des Literals
4 String s = "Erna Etepete";
5
6 // String mit Konstruktor
7 String t = new String("Hugo Helmchen")
```

- Die Literale einfacher Zeichen werden mit **einfachen Anführungsstrichen** 'M' vereinbart!
- Die Literale von Strings werden mit **doppelten Anführungsstrichen** "abc" vereinbart!

# Stringoperationen

## Verkettung (Konkatenation) von Strings mit +

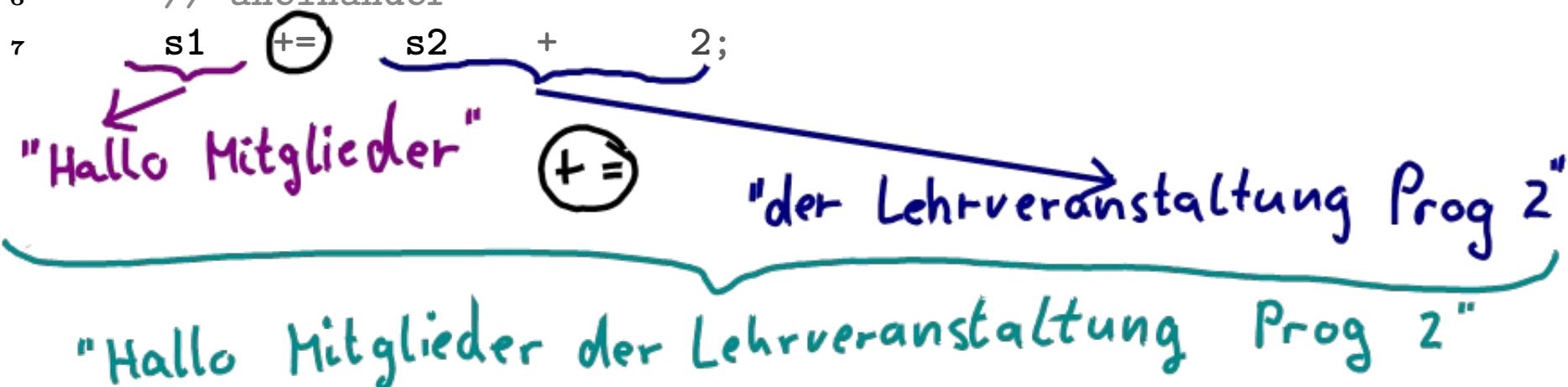
```
1 String s1 = "Hallo Mitglieder ";
2 String s2 = "der Lehrveranstaltung Prog ";
3
4 // Konkatenierungsoperator:
5 // Haengt s1 und s2 und die Zahl 2 aneinander
6 String s3 = s1 + s2 + 2;
```



# Stringoperationen

## Verkettung (Konkatenation) von Strings mit +=

```
1  String s1 = "Hallo Mitglieder ";
2  String s2 = "der Lehrveranstaltung Prog ";
3
4  // Konkatenierungsoperator +=: Haengt an s1
5  // die Strings s1, gefolgt von der Ziffer 2
6  // aneinander
7  s1 += s2 + 2;
```



# Stringoperationen

## Vergleich von Strings – erster Versuch mit ==

```
1 String s = "Erna";
2 String t = "Erna";
3
4 if (s == t) {
5     System.out.println("gleich!");
6 }
7 else {
8     System.out.println("ungleich!");
9 } // end else
```

Ausgabe lautet:

gleich!

→ Die Strings s und t scheinen gleich zu sein!

# Stringoperationen

## Vergleich von Strings – zweiter Versuch mit ==

```
1 String s = "Erna";  
2 String t = new String("Erna");  
3  
4 if (s == t) {  
5     System.out.println("gleich!");  
6 }  
7 else {  
8     System.out.println("ungleich!");  
9 } // end else
```

Ausgabe lautet nun:

ungleich!

→ Die Strings s und t scheinen trotz gleichem Inhalt **nicht gleich** zu sein!

# Stringoperationen

## Vergleich von Strings – Erklärung des ersten Versuchs mit ==

```
1 String s = "Erna"; → Erna
2 String t = "Erna";
3
4 if (s == t) {
5 // ...
6 }
```

### Erklärung:

- Die Variablen **s** und **t** enthalten eine Referenz auf die gleiche Speicherstelle!
- Das **Objekt** mit dem Inhalt "**Erna**" existiert **nur einmal**!
- Bei der zweiten Zuweisung an **t** wird kein neues Objekt mehr erzeugt.
- Der **==**-Operator vergleicht hier nur zwei (gleiche) Adressen.

# Stringoperationen

## Vergleich von Strings – Erklärung des zweiten Versuchs mit ==

```
1 String s = "Erna";
```



```
2
```

```
3 String t = new String("Erna");
```



```
4
```

```
5 if (s == t) // ...
```

### Erklärung:

- Die Variablen **s** und **t** enthalten Referenzen auf **verschiedene Speicherstellen**
- Es existieren durch die **new**-Operation **zwei Objekte mit gleichartigem Inhalt!**
- Der **==**-Operator vergleicht hier zwei **unterschiedliche Adressen**.
- Daher sind die Variablen **s** und **t ungleich!**

# Stringoperationen

Wir merken uns: Stringvergleiche mit ==

- ... sind keine gute Idee!
- Der ==-Operator vergleicht **nur die Adressen** der Strings, **nicht aber die Inhalte!**
- Ebenso führt der !=-Operator nur einen Vergleich der **Adressen** durch, **nicht aber die Inhalte!**

# Stringoperationen

Wie machen wir aber nun Inhaltsvergleiche mit Strings?!!!

- Möglichkeit 1: Verwendung der `equals()`-Methode der Klasse `String`.
- Möglichkeit 2: Verwendung von `switch-case` , sofern ein String mit konstanten Literalen verglichen werden soll.

# Stringoperationen

## Stringvergleich mit equals()

```
1 String s = "Erna"; → Erna
2 String t = new String("Erna"); → Erna
3
4 if (s.equals(t)) {
5     System.out.println("gleich!");
6 }
7 else {
8     System.out.println("ungleich!");
9 } // end else
```

Ausgabe lautet: **gleich!**

- Die Strings **s** und **t** haben den **gleichen Inhalt!**
- Daher gibt **s.equals(t)** den Wert **true** zurück!

# Stringoperationen

Auch der folgende Vergleich mit equals() funktioniert:

```
1 String t = new String("Erna"); → Erna
2
3 if (t.equals("Erna")) {
4     System.out.println("gleich!");
5 }
6 else {
7     System.out.println("ungleich!");
8 } // end else
```

Ausgabe lautet auch hier: **gleich!**



- Der Inhalt von t wird direkt mit dem **Inhalt eines String-Literals** verglichen.
- Auch hier ist der Inhalt gleich → equals() gibt true zurück.

# Stringoperationen

## Stringvergleich mit switch-case

```
1 String t = new String("Erna");  
2  
3 switch (t) {  
4 case "Emil": System.out.println("Hallo Emil!");  
5                         break;  
6 case "Erna": System.out.println("Hallo Erna!");  
7                         break;  
8 case "Hugo": System.out.println("Hallo Hugo!");  
9                         break;  
10 default: System.out.println("nichts davon!");  
11 } // end switch
```

Ausgabe lautet hier: Hallo Erna!

# Stringoperationen

## Stringvergleich mit switch-case

Seit Java 7 gilt:

Der Stringvergleich mit switch-case ist äquivalent zum Vergleich mit Hilfe der equals()-Methode!

In den cases werden jeweils **String-Literale** abgefragt.

# Wichtigste Methoden der Klasse String

## Eigenschaften von Strings abfragen

Rückgabe-type	Methode
char	<code>charAt(int pos)</code> gibt das Zeichen an Position <code>pos</code> zurück
boolean	<code>endsWith(String substr)</code> prüft, ob der String mit dem übergebenen <code>substr</code> endet.
boolean	<code>isEmpty()</code> prüft, ob der String leer ist.
int	<code>length()</code> gibt die Länge des Strings zurück.
boolean	<code>startsWith(String substr)</code> prüft, ob der String mit dem übergebenen <code>substr</code> beginnt.

# Methoden der Klasse String

Eigenschaften von Strings abfragen – Position eines Zeichens

```
1 String s1 = new String("Hallo");
2
3 // gibt 'a' zurueck
4 char c = s1.charAt(1);
```

# Methoden der Klasse String

## Eigenschaften von Strings abfragen – Länge

```
1 String s1 = new String("Hallo");
2 int [] arr = {44,-55,12,8};
3
4 // gibt 5 zurueck
5 int laengeString = s1.length();
6
7 // gibt 4 zurueck
8 int laengeArray = arr.length;
```

### Achtung:

Die Länge eines Strings wird mit seiner **Methode length()** abgefragt.

Die Länge eines Arrays wird mit seiner **Variablen length** abgefragt.

# Wichtigste Methoden der Klasse String

## Substrings und deren Indizes

Rückgabe- typ	Methode
int	<b>indexOf(char c)</b> gibt die Position des erstmaligen Vorkommens von c zurück. -1 bei Nichtvorkommen.
int	<b>indexOf(String s)</b> gibt die Position des erstmaligen Vorkommens von s zurück. -1 bei Nichtvorkommen.
int	<b>indexOf(char c, int pos)</b> gibt die Position des erstmaligen Vorkommens von c ab Position pos zurück. -1 bei Nichtvorkommen.

# Wichtigste Methoden der Klasse String

## Substrings und deren Indizes

Rückgabeytyp	Methode
int	<code>indexOf(String s, int pos)</code> gibt die Position des erstmaligen Vorkommens von <code>s</code> ab Position <code>pos</code> zurück. -1 bei Nichtvorkommen.
String	<code>substring(int pos1)</code> gibt den Teilstring ab Position <code>pos1</code> zurück. Leerer String, falls die Position nicht gültig ist. In diesem Falle wirft die Methode außerdem eine <code>IndexOutOfBoundsException</code>

# Wichtigste Methoden der Klasse String

## Substrings und deren Indizes

Rückgabe- typ	Methode
String	<b>substring(int pos1, int pos2)</b> gibt den Teilstring ab Position pos1 bis vor Position pos2 zurück. Leerer String, falls die Position nicht gültig ist. In diesem Falle wirft die Methode außerdem eine <b>IndexOutOfBoundsException</b>

# Methoden der Klasse String

## Indizes von Strings abfragen – Beispiel 1

```
1 String s1 = "Beim Flachdach ist das Dach flach";  
2  
3 // 'A' kommt nicht vor --> pos1 = -1  
4 int pos1 = s1.indexOf('A');
```

# Methoden der Klasse String

## Indizes von Strings abfragen – Beispiel 2

```
1 String s1 = "Beim Flachdach ist das Dach flach";  
2  
3 // Ab Position 5 (von 0 an gezaehlt) beginnt das  
4 // erste Vorkommen von "dach" ab Position 10  
5 // --> pos2 = 10  
6 int pos2 = s1.indexOf("dach", 5);
```

# Methoden der Klasse String

## Substrings abfragen – Beispiel 1

```
1 String s1 = "Beim Flachdach ist das Dach flach";  
2  
3 // liefert den Teilstring zwischen Position  
4 // 2 und vor Position 6: --> "im F"  
5 String s3 = s1.substring(2,6);
```

# Methoden der Klasse String

## Konvertierung von String – Elementare Datentypen

- Zur Konvertierung **Elementarer Datentyp** → **String**: Hier gibt es die `valueOf()`-Methoden der Klasse String.
- Zur Konvertierung **String** → **Elementarer Datentyp**: Hier gibt es die `parse...()`-Methoden der **Wrapper-Klassen**

- `Short`
- `Integer`
- `Long`
- `Float`
- `Double`
- `Boolean`

# Methoden der Klasse String

## Beispiel 1: Konvertierung Elementarer Datentyp → String

```
1 int iWert = 12;  
2 double dWert = 1.5;  
3  
4 // iString enthaelt anschliessend die Zeichenfolge "12"  
5 String iString = String.valueOf(iWert);  
6  
7 // dString enthaelt anschliessend die Zeichenfolge "1.5"  
8 String dString = String.valueOf(dWert);
```

### Achtung:

Die valueOf...()-Methoden der Klasse String sind statisch. Daher lassen sie sich direkt über den Klassennamen – **ohne** Arbeitsobjekt aufrufen.

# Methoden der Klasse String

Beispiel 2: Konvertierung String → Elementarer Datentyp mit Wrapper-Klassen

```
1 String iString = "345";
2 String dString = new String("22.5");
3
4 // iWertNeu enthaelt nun den Wert 345
5 int iWertNeu = Integer.parseInt(iString);
6
7 // dWertNeu enthaelt nun den Wert 22.5
8 double dWertNeu = Double.parseDouble(dString);
```

## Achtung:

Die parse...()-Methoden der Wrapper-Klassen sind statisch. Daher lassen sie sich direkt über den Klassennamen – **ohne** Arbeitsobjekt aufrufen.

Fakultät Informatik

# Programmierung 2

UML-Klassendiagramme – Teil 1

## Vorwort

## Einstieg in Java

## Einstieg in die OOP: Klassen und Objekte/Instanzen

## Die Klasse String

# Graphische Darstellung von Klassen mit der UML – Teil 1

## Einfache Klassendarstellungen

## Klassenbeziehungen – Teil 1

Assoziation

Aggregation und Komposition



---

# Vererbungsbeziehungen

## Exceptions

## Einstieg Streams

# Bisher behandelt: Klassendarstellung direkt in Java

Bisher: Klassendarstellung in Java mit ...

- Klassenname und ggf. Modifier,
- Attributen und deren Modifier,
- Konstruktoren und sonstigen Methoden und deren Modifier
- Bei Konstruktoren und Methoden ist auch sofort die Implementierung sichtbar
- **→wird schnell unübersichtlich!**

# Bisher behandelt: Klassendarstellung direkt in Java

## Bisher: Buch-Klasse in Java – Teil 1

```
1 public class Buch1 {  
2     private String autor;  
3     private String titel;  
4     private int auflage;  
5  
6     public Buch1(String autor, String titel, int auflage) {  
7         this.autor = autor;  
8         this.titel = titel;  
9         this.auflage = auflage;  
10    } // end constructor
```

# Bisher behandelt: Klassendarstellung direkt in Java

## Bisher: Buch-Klasse in Java – Teil 2

```
10  public String getAutor () {  
11      return autor;  
12  } // end method  
13  
14  public String getTitel() {  
15      return titel;  
16  } // end
```

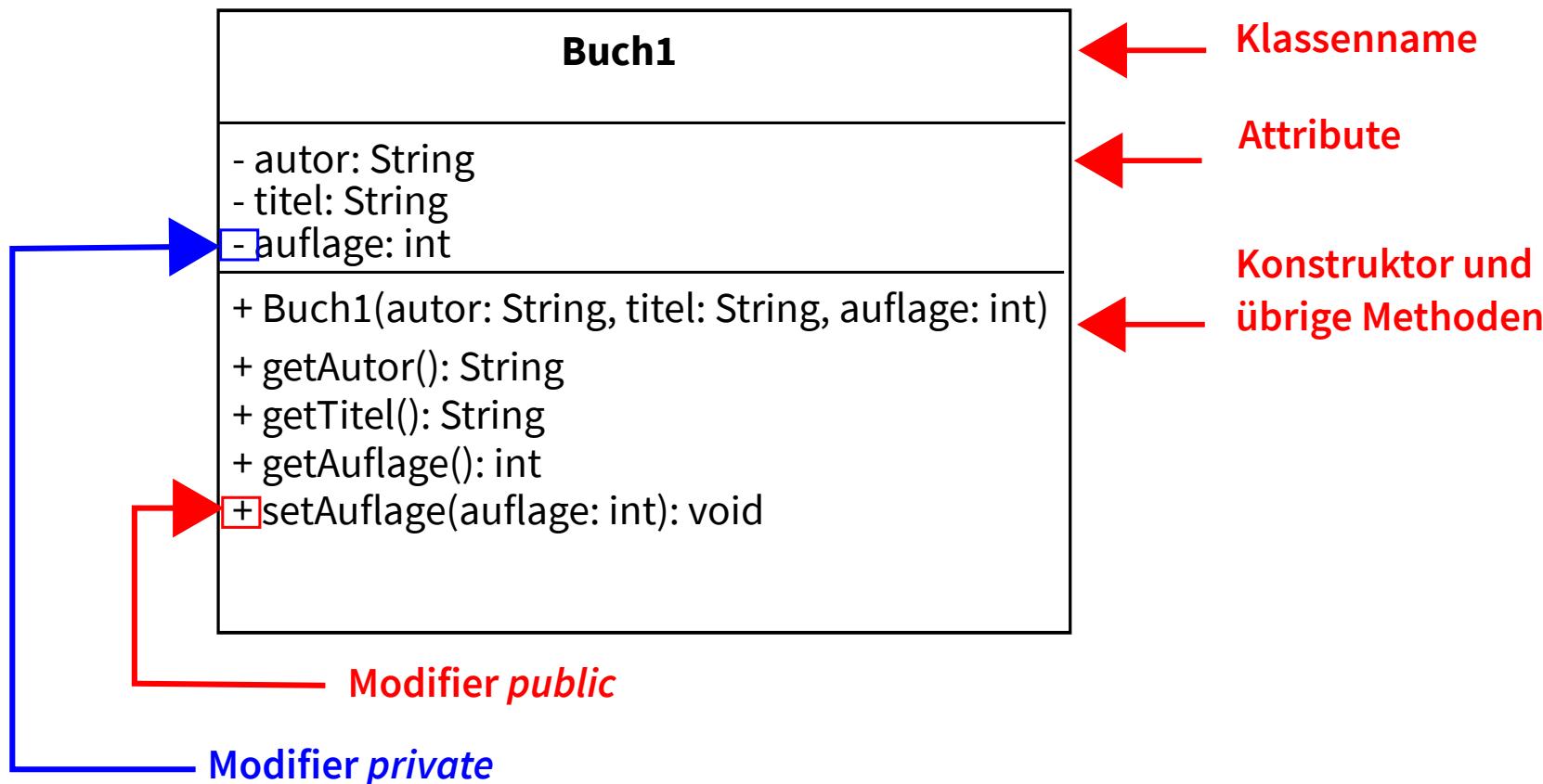
# Bisher behandelt: Klassendarstellung direkt in Java

## Bisher: Buch-Klasse in Java – Teil 3

```
16  public int getAuflage() {  
17      return auflage;  
18  } // end method  
19  
20  public void setAuflage(int auflage) {  
21      this.auflage = auflage;  
22  } // end method  
23 } // end class
```

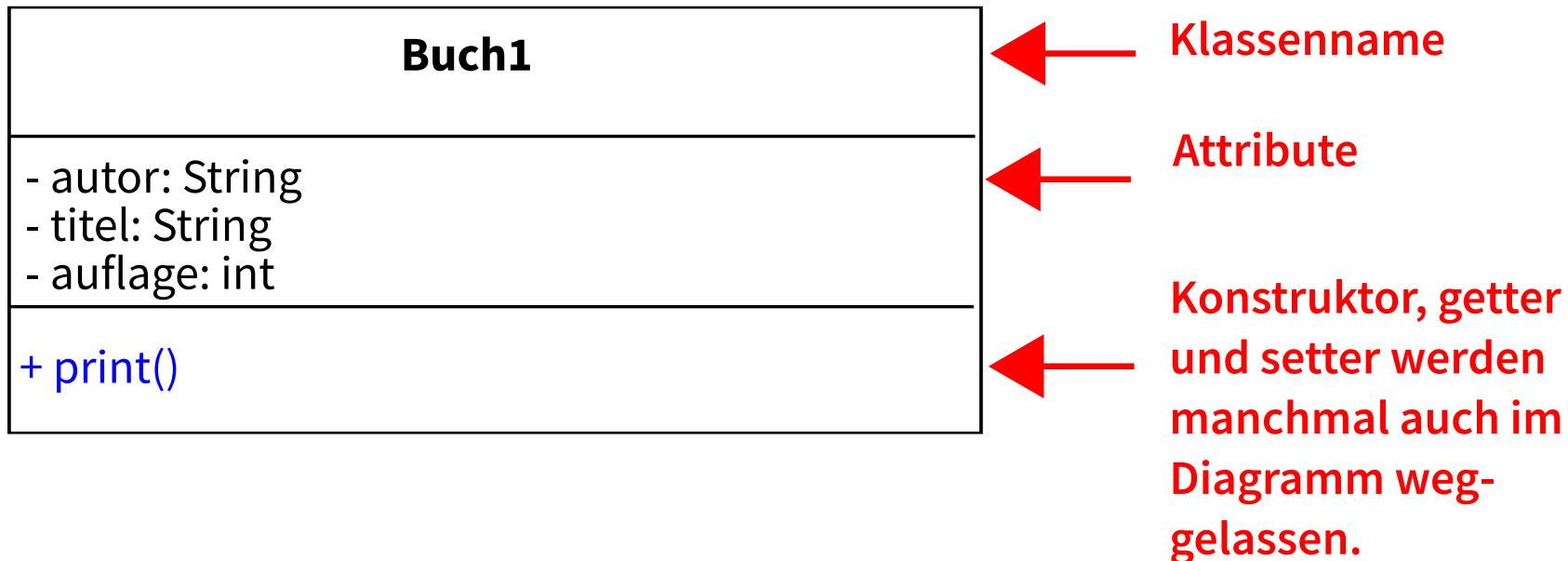
# Neu: Graphische Darstellung von Klassen in der UML2

## Einfache Klassendiagramme: UML-Darstellung einer Klasse Buch1



# Neu: Graphische Darstellung von Klassen in der UML2

## Einfache Klassendiagramme: Reduzierte UML-Darstellung einer Klasse Buch1



Nur "echte" Methoden werden in dieser Variante des Diagramms gezeigt!

# Neu: Graphische Darstellung von Klassen in der UML2

## Einfache Klassendiagramme

### Erläuterung der UML-Elemente des Klassendiagramms

- Der obere Block enthält immer den Klassennamen.
- Der mittlere Block enthält die Spezifikation der Attribute der Klasse.
- Der untere Block enthält die Spezifikation der Konstruktoren und der sonstigen Methoden der Klasse.
- Der Modifier *public* erscheint als + -Zeichen vor den Attributen bzw. Methoden.
- Der Modifier *private* erscheint als - - Zeichen vor den Attributen bzw. Methoden.

# Neu: Graphische Darstellung von Klassen in der UML2

## Einfache Klassendiagramme

### Erläuterung der UML-Elemente des Klassendiagramms

- Der Datentyp von Attributten und Parametervariablen wird – entgegen unserer gewohnten Syntax von Java – **nach Bezeichner und Doppelpunkt** angegeben mit:  
`variablename : Datentyp`
  
- Auch der Rückgabetyp von Methoden wird – entgegen unserer gewohnten Syntax von Java – **nach Methodenkopf und Doppelpunkt** angegeben mit:  
`methodenKopf(...) : Datentyp`

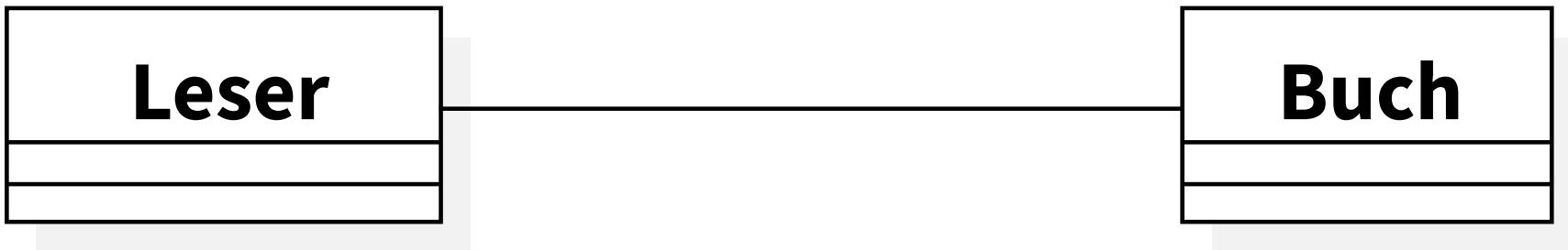
# Klassenbeziehungen – Teil1

Bisher: Klassen wurden einzeln dargestellt ...

- ... aber standen nicht in Beziehung zueinander.
- In Real-World-Anwendungen gibt es nicht nur eine Klasse, sondern viele Klassen.
- Zwischen den Klassen bestehen Abhängigkeiten, wie beispielsweise
- *Leser besitzt Bücher* (1:n-Beziehung), oder ...
- *Leser liest Buch*(1:1-Beziehung)

# Assoziation

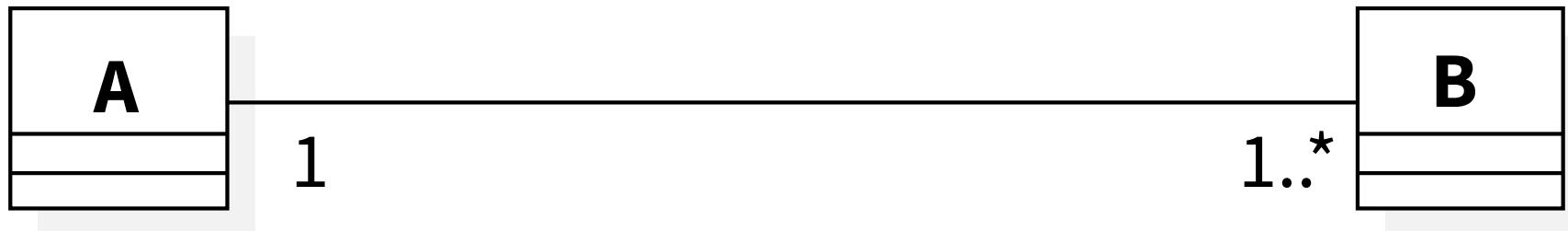
## Einfache, ungerichtete Assoziation



- Welche Klasse über Attribute hier auf wen Zugriff hat, ist noch nicht ersichtlich (Richtung fehlt).
- Leserichtung (Buch **hat** Leser, Leser **liest** Buch) der Beziehung ist auch noch nicht erkennbar.
- Multiplizität (1:1, 1:n, etc) fehlt noch.

# Assoziation

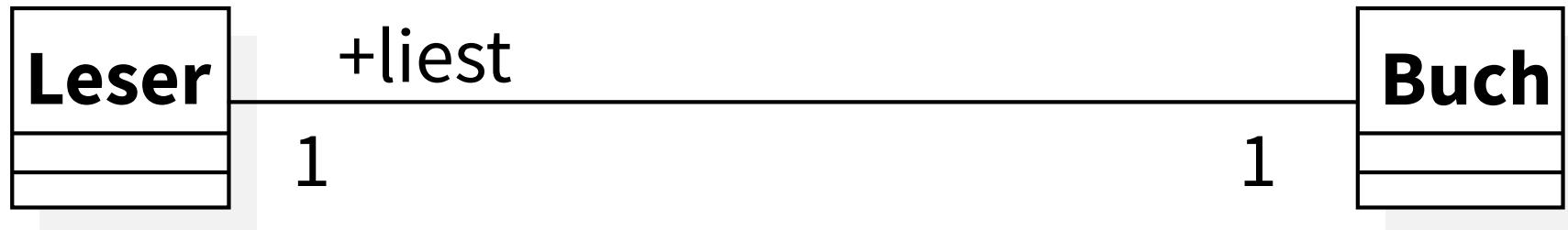
## Einfache, ungerichtete Assoziation mit Multiplizitäten



- Multiplizität / Kardinalität: gibt an, wie viele Objekte der Klasse B ein Objekt der Klasse A kennen kann oder muss.
  - Multiplizität wird an beiden Enden der Assoziation angegeben.
  - Hier: 1 Objekt der Klasse A kennt 1 bis mehrere Objekte der Klasse B.
- Folgende Multiplizitäten werden an den Assoziationsenden häufig verwendet:
  - **1** steht für: 1 Objekt von Klasse X
  - **1..\*** steht für: 1 oder mehrere Objekte von Klasse X
  - **0..\*** steht für: 0 oder mehrere Objekte von Klasse X (also: Objekte von X können optional vorkommen)

# Assoziation

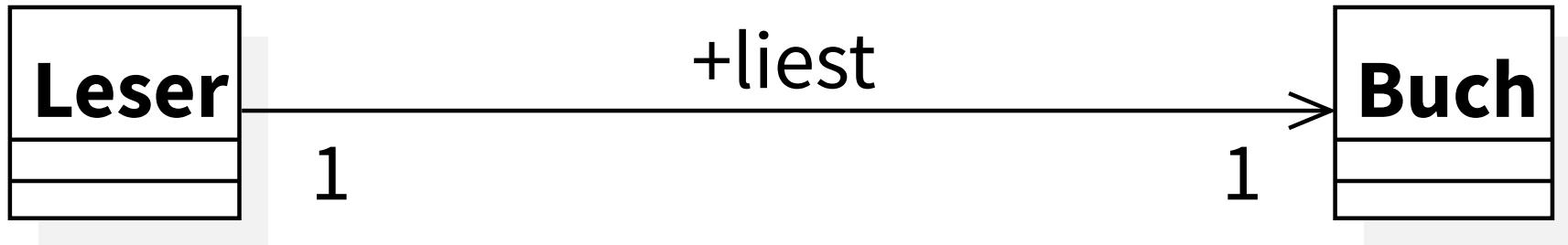
Einfache, ungerichtete Assoziation mit Multiplizitäten



- Buch-Beispiel:
- 1 Leser liest zu einer Zeit 1 Buch
- Assoziation hat einen Namen (**liest**)
- Assoziation ist (**public**)

# Assoziation

## Gerichtete Assoziation mit Multiplizitäten und Sichtbarkeit



- 1 Objekt der Klasse Leser greift auf 1 Objekt der Klasse Buch zu
- Zugriff erfolgt z. B. über ein Attribut vom Typ Buch

# Assoziation

Gerichtete Assoziation mit Multiplizitäten und Sichtbarkeit



- 1 Leser kann zu einer Zeit 0 oder mehrere Bücher besitzen
- Zugriff erfolgt z.B. über eine Liste von Buch-Referenzen

# Assoziation

## Gerichtete Assoziation – Zugriff und Leserichtung



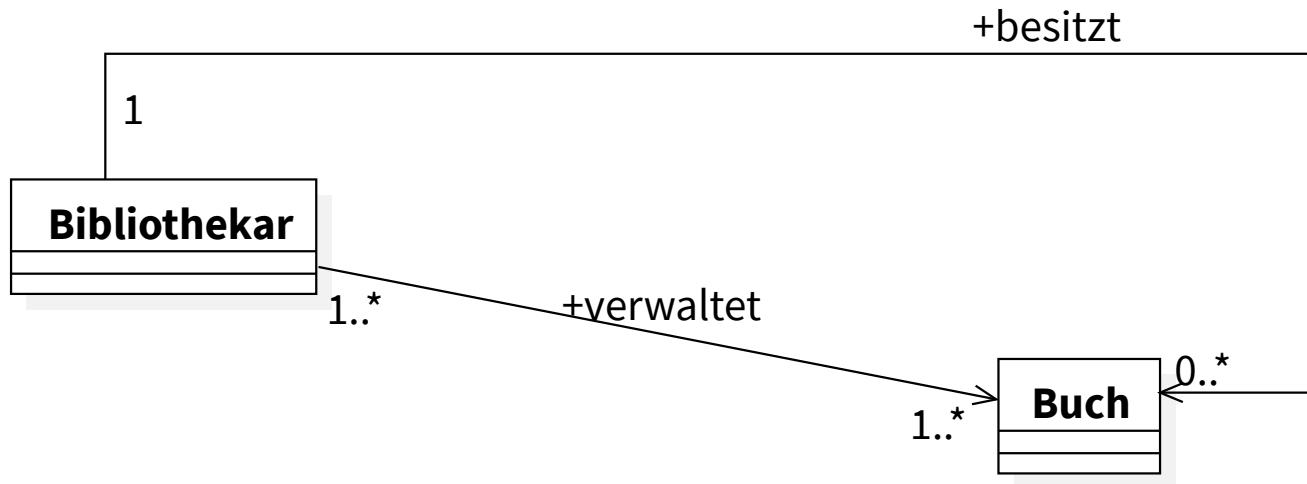
→ : **Offene Pfeilspitze**: "hat Zugriff auf" (hier: Klasse Autor enthält eine Liste mit Buch-Referenzen; Klasse Buch enthält eine Liste mit seinen Autoren)

► : **Dreieck**: Leserichtung (hier: Autor kann Bücher schreiben, aber Bücher können keine Autoren Schreiben)

- Pfeilspitzen an den **Enden der Assoziation** beschreiben, wer auf wen Zugriff hat-
- Dreieck **über der Assoziation** sagt aus, in welcher Richtung die Assoziation zu lesen ist.

# Assoziation

## Gerichtete Assoziation – mehr als 2 Klassen



- **1 oder mehrere** Bibliothekare verwalten **1 oder mehrere** Bücher – nämlich den Bibliotheksbestand.
  - **Achtung:** Nach dieser Modellierung darf der Bibliotheksbestand **nicht leer** sein!
  - → Das muss beim Entnehmen überprüft werden! (Sinnvoll so???)
- **1** Bibliothekar kann aber privat auch **0 oder mehrere** Bücher privat aus dem Bestand der Bibliothek erworben haben (falls die Bibliothek so etwas ermöglicht)

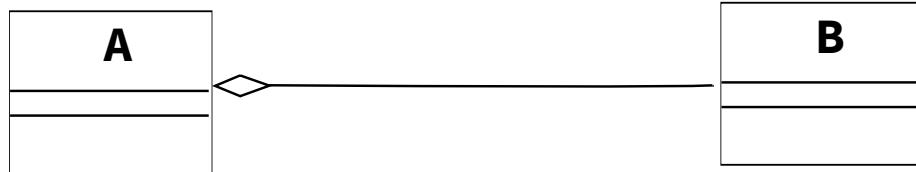
# Aggregation und Komposition

... sind spezielle Assoziationen.

- Beide drücken aus: Objekt von Klasse A **ist Bestandteil von** Objekt von Klasse B.
- Auch hier sind Multiplizitäten möglich.

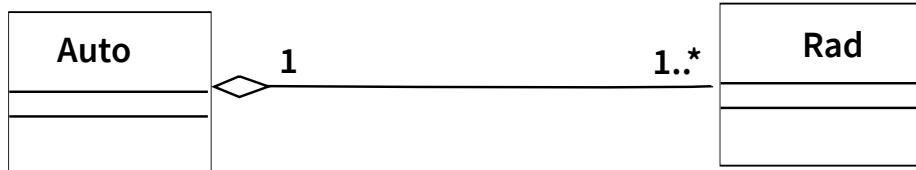
# Aggregation

## Formale Darstellung



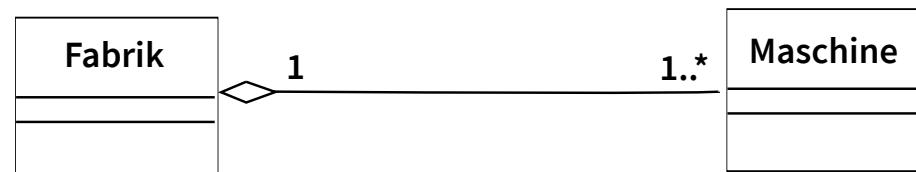
### Aggregation:

- Objekt(e) von Klasse B sind Bestandteil von Klasse A
- Sie existieren aber auch noch weiter, wenn Objekt der Klasse A gelöscht wird.



### Beispiel 1:

- Die Räder sind Bestandteil des Autos.
- Sie existieren aber auch noch weiter, wenn das Auto in die Schrottplresse kommt. Sie können z. B. weiterverkauft werden.

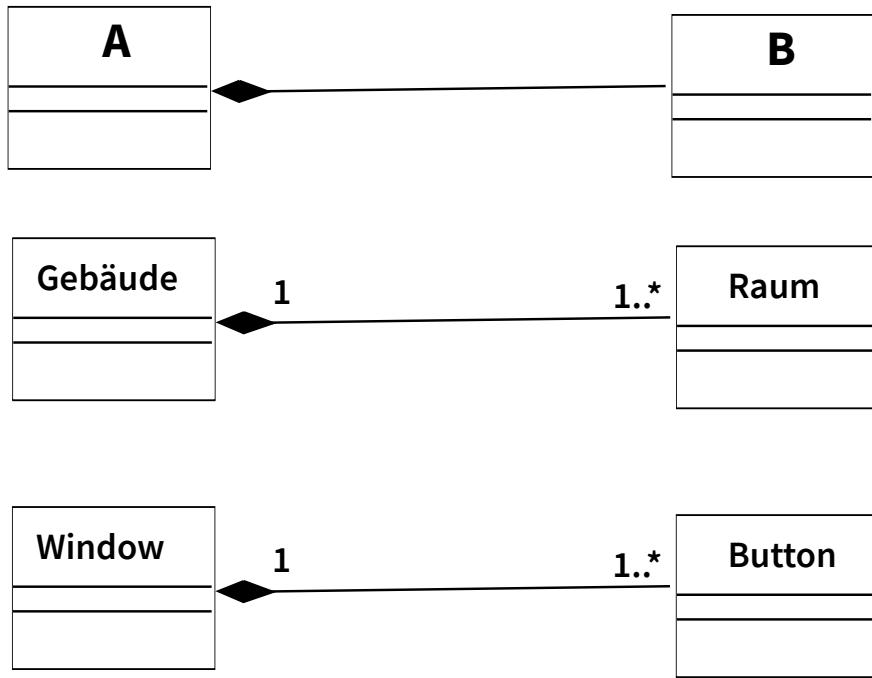


### Beispiel 2:

- Maschinen stehen im Fabrikgebäude einer Firma.
- Sie existieren aber auch noch weiter, wenn das Gebäude aufgegeben wird. Sie können umplaziert oder weiterverkauft werden.

# Komposition

## Formale Darstellung



### Komposition:

- Auch hier: Objekt(e) von B sind Bestandteil von A
- Aber: Wenn aber A gelöscht wird, so existiert B ebenfalls nicht mehr, sondern wird ebenfalls gelöscht.

### Beispiel 1:

- 1 oder mehrere Räume sind Bestandteil eines Gebäudes (z. B. Gebäude 210 in der Gartenstraße)
- Wenn die HS das Gebäude **nicht mehr mietet**, kann sie im WebUntis auch nicht mehr die Räume verplanen! (Sie sind also **mit gelöscht**)

### Beispiel 2:

- 1 oder mehrere Buttons sind in einem Window (z. B. Öffnen-Dialog) platziert.
- Wenn das Window gelöscht wird (z. B. nach dem Zuklappen), so sind die Buttons ebenfalls gelöscht.

Fakultät Informatik

# Programmierung 2

Vererbungsbeziehungen

Vorwort

Einstieg in Java

Einstieg in die OOP: Klassen und Objekte/Instanzen

Die Klasse String

Graphische Darstellung von Klassen mit der UML – Teil 1

Vererbungsbeziehungen

Wozu Vererbung?

Einfache Vererbungsbeziehungen in Java

Grundzüge Vererbung

Vererbungsprinzip in Java – Eltern- und Kindklassen

Einfachvererbung – Mehrfachvererbung in Java

Vererbungsprinzip in Java – Objekterzeugung

Polymorphie – Vielgestaltigkeit von Objekten

Die Superklasse aller Java-Klassen: Die Klasse Object

Überladen vs. Überschreiben in der OOP

## Abstrakte Klassen

Wozu abstrakte Klassen?

Darstellung abstrakter Klassen

Einschub: Annotation @Override

## Interfaces

Grundlagen Interfaces

Einfache Vererbungsbeziehungen mit Interfaces

Default-Implementierung von Interface-Methoden

Mehrstufige Vererbungshierarchien

Auswirkungen von Interfaces auf Polymorphie

## Der instanceof-Operator

## Exceptions



# Einstieg Streams

## Bisher bekannte OOP-Konzepte

Sie kennen bereits ...

- Klassen mit Attributen und Methoden
- Zugriffsschutz-Modifizierer
- Modifizierer `static`
- Assoziative Klassenbeziehungen – können **hat**-Beziehungen modellieren.

# Bisher bekannte OOP-Konzepte

## Was tun in folgendem Fall? – UML-Sicht

Person
-vorname: String
-nachname: String
+Person(vorname: String, nachname: String)
+getNachname(): String
+setNachname(nachname: String): void
+getVorname(): String

Lehrende
-vorname: String
-nachname: String
-persnr: int
-jahresgehalt: double
+Lehrende(vorname: String, nachname: String, persnr: int, jahresgehalt: double)
+getNachname(): String
+setNachname(nachname: String): void
+getVorname(): String
+getPersnr(): int
+getJahresgehalt(): double

Studierende
-vorname: String
-nachname: String
-matrikelnr: int
+Studierende(vorname: String, nachname: String, matrikelnr: int)
+getNachname(): String
+setNachname(nachname: String): void
+getVorname(): String
+getMatrikelnr(): int

### Die Attribute

- vorname
- nachname

müssen in jeder Klasse neu implementiert werden!

Ebenso die getter() bzw. setter()

## Bisher bekannte OOP-Konzepte

Was tun in folgendem Fall? – Java-Sicht – Klasse Person

```
1 public class Person {  
2     private String vorname;  
3     private String nachname;  
4     public Person(String vorname, String nachname) {  
5         this.vorname = vorname;  
6         this.nachname = nachname;  
8     } // end constructor
```

# Bisher bekannte OOP-Konzepte

Was tun in folgendem Fall? – Java-Sicht – Klasse Person

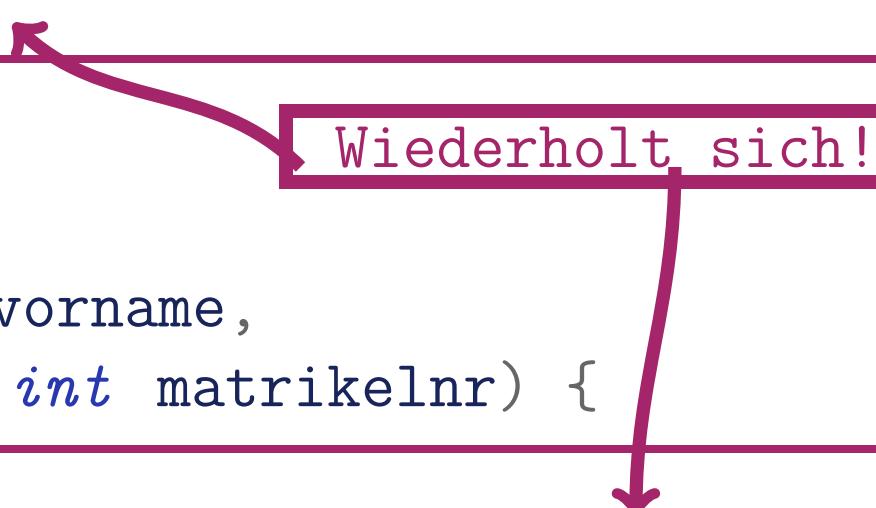
```
9   public String getNachname() {  
10      return nachname;  
11  } // end method  
  
12  
13  public void setNachname(String nachname) {  
14      this.nachname = nachname;  
15  } // end method  
  
16  
17  public String getVorname() {  
18      return vorname;  
19  } // end method  
  
20 } // end class
```

## Bisher bekannte OOP-Konzepte

Was tun in folgendem Fall? – Java-Sicht – Klasse Studierende

```
1 public class Studierende {  
2     private String vorname;  
3     private String nachname;  
4     private int matrikelnr;  
5  
6     public Studierende(String vorname,  
7                         String nachname, int matrikelnr) {  
8         this.vorname = vorname;  
9         this.nachname = nachname;  
10        this.matrikelnr = matrikelnr;  
11    } // end constructor
```

Wiederholt sich!



## Bisher bekannte OOP-Konzepte

Was tun in folgendem Fall? – Java-Sicht – Klasse Studierende

```
12  public String getNachname() {  
13      return nachname;  
14  } // end method  
  
15  
16  public void setNachname(String nachname) {  
17      this.nachname = nachname;  
18  } // end method  
  
19  
20  public String getVorname() {  
21      return vorname;  
22  } // end method
```

Wiederholt sich!

## Bisher bekannte OOP-Konzepte

Was tun in folgendem Fall? – Java-Sicht – Klasse Studierende

```
23  public int getMatrikelnr() {  
24      return matrikelnr;  
25  } // end method  
26 } // end class
```

## Bisher bekannte OOP-Konzepte

Was tun in folgendem Fall? – Java-Sicht – Objekterzeugung Person und Studierende

```
1 public class PersonenMain1 {  
2     public static void main(String[] args) {  
3         // Eine Person mit  
4         // Vorname Hugo, Nachname Helmchen  
5         Person p1 = new Person("Hugo","Helmchen");  
6  
7         // Eine Studierende mit  
8         // Vorname Erna, Nachname Etepetae  
9         // Matrikelnr. 4711  
10        Studierende s1 = new Studierende("Erna","Etepetae",4711);
```

## Bisher bekannte OOP-Konzepte

Was tun in folgendem Fall? – Java-Sicht – Objekterzeugung Person und Studierende

```
11 // Aufruf getter()-Implementierung von Person
12 String nachname1 = p1.getNachname();
13
14 // Aufruf getter-Implementierung von Studierende
15 String nachname2 = s1.getNachname();
16 } // end method main
17 } // end class
```

## Bisher bekannte OOP-Konzepte

Mit den bisherigen Konzepten können wir nicht ...

- **ist**-Beziehungen modellieren.
- Beispiel: Ein/e Studierende/r **ist** eine Person mit zusätzlichen Attributen
- Ebenso ein/e Lehrende/r
- Eigentlich gemeinsame Eigenschaften werden in jeder Klasse  
**neu implementiert!**

## Bisher bekannte OOP-Konzepte

Das gezeigte Klassensystem widerspricht dem DRY-Prinzip

**DRY }**    **Don't Repeat Yourself!**

- Code-Anteile sollten nur einmal implementiert werden!
- Wiederholungen sind fehleranfällig!

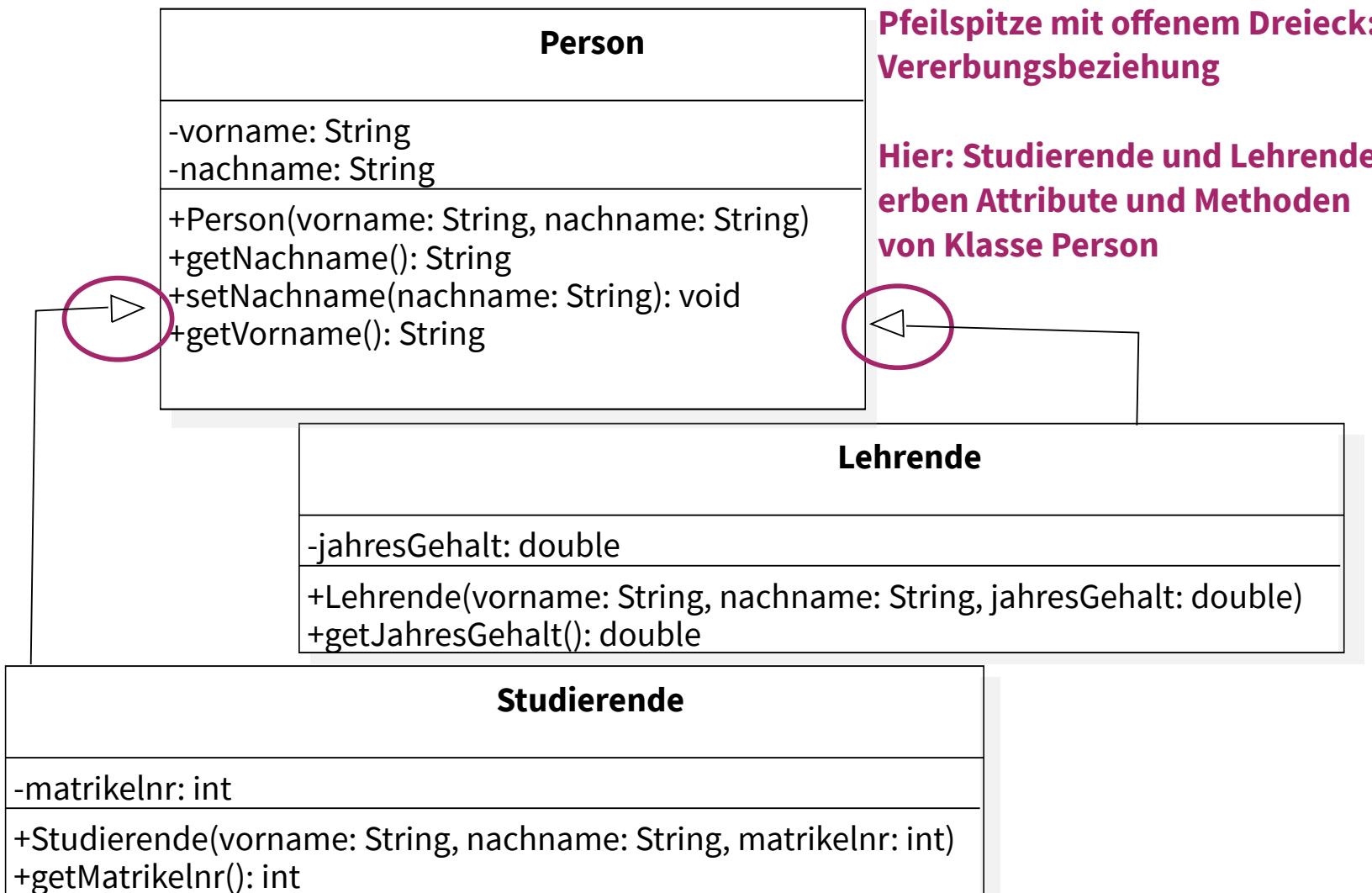
## Neuer Ansatz: Vererbungsprinzip

Mit dem Vererbungsprinzip können wir ...

- Gemeinsame Eigenschaften (z.B. Vorname und Nachname) werden in einer **Superklasse** implementieren.
  - Synonym: **Elternklasse**,
  - **Basisklasse**
  - Oder, da Klassen eben auch Datentypen sind: **Supertyp**
- Unterscheidende Eigenschaften (z. B. Matrikelnummer oder Jahresgehalt) werden in **Subklassen** implementieren.
  - Synonym: **Kindklasse**,
  - **abgeleitete Klasse**
  - Oder, da Klassen eben auch Datentypen sind: **Subtyp**

# Vererbungsprinzip – Klassenstruktur

## Klassensystem mit Vererbungsbeziehung – UML-Sicht



# Vererbungsprinzip – Klassenstruktur

## Klassensystem mit Vererbungsbeziehung – UML-Sicht

- Die Beziehung Klasse *B* **erbt von** Klasse *A* wird in der UML ausgedrückt durch:



- Jede Klasse enthält nur noch die Inhalte, für die sie wirklich zuständig ist!

# Vererbungsprinzip – Klassenstruktur

## Klassensystem mit Vererbungsbeziehung – Java-Sicht – Klasse Person

```
1 public class Person {  
2  
3     private String vorname; // Attribute sind  
4     private String nachname; // unveraendert  
5  
6     public Person(String vorname, String nachname) {  
7  
8         this.vorname = vorname;  
9         this.nachname = nachname;  
10    } // end constructor
```

# Vererbungsprinzip – Klassenstruktur

## Klassensystem mit Vererbungsbeziehung – Java-Sicht – Klasse Person

```
11  public String getNachname() {  
12      return nachname;  
13  } // end method  
14  
15  public void setNachname(String nachname) {  
16      this.nachname = nachname;  
17  } // end method  
18  
19  public String getVorname() {  
20      return vorname;  
21  } // end method  
22 } // end class
```

# Vererbungsprinzip – Klassenstruktur

## Java-Sicht – Klasse Studierende – Attribute

```
1 public class Studierende extends Person {  
2     // nur Attribute, die NICHT SCHON in der Superklasse  
3     // stehen, werden hier noch implementiert.  
4     private int matrikelnr;
```

- Klasse *Studierende* **erbt von** Klasse *Person*.
- Anders ausgedrückt: Klasse *Studierende* **erweitert** Klasse *Person* um bestimmte Eigenschaften!
- Die Vererbungsbeziehung **erbt von** wird in Java mit dem Schlüsselwort **extends** ausgedrückt.

# Vererbungsprinzip – Klassenstruktur

## Java-Sicht – Klasse Studierende – Konstruktor

```
5   public Studierende(String vorname,
6                     String nachname, int matrikelnr) {
7     // Aufruf des Superklassenkonstruktors -- er setzt
8     // Vorname und Nachname in der Elternklasse
9     super(vorname, nachname);
10
11    // Setzen der Matrikelnr. im eigenen Konstruktor
12    this.matrikelnr = matrikelnr;
13  } // end constructor
```

- Konstruktor von Klasse *Studierende* **muss benötigte Parameter** an den Konstruktor von Klasse *Person* weitergeben.
- Dies geschieht durch den `super()`-Aufruf an erster Stelle im Rumpf des Konstruktors von Klasse *Studierende*.

# Vererbungsprinzip – Klassenstruktur

Java-Sicht – Klasse Studierende – Konstruktor – Das geht **NICHT**:

```
5  public Studierende(String vorname,
6      String nachname, int matrikelnr) {
7      // Setzen der ererbten Attribute
8      this.vorname = vorname;
9      this.nachname = nachname;
10     this.matrikelnr = matrikelnr; // Setzen der Matrikelnr.
11 } // end constructor
```

- Die Attribute `vorname` und `nachname` sind **private**!
- **Kind-Klassen** können auf **ererbte, private** Attribute **nicht direkt** zugreifen!
- Dies ist nur über den **Superklassen-Konstruktor** möglich!

## Vererbungsprinzip – Klassenstruktur

Java-Sicht – Klasse Studierende – Konstruktor – Auch das geht **NICHT**:

```
5  public Studierende(String vorname,  
6                      String nachname, int matrikelnr) {  
7  
8      this.matrikelnr = matrikelnr; // Setzen der Matrikelnr.  
9      super(vorname, nachname); // Aufruf Superklassenkonstr.  
10  
11 } // end constructor
```

- Der Superklassen-Konstruktor muss immer **zuerst** aufgerufen werden!
- Erst **danach** können weitere Statements abgesetzt werden!

# Vererbungsprinzip – Klassenstruktur

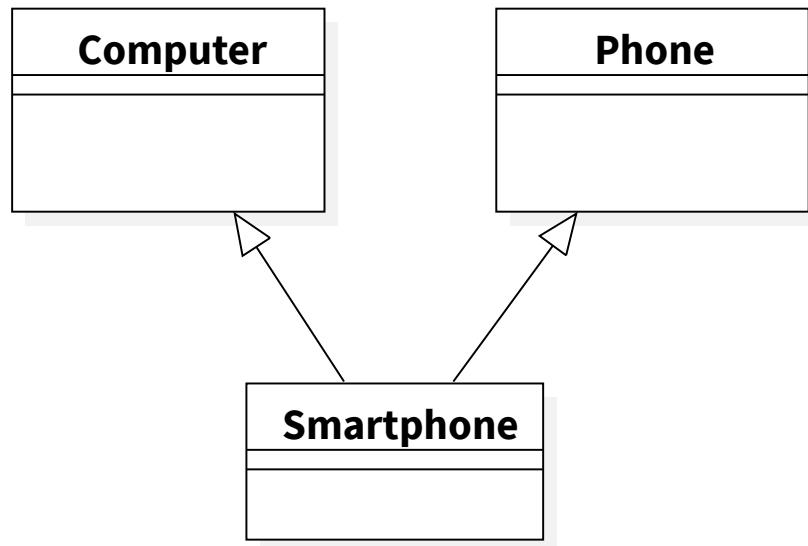
## Java-Sicht – Klasse Studierende – getter/setter:

```
14 // Nur noch getter fuer die eigenen Attribute --
15 // getter und setter der ererbten Attribute sind
16 // in der Superklasse implementiert!
17 public int getMatrikelnr() {
18     return matrikelnr;
19 } // end method
20 } // end class
```

- Die Klasse *Studierende* enthält nur noch getter/setter für die **eigenen** Attribute.
- Sie **erbt** die getter/setter für die ererbten Attribute.

# Vererbungsprinzip – Mehrfachvererbung

## Mehrfachvererbung – Darstellung in der UML



- Mehrfachvererbung: Eine Kindklasse hat **mehrere Elternklassen**
- Dieses Prinzip ist in der OOP zwar bekannt, aber ...
- **Mehrfachvererbung wird von Java nicht unterstützt! (Von C++: ja!)**
- →**Java-Klassen können nur eine Eltern-Klasse haben!**

# Vererbungsprinzip – Objekterzeugung

## Java-Sicht – Objekterzeugung mit Vererbungsprinzip – einfachster Fall

```
1 // Objekterzeugung
2 Person p1 = new Person("Hugo", "Helmchen");
3 Studierende s1 = new Studierende("Erna", "Etepeta", 4711);
4
5 // Aufruf der eigenen Methode getNachname()
6 String nachname1 = p1.getNachname();
7
8 // Aufruf der ererbten Methode getNachname()
9 String nachname2 = s1.getNachname();
```

- Von s1 wird die **ererbte** Methode getNachname() aufgerufen.
- Sie verhält sich so, als ob zu Klasse Studierende gehört!

# Vererbungsprinzip – Polymorphie

## Java-Sicht – Objekterzeugung mit Vererbungsprinzip – Polymorphie

```
1 // Objekterzeugung
2 Person s1 = new Studierende("Erna", "Etepete", 4711);
3
4 // Aufruf der ererbten Methode getNachname()
5 String nachname1 = s1.getNachname();
```

- Von s1 kann als *Person* deklariert werden ...
- ... und als *Studierende* erzeugt werden.
- Eine Instanz vom Typ *Studierende* **ist** eine *Person!*

## Vererbungsprinzip – Polymorphie

Java-Sicht – Objekterzeugung mit Vererbungsprinzip – Polymorphie – das geht **NICHT**:

```
1 // Objekterzeugung
2 Person s1 = new Studierende("Erna", "Etepete", 4711);
3
4 // Aufruf der ererbten Methode getNachname() -- OK
5 String nachname = s1.getNachname();
6
7 // Aufruf einer Methode der Klasse Studierende -- NICHT OK
8 int matrikeln = s1.getMatrikelnr();
```

- Instanz s1 ist vom Typ *Person* ...
- ... *Person* kennt keine Methode `getMatrikelnr()`!

## Vererbungsprinzip – Polymorphie

Java-Sicht – Objekterzeugung mit Vererbungsprinzip – Polymorphie – das geht:

```
1 // Objekterzeugung
2 Person s1 = new Studierende("Erna", "Etepete", 4711);
3
4 // Aufruf der ererbten Methode getNachname() -- OK
5 String nachname = s1.getNachname();
6
7 // Aufruf einer Methode der Klasse Studierende
8 // mit TypeCast -- Klammerung, weil Punkt vor Cast (Prio)
9 int matnr = ((Studierende) s1).getMatrikelnr();
```

- Methode aus Klasse *Studierende* ist nach TypeCast wieder sichtbar ...
- ... und kann verwendet werden – sie wurde ja beim `new`-Aufruf erzeugt!

# Vererbungsprinzip – Polymorphie

## Java-Sicht – Objekterzeugung mit Vererbungsprinzip – Polymorphie – Zusammenfassung

- Objekte von Kindklassen können an eine Referenz vom Typ Elternklasse zugewiesen werden:  
`Elternklasse var = new Kindklasse();`
- In diesem Fall sind die Methoden der Kindklasse nicht sichtbar.
- Sie können beim Aufruf wieder sichtbar gemacht werden mit:  
`((Kindklasse)var).kindMethode();`
- Achtung: Der Punkt `.` hat Priorität vor dem Cast (**Kindklasse**). Damit zuerst gecastet wird und dann erst mit `.` auf die Methode zugegriffen wird, muss der Cast geklammert werden!

# DIE Superklasse aller Java-Klassen: Die Klasse Object

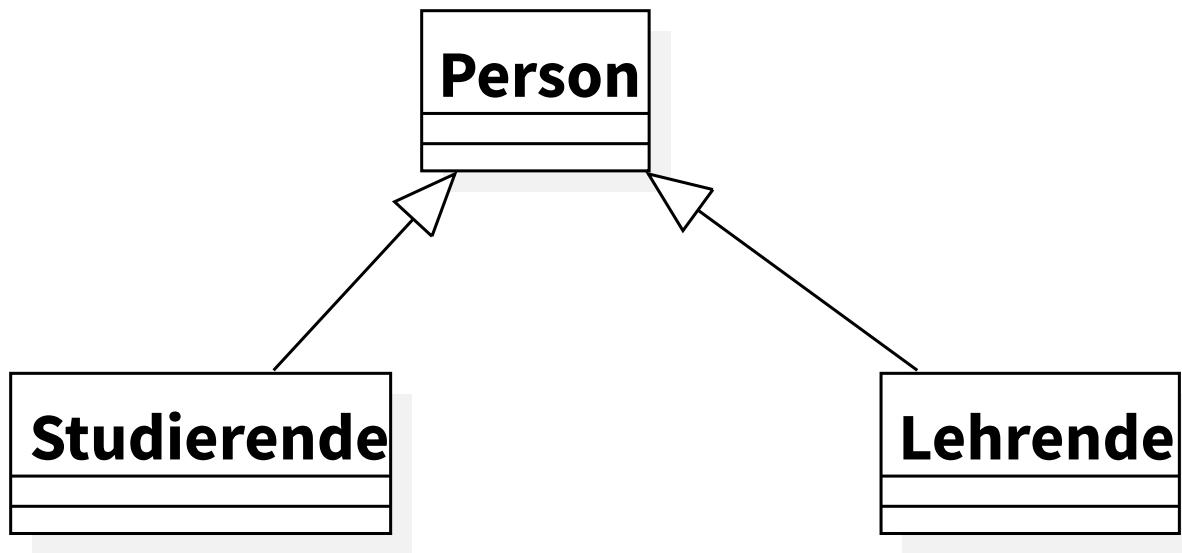
Wenn eine Java-Klasse scheinbar keine Superklasse hat ...

```
1 public class Person {  
2     // Attribute ...  
3     // Konstruktoren ...  
4     // Methoden ...  
5 }
```

- ... so erbt sie von der API-Klasse **Object**.
- Diese enthält für jedes Objekt dieser Klasse Metainformationen wie Klassename, usw., die zur Laufzeit abgefragt werden können.
- Sie enthält auch interessante Methoden z.B. zum Vergleich von Objektinhalten.

# DIE Superklasse aller Java-Klassen: Die Klasse Object

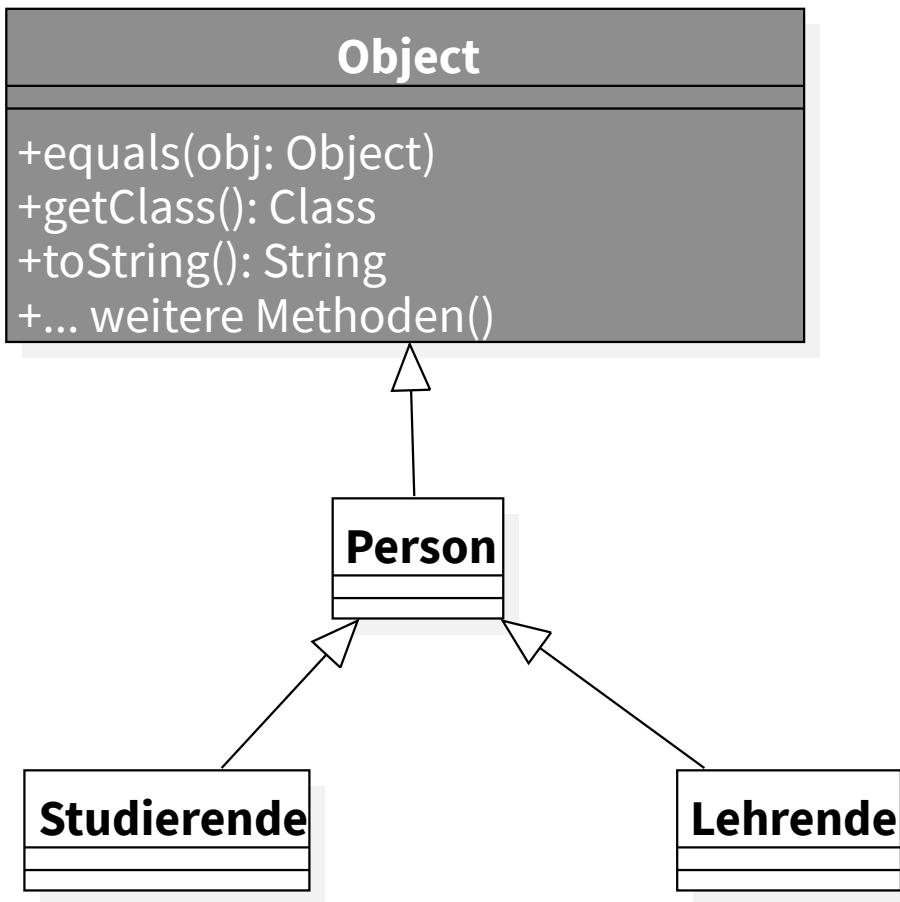
## Bisher bekannte Vererbungshierarchie



Aus dieser Konstruktion wird also ...

# DIE Superklasse aller Java-Klassen: Die Klasse Object

## Vererbungshierarchie mit Betrachtung der Klasse Object



... in Wirklichkeit diese!

# DIE Superklasse aller Java-Klassen: Die Klasse Object

Szenario zum Testen von Klasse Object: Klasse Person

```
1 public class Person {  
2     private String vorname;  
3     private String nachname;  
4  
5     public Person(String vorname, String nachname) {  
6         this.vorname = vorname;  
7         this.nachname = nachname;  
8     } // end constructor  
9     // Rest wie vorher ...  
10 } // end class
```

# DIE Superklasse aller Java-Klassen: Die Klasse Object

Szenario zum Testen von Klasse Object: Klasse Studierende

```
1 public class Studierende extends Person{  
2  
3     private int matrikelnr;  
4  
5     public Studierende(String vorname,  
6                         String nachname, int matrikelnr) {  
7         // Aufruf Superklassenkonstruktor  
8         super(vorname,nachname);  
9         this.matrikelnr = matrikelnr;  
10    } // end constructor  
11    // Rest wie vorher ...  
12 } // end class
```

# DIE Superklasse aller Java-Klassen: Die Klasse Object

Szenario zum Testen von Klasse Object: Klasse Lehrende

```
1 public class Lehrende extends Person{  
2     private int persnr;  
3     private double jahresgehalt;  
4  
5     public Lehrende(String vorname, String nachname,  
6                     int persnr, double jahresgehalt) {  
7         // Aufruf Superklassenkonstruktor  
8         super(vorname,nachname);  
9         this.persnr = persnr;  
10        this.jahresgehalt = jahresgehalt;  
11    } // end constructor  
12    // Rest wie vorher ...  
13 } // end class
```

# DIE Superklasse aller Java-Klassen: Die Klasse Object

Szenario zum Testen von Klasse Object: Klasse TesteObjectMain

```
1 import java.util.Scanner;  
2  
3 public class TesteObjectMain {  
4  
5     public static void main(String[] args) {  
6  
7         Person pers=null;  
8         Scanner s = new Scanner(System.in);  
9  
10        System.out.println("Was wollen Sie erzeugen?");  
11        System.out.println("Person, Studierende oder Lehrende?");  
12        String clazzname = s.next();
```

# DIE Superklasse aller Java-Klassen: Die Klasse Object

Szenario zum Testen von Klasse Object: Klasse TestObjectMain

```
13     switch(clazzname) {  
14         case "Person": pers =  
15             new Person("Hans", "Koch"); break;  
16         case "Studierende":  
17             pers = new Studierende("Hans", "Koch", 4711);  
18             break;  
19         case "Lehrende":  
20             pers = new Lehrende("Hans", "Koch", 5678, 35000.0);  
21             break;  
22     } // end switch
```

# DIE Superklasse aller Java-Klassen: Die Klasse Object

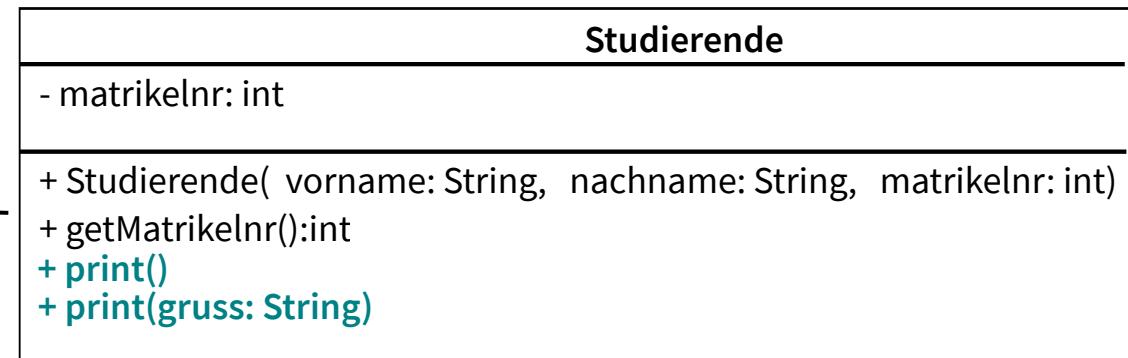
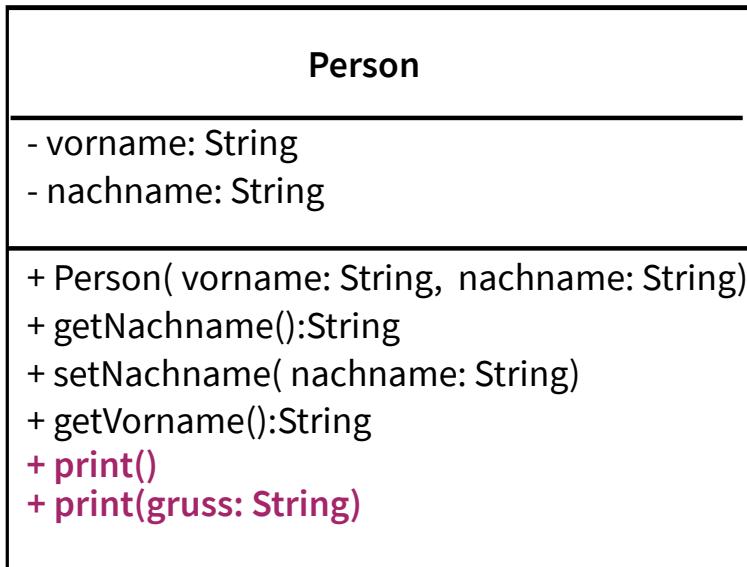
Szenario zum Testen von Klasse Object: Klasse TestObjectMain

```
21  String clazzchosen = pers.getClass().getName();
22  System.out.println(
23      "Sie haben gewaehlt: " + clazzchosen);
24 } // end method main()
25 } // end class
```

→ Über von Object ererbte Methode `getClass()` fragen wir den Klassennamen ab!

# Überladen und Überschreiben in der OOP

## Überladen von Methoden – UML-Sicht



Die `print()`-Methode wurde in  
der Klasse Person 2x überladen

Die `print()`-Methode wurde in  
der Klasse Studierende ebenfalls 2x überladen

Die Methoden der Superklasse Person  
wurden dabei überschrieben.

# Überladen und Überschreiben in der OOP

## Überladen von Methoden – Grundlagen

- Überladen von Methoden innerhalb einer Klasse bedeutet: Wir stellen mehrere Varianten einer Methode zur Verfügung.
- Die Varianten **heissen gleich**, haben aber **unterschiedliche Signaturen**.
- → Die Parameterlisten müssen sich unterscheiden!
- Vorteil des Überladens von Methoden: Methoden die gleiche Dinge tun, aber unterschiedliche Parameter benötigen, können gleich heißen.
- Wichtig: Auch Konstruktoren können mehrfach überladen werden!

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überladen von Methoden – Klasse Person

```
1 public class Person {  
2  
3     private String vorname;  
4     private String nachname;  
5  
6     public Person(String vorname, String nachname) {  
7         this.vorname = vorname;  
8         this.nachname = nachname;  
9     } // end constructor
```

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überladen von Methoden – Klasse Person

```
10  public String getNachname() {  
11      return nachname;  
12  } // end method  
  
13  
14  public void setNachname(String nachname) {  
15      this.nachname = nachname;  
16  } // end method  
  
17  
18  public String getVorname() {  
19      return vorname;  
20  } // end method
```

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überladen von Methoden – Klasse Person

```
21 public void print() {  
22     System.out.println("Ich heisse "  
23         + vorname + " " + nachname);  
24 } // end method  
  
25  
26 public void print(String gruss) {  
27     System.out.println(gruss + ", ich heisse "  
28         + vorname + " " + nachname);  
29 } // end method  
  
31 } // end class
```

→ 2 überladene Versionen von **print(...)**

# Überladen und Überschreiben in der OOP

## Überladen von Methoden – Zusammenfassung

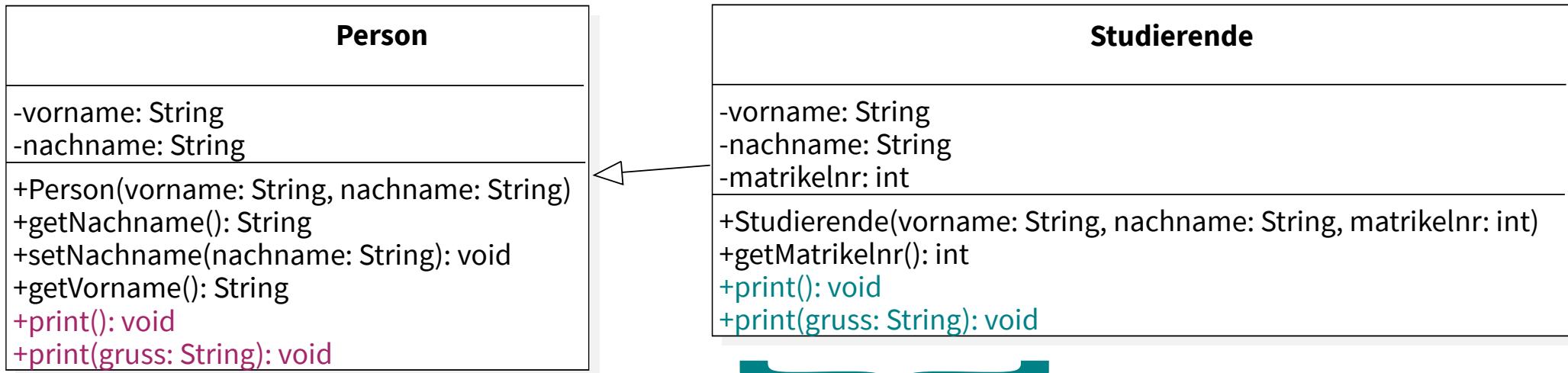
Eine Methode / einen Konstruktor mehrfach überladen



Mehrere Versionen der Methode/des Konstruktors mit unterschiedlichen Parameterlisten innerhalb einer Klasse bereitstellen.

# Überladen und Überschreiben in der OOP

## Überschreiben von Methoden – UML-Sicht



Die `print()`-Methode wurde in  
Klasse Person 2 x überladen

Die `print()`-Methode wurde in Klasse Studierende ebenfalls  
2 x überladen

Die `print()`-Methoden aus Klasse Person werden überschrieben!

# Überladen und Überschreiben in der OOP

## Überschreiben von Methoden – Grundlagen

- Überschreiben von Methoden innerhalb einer Kindklasse bedeutet: Wir implementieren Methoden neu, die bereits mit gleicher Signatur in der Elternklasse existieren.
- Beim Aufruf  
`objKindklasse.methode();`  
wird so die Neuimplementierung aus der Kindklasse aufgerufen.
- →Die Implementierung aus der Elternklasse wird überlagert bzw.  
**überschrieben!**

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überschreiben von Methoden – Klasse Studierende

```
1 public class Studierende extends Person {  
2  
3     private int matrikelnr;  
4  
5     // voll qualifizierter Konstruktor  
6     public Studierende(String vorname,  
7                         String nachname, int matrikelnr) {  
8         // Aufruf Superklassen-Konstruktor  
9         super(vorname, nachname);  
10        this.matrikelnr = matrikelnr;  
11    } // end constructor
```

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überschreiben von Methoden – Klasse Studierende

```
12 // getter fuer Matrikelnr.  
13 public int getMatrikelnr() {  
14     return matrikelnr;  
15 } // end method
```

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überschreiben von Methoden – Klasse Studierende

```
16 public void print() {  
17     // Aufruf print-Methode aus Superklasse Person  
18     super.print(); ←  
19     // eigener printout  
20     System.out.println("Meine Matrnr: "+matrikelnr);  
21 } // end method
```

- `print()` überschreibt die parameterlose Version aus Superklasse `Person`.
- Über die `super`-Referenz rufen wir zuerst die `print()`-Methode der Superklasse auf.

# Überladen und Überschreiben in der OOP

Alternative zur `print()`-Methode auf der vorigen Seite

## Java-Sicht – Überschreiben von Methoden – Klasse Studierende

```
16 public void print() {  
17     // Kann aber auch vollstaendig mit eigenen printouts  
18     // geloest werden:  
19     System.out.println("Ich heisse: " +this.getVorname()  
20             + " " + this.getNachname()); ←  
21     System.out.println("Meine Matrnr: "+matrikelnr);  
22 } // end method
```

- Auch hier: `print()` überschreibt die parameterlose Version aus Superklasse `Person`.
- Aber: ohne Zugriff auf `print()` der Superklasse!

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überschreiben von Methoden – Klasse Studierende

```
22 public void print(String gruss) {  
23     // Aufruf print-Methode aus Superklasse Person  
24     super.print(gruss); ←  
25  
26     // eigener printout  
27     System.out.println("Meine Matrnr lautet: "+matrikelnr);  
28 } // end method  
29 } // end class
```

- `print(...)` überschreibt die Version mit String-Parameter aus Superklasse `Person`.
- Über die `super`-Referenz rufen wir zuerst die `print(...)`-Methode der Superklasse auf.

# Überladen und Überschreiben in der OOP

## Java-Sicht – Überschreiben von Methoden – Wirkungsweise bei Objekten

```
1 Person p1 = new Person("Hugo", "Helmchen");  
2 p1.print("Moin");
```



→ Aufruf der Methode aus der Elternklasse

# Überladen und Überschreiben in der OOP

Java-Sicht – Überschreiben von Methoden – Wirkungsweise bei Objekten

```
1 Studierende s1 = new Studierende("Hugo", "Meier", 4711);  
2 s1.print("Servus");
```



→ Aufruf der Methode aus der Kindklasse

# Überladen und Überschreiben in der OOP

Java-Sicht – Überschreiben von Methoden – Wirkungsweise bei Objekten

```
1 Person s2 = new Studierende("Erna", "Eckert", 4712);  
2 s2.print("Servus");
```



→ Auch hier: Aufruf der Methode aus der Kindklasse

# Überladen und Überschreiben in der OOP

Überschreiben von Methoden – Zusammenfassung

Kindklassen können Methoden aus der Elternklasse **überschreiben**:

Dies geschieht durch Neuimplementierung von Methoden der Elternklasse in der Kindklasse.

# Abstrakte Klassen

## Abstrakte Klassen – was ist das und wozu wird's benötigt?

- Es gibt Softwarearchitekturen, in denen man „erzwingen“ will, dass von der obersten Superklasse kein Objekt erzeugt werden kann, sondern nur von einer Kindklasse.
- Es gibt Softwarearchitekturen, in denen man „erzwingen“ will, dass in den Kindklassen bestimmte Methoden auf jeden Fall implementiert werden.
- Wenn „wichtige“ Methoden in einem Softwareentwurf „vergessen“ werden, so wird die Software später fehlerhaft laufen.
- Diese Methoden werden in der Superklasse **nur als Methodenkopf deklariert**, nicht aber implementiert →**abstrakte Methoden!**

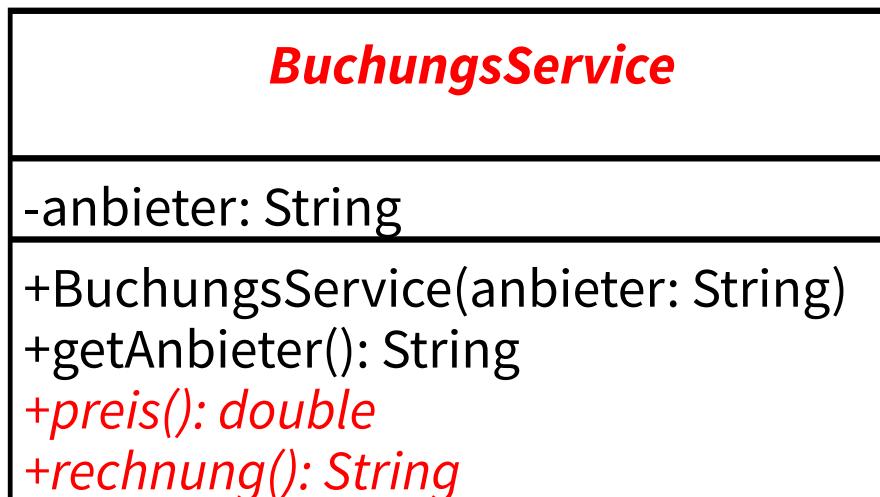
# Abstrakte Klassen

Abstrakte Klassen – was ist das und wozu wird's benötigt?

- Abstrakte Klassen werden am Klassenkopf mit dem Schlüsselwort `abstract` gekennzeichnet.
- Abstrakte Klassen können abstrakte Methoden enthalten (müssen aber nicht!)
- Von abstrakten Klassen können **keine Objekte gebildet werden**.

# Abstrakte Klassen – Darstellung in der UML

## Darstellungsmöglichkeit 1



Klassenname ***kursiv***:  
***Abstrakte Klasse***

Methodenname ***kursiv***:  
***Abstrakte Methode***

- Die UML kennt zwei Darstellungsmöglichkeiten für abstrakte Klassen.
- Entweder werden ***Klassename*** und ***abstrakte Methoden kursiv*** geschrieben, ...

# Abstrakte Klassen – Darstellung in der UML

## Darstellungsmöglichkeit 2

BuchungsService <i>{abstract}</i>
-anbieter: String
+BuchungsService(anbieter: String)
+getAnbieter(): String
+preis(): double <i>{abstract}</i>
+rechnung(): String <i>{abstract}</i>

Zusicherung **{abstract}**  
unter dem Klassennamen:  
**Abstrakte Klasse**

Zusicherung **{abstract}**  
hinter der Methoden-  
deklaration:  
**Abstrakte Methode**

- ... oder die Zusicherung **{abstract}** wird unter den Klassennamen und
- neben die Methodenköpfe geschrieben.

# Abstrakte Klassen – Darstellung in Java

## Abstrakte Klassen in Java – Syntax

```
1 [weitere Modifizierer] abstract class Klassename {  
2   [Attribute]  
3   [Konstruktoren]  
4   [Methoden]  
5   [Modifizierer] abstract returnType methodename() ;  
6 }
```

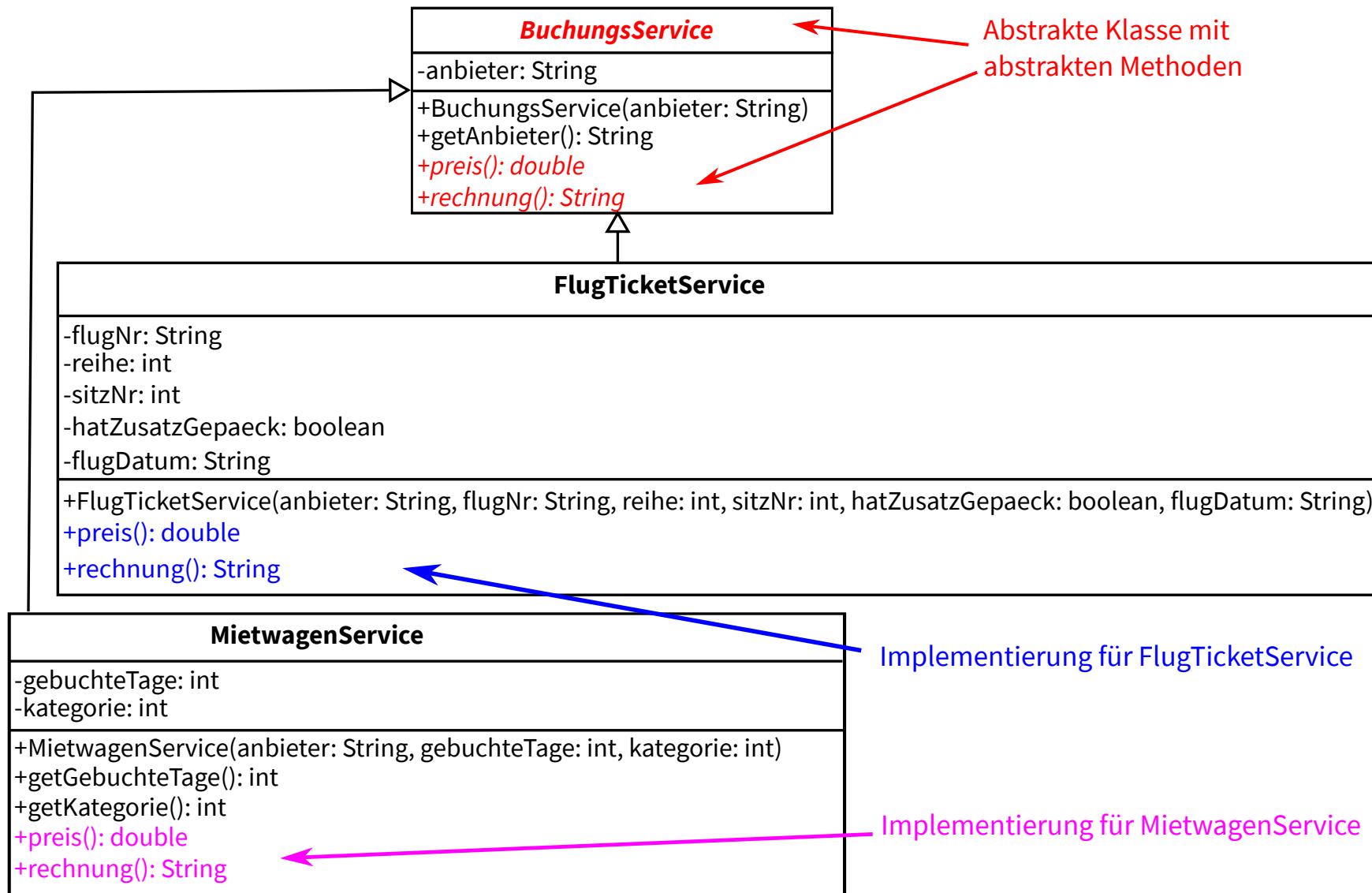
- Schlüsselwort **abstract** vor dem Schlüsselwort **class**.
- Schlüsselwort **abstract** vor dem **Methodenkopf**.
- Semikolon **;** hinter dem **Methodenkopf**.
- **Kein Methodenrumpf**, da abstrakte Methoden erst in den abgeleiteten Klassen implementiert werden.

# Abstrakte Klassen – Anwendungsbeispiel

## Buchungs-Service mit abstrakter Superklasse – Beschreibung

- Der Service muss bezahlt werden – es muss ein **Preis** berechnet werden  
→Methode **preis()**.
- Es soll eine **Rechnung** für den Service erstellt werden können –  
→Methode **rechnung()**.
- Beide Methoden sind davon abhängig, **welche Art Service** wir vor uns haben:
  - Flugbuchung,
  - Mietwagenbuchung,etc.
- Daher können diese Methoden noch nicht in der Elternklasse implementiert werden.

# Abstrakte Klassen – Anwendungsbeispiel – UML-Darstellung



# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

## Abstrakte Klasse *BuchungsService*

```
1 public abstract class BuchungsService {  
2  
3     private String anbieter; // Name des Anbieters  
4  
5     public BuchungsService(String anbieter) {  
6         this.anbieter = anbieter;  
7     } // end constructor  
8  
9     public String getAnbieter() {  
10        return anbieter;  
11    } // end method
```

- Abstrakte Klasse enthält Anbiaternamen als String,
- voll qualifizierten Konstruktor, getter-Methode, sowie ...

# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

## Abstrakte Klasse BuchungsService

```
12 // Preisberechnung muss in der Kindklasse
13 // implementiert werden!
14 public abstract double preis();  

15
16 // Rechnungserstellung muss in der Kindklasse
17 // implementiert werden!
18 public abstract String rechnung();  

19
20 } // end class
```

- ... 2 abstrakte Methoden

# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

## Abgeleitete Klasse *MietwagenService* – Attribute und Konstruktor

```
1 public class MietwagenService extends BuchungsService {  
2  
3     private int gebuchteTage;  
4     private int kategorie;  
5  
6     public MietwagenService(String anbieter,  
7                             int gebuchteTage, int kategorie) {  
8         // Anbietername an Superklassenkonstruktor geben  
9         super(anbieter);  
10        this.gebuchteTage = gebuchteTage;  
11        this.kategorie = kategorie;  
12    } // end constructor
```

# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

## Abgeleitete Klasse *MietwagenService* – getter-Methoden

```
13  public int getGebuchteTage() {  
14      return gebuchteTage;  
15  } // end method  
16  
17  public int getKategorie() {  
18      return kategorie;  
19  } // end method
```

# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

Abgeleitete Klasse *MietwagenService* – Implementierung der Methode *preis()*

```
20 @Override public double preis () {  
21     double preis = 0.0;  
22     if (kategorie == 0) {  
23         preis = 35.50 * gebuchteTage;  
24     }  
25     else if (kategorie >= 1 ) {  
26         preis = 43.20 * gebuchteTage;  
27     }  
28     return preis;  
29 } // end method
```

**Annotation**  
**@Override**

# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

## Klasse *MietwagenService* – Implementierung der Methode *rechnung()*

```
30  @Override public String rechnung() {  
31      String rechnungsString = "";  
32      rechnungsString += "Rechnung von "+getAnbieter() + "\n";  
33      rechnungsString += "Gebuchte Tage: "+gebuchteTage + "\n";  
34      rechnungsString += "Endpreis: "+preis() + "EUR" +"\n";  
35  
36      return rechnungsString;  
37  } // end method  
38 } // end class
```

Achtung: Die Mehrwertsteuer wurde in dieser Rechnung vernachlässigt!

# Einschub: Annotationen am Beispiel @Override

## Was ist eine Annotation?

- Metainformation zu Deklarationen von
  - Klassen,
  - Methoden,
  - Attributen
- Beginnt immer mit dem Zeichen @, gefolgt von dem Annotationsnamen
- Werden von einem Bestandteil des Compilers ausgewertet
- Unterschied zu Modifizierern: In Java können auch eigene Annotationen implementiert werden (späteres Kapitel ...).

# Einschub: Annotationen am Beispiel @Override

## Häufig genutzte Annotationen

Name der Annotation	Bedeutung
@Override	Wird am Kopf von Methoden angegeben, die eine Methode einer Elternklasse überschreiben – insbesondere bei der Implementierung von abstrakten Methoden, die von einer abstrakten Elternklasse oder von einem Interface vorgegeben werden. (Nicht zwingend, aber empfehlenswert! Compiler kontrolliert so, ob wir auch die richtige Methode überschreiben!)
@Deprecated	Wird bei veralteten Methoden oder Attributen angegeben, die man aus Kompatibilitätsgründen noch nicht löschen will.

# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

Nutzung MietwagenService in einer anwendenden *main()*-Klasse  
(Codefragment)

```
1 String firmenName = "MietwagenService Rostlaube";
2 int tage = 2;
3 int kategorie = 1;
4
5 // Buchung eines Mietwagens
6 BuchungsService buchung1 =
7     new MietwagenService(firmenName, tage, kategorie);
8
9 double preis = buchung1.preis();
10 String rechnung = buchung1.rechnung();
11 System.out.println(rechnung);
```

# Abstrakte Klassen – Anwendungsbeispiel – Java-Darstellung

Was NICHT geht: Objekterstellung von der abstrakten Klasse

```
1 String firmenName = "MietwagenService Rostlaube";  
2 BuchungsService buchung1 =  
3 new BuchungsService(firmenName);
```

- Objekterstellung von abstrakten Klassen ist **nicht möglich**.
- Welcher Code sollte hier auch für die abstrakten Methoden aufgerufen werden?!

# Zusammenfassung

## Wir merken uns zu Abstrakten Klassen

- Es gibt Softwarearchitekturen, in denen man „erzwingen“ will, dass in den Kindklassen bestimmte Methoden auf jeden Fall implementiert werden.
- Derartige Methoden werden in der Elternklasse als **abstract** vereinbart:  
`datentyp methodName(...);`
- Eine Klasse mit abstrakten Methoden muss ebenfalls als **abstract** vereinbart werden.
- Die konkrete Methodenimplementierung erfolgt in einer **abgeleiteten Klasse**.
- Von einer abstrakten Klasse können wir **keine Objekte instanziieren!**

# Interfaces

## Interfaces – was ist das und wozu wird's benötigt?

- i. d. R. Abstrakte Vorgaben von Methodenschnittstellen für abgeleitete Klassen.
- Wenn eine Klasse von einem Interface „erbt“, so sagt wir: Die Klasse **implementiert das Interface**.
- Wir benötigen Interfaces, wenn wir erzwingen wollen, dass bestimmte Methoden implementiert werden, aber keine normale Klasse schreiben wollen.
- Eine Klasse kann **mehrere Interfaces implementieren**, aber nur von einer Klasse abgeleitet sein! → Interfaces können also indirekt Mehrfachvererbung möglich machen.

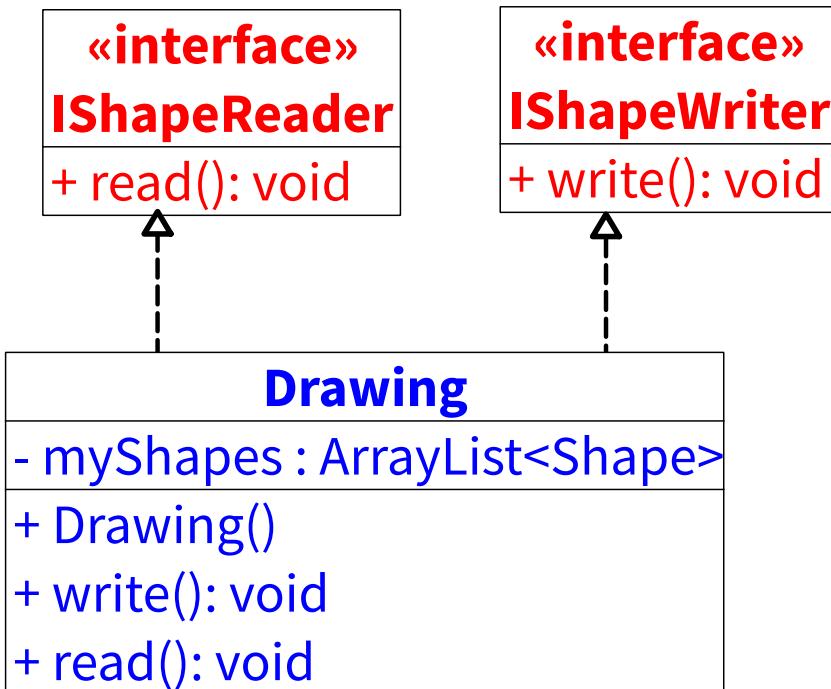
# Interfaces

## Interfaces – was ist das und wozu wird's benötigt?

- Bis Java 7 gilt: Interfaces können keine Methodenimplementierungen enthalten, sondern nur:
  - Rein abstrakte Methoden (Methodenköpfe). Diese sind automatisch **public**
  - Konstanten. Diese sind automatisch **public static final**.
- Ab Java 8 gilt:
  - Interfaces können Default-Implementierungen ihrer Methodenvorgaben enthalten.
  - Dies wird durch das Schlüsselwort **default** vor dem Rückgabetyp des Methodenkopfes kennlich gemacht.

# Interfaces

## Interfaces – Darstellung in der UML



Zwei Interfaces, die jeweils eine Methode vorgeben

Eine Klasse, die beide Interfaces implementiert

→ Vererbungspfeil ist gestrichelt!

# Interfaces

## Interfaces – Syntax in Java

```
1 [weitere Modifizierer] interface InterfaceName1 {  
2   [public static final Attribute]  
3  
4   [Modifizierer] returnTyp methodename() ;  
5   // Weitere Methodenprototypen  
6 }
```

# Interfaces

## Interfaces – Syntax in Java bei implementierenden Klassen

```
1 [weitere Modifizierer] class Klassename
2     implements InterfaceName1,
3             InterfaceName2, ... {
4     [Attribute]
5     [Konstruktoren]
6     [Methoden]
7     [Implementierung der Methoden, die von den
8      Interfaces vorgegeben sind]
9 }
```

→ Klasse „erbt“ von den implementierten Interfaces

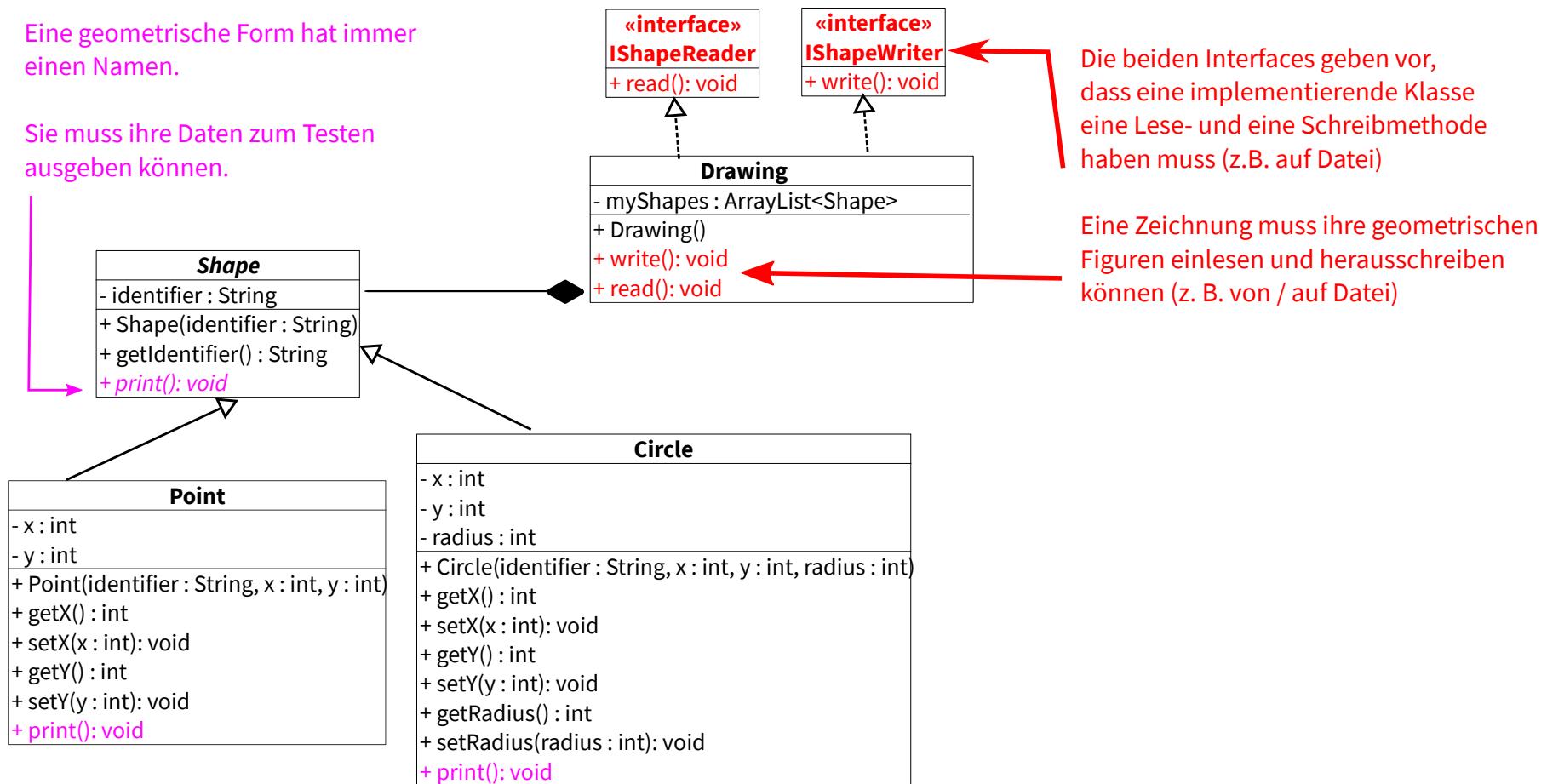
# Interfaces – Anwendungsbeispiel 1 – UML-Darstellung

## Zeichnung mit verschiedenen geometrischen Formen

Die abstrakte Klasse **Shape** gibt vor:

Eine geometrische Form hat immer einen Namen.

Sie muss ihre Daten zum Testen ausgeben können.



# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Abstrakte Klasse Shape

```
1 // Superklasse fuer geometrische Figuren
2 public abstract class Shape {
3
4     // Eine geometrische Form hat immer
5     // einen Bezeichner / Namen, zB. kreis1
6     private String identifier;
7
8     public Shape(String identifier) {
9         this.identifier = identifier;
10    } // end constructor
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Abstrakte Klasse Shape

```
11  public String getIdentifier() {  
12      return identifier;  
13  } // end method  
14  
15  // Abstrakte Methode: Eine geometrische Form soll  
16  // immer ihre Daten ausgeben koennen  
17  public abstract void print();  
18 } // end class
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse Point

```
1 // Punkt in einem 2-dimensionalen, ganzzahligen
2 // Koordinatensystem (zB. Bildschirm)
3 public class Point extends Shape {
4     private int x;
5     private int y;
6
7     public Point(String identifier, int x, int y) {
8         super(identifier); // identifier an Superklasse
9         this.x = x;
10        this.y = y;
11    } // end constructor
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse Point

```
12 // getter / setter
13 public int getX() { return x; }

14
15 public void setX(int x) { this.x = x; }

16
17 public int getY() { return y; }

18
19 public void setY(int y) { this.y = y; }
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse Point

```
20 @Override
21 public void print() {
22     System.out.println(
23         "Dies ist ein Punkt mit dem Bezeichner" +
24         this.getIdentifier());
25     System.out.println("x = " + x);
26     System.out.println("y = " + y);
27
28 } // end method print()
29 } // end class
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse Circle

```
1  public class Circle extends Shape {  
2  
3      private int x;  
4      private int y;  
5      private int radius;  
6  
7      public Circle(String identifier, int x, int y,  
8                      int radius) {  
9          super(identifier);  
10         this.x = x;  
11         this.y = y;  
12         this.radius = radius;  
13     } // end constructor
```

# Interfaces – Anwendungsbeispiel 1– Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse Circle

```
14 // getter / setter
15 public int getX() { return x; }

16
17 public void setX(int x) { this.x = x; }

18
19 public int getY() { return y; }

20
21 public void setY(int y) { this.y = y; }

22
23 public int getRadius() { return radius; }

24
25 public void setRadius(int radius) { this.radius = radius;
26 } // end method
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse Circle

```
27 @Override
28 public void print() {
29     System.out.println(
30         "Dies ist ein Kreis mit dem Bezeichner"
31         + this.getIdentifier());
32     System.out.println("Mittelpunkt x = " + x);
33     System.out.println("Mittelpunkt y = " + y);
34     System.out.println("Radius = " + radius);
35 } // end method print()
36 } // end class
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Interface *IShapeReader*

```
1 public interface IShapeReader {  
2  
3     // Shape einlesen (zB von Tastatur oder Datei)  
4     public void read();  
5 } // end interface
```

→Interface gibt der implementierenden Klasse eine Einlesemethode vor

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Interface *IShapeWriter*

```
1 public interface IShapeWriter {  
2     // Shape ausgeben (zB auf Konsole oder Datei)  
3     public void write();  
4 }
```

→Interface gibt der implementierenden Klasse eine Herausschreibemethode vor

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse Drawing

```
1 import java.util.ArrayList;
2 import java.util.Scanner;
3
4 public class Drawing implements IShapeReader, IShapeWriter {
5
6     private ArrayList<Shape> myShapes; ← Liste für Shapes (Point-  
7                                         und Circle-Objekte)  
8
9     public Drawing() {                                     → Polymorphie
10        // Lege leere ArrayList fuer Shapes
11        // an. (Hier konkret: Point oder Circle)
12        myShapes = new ArrayList<Shape>();
13    } // end constructor
```

→Klasse implementiert die Interfaces *IShapeReader* und *IShapeWriter*

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse *Drawing* – Methode *write()*/Variante 1

```
13  @Override public void write( ) {  
14      // Rufe von allen Shapes die print()-Methode  
15      // auf --> write() erfolgt auf Konsole  
16      for (int i=0; i<myShapes.size(); i++) {  
17          myShapes.get(i).print();  
18      } // end for  
19  } // end implementation of method write()
```

- Implementierung der Methodenvorgabe von Interface *IShapeWriter*
- Schreiben erfolgt auf die Konsole

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse *Drawing* – Methode *write()*/Variante 2

```
13  @Override public void write( ) {  
14      // Rufe von allen Shapes die print()-Methode  
15      // auf --> write() erfolgt auf Konsole  
16      for (Shape s : myShapes) {  
17          s.print();  
18      } // end for  
19  } // end implementation of method write()
```

→ Implementierung der Methodenvorgabe von Interface *IShapeWriter*  
→ Schreiben erfolgt auf die Konsole

# Interfaces – Anwendungsbeispiel 1– Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse *Drawing* – Methode *read()*/Teil1

```
20  @Override public void read() {  
21      // read() erfolgt von Tastatur!  
22      Scanner tastatur = new Scanner(System.in);  
23      String eingabe = "";  
24      String s ; // Name der Shape  
25      int x,y; // x/y-Koordinate der Shape  
26  
27      do {  
28          // Nutzer gibt gewünschte Objektart ein  
29          System.out.print("Welche Objektart? ");  
30          eingabe = tastatur.next();
```

- Implementierung der Methodenvorgabe von Interface *IShapeReader*
- Einlesen erfolgt von Tastatur in *do – while*-Schleife

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse *Drawing* – Methode *read()*/Teil2

```
31 // Je nach Objektart -- andere Objekterzeugung
32 switch (eingabe) {
33
34     case "point":
35         System.out.print("Bezeichner: ");
36         s = tastatur.next();
37         System.out.print("x-Wert: ");
38         x = tastatur.nextInt(); ← Nutzer gibt Daten für
39         System.out.print("y-Wert: ");
40         y = tastatur.nextInt(); den gewünschten Punkt
41
42         Point p = new Point(s,x,y);
43         myShapes.add(p); break; ← Shapes-Liste bekommt
                                Point hinzugefügt
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse *Drawing* – Methode *read()*/Teil3

```
44     case "circle":  
45         // Daten fuer Kreis einlesen  
46         System.out.print("Bezeichner: ");  
47         s = tastatur.next();  
48         System.out.print("Mittelpunkt x-Wert: ");  
49         x = tastatur.nextInt();  
50         System.out.print("Mittelpunkt y-Wert: ");  
51         y = tastatur.nextInt();  
52         System.out.print("Radius: ");  
53         int radius = tastatur.nextInt();  
54  
55         Circle c = new Circle(s,x,y, radius);  
56         myShapes.add(c); // Kreis in Liste haengen
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – Klasse *Drawing* – Methode *read()*/Teil4

```
57     case "ende": break;
58
59     default: break;
60
61 }while(!eingabe.equals("ende")); // end do-while
62
63 } // // // end implementation of method read()
64 } // end class
```

# Interfaces – Anwendungsbeispiel 1 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 1 – *main()*-Klasse

```
1 public class ShapesMain1 {  
2     public static void main(String[] args) {  
3         Drawing d = new Drawing();  
4         d.read(); // Shapes vom Nutzer einlesen  
5         d.write(); // Shapes auf die Konsole ausgeben  
6     }  
7 }
```

# Interfaces – Default-Implementierung von Methoden

## Wozu Default-Implementierung von Interface-Methoden?

- Bequemlichkeit für diejenigen, die das Interface nutzen!
- Nicht jede „Mini-Methode“ des Interfaces muss dann implementiert werden.
- Sehr häufig sind diese Default-Implementierungen leer – nämlich dann, wenn die Methode nicht in jeder implementierenden Klasse benötigt wird.

# Interfaces – Default-Implementierung von Methoden

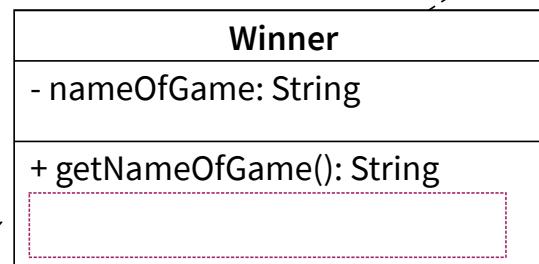
## Interfaces – Syntax bei der Default-Implementierung von Methoden

```
1 public interface Ixyz {  
2  
3     default returnType method1 (...) {  
4         // code of implementation  
5     } // end of implementation  
6  
7     // other method prototypes ...  
8  
9 } // end interface
```

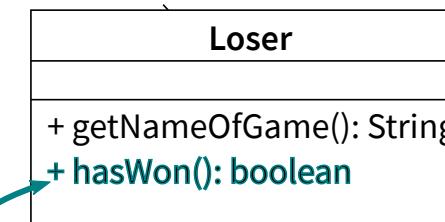
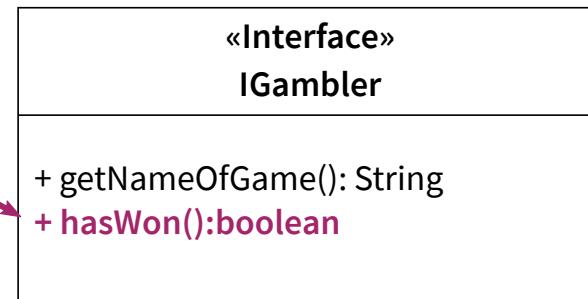
# Interfaces – Anwendungsbeispiel 2 – Default-Implementierung

## Interfaces – Anwendungsbeispiel 2 – UML-Darstellung

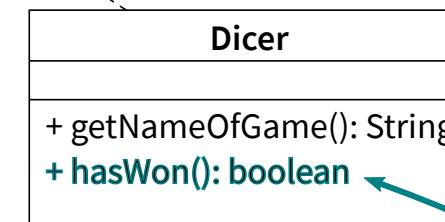
Methode `hasWon()` hat eine default-Implementierung, die immer true zurückgibt.



Klasse Winner hat keine eigene `hasWon()`-Methode, sondern übernimmt die Default-Implementierung aus dem Interface `IGambler`



Loser verliert immer -- seine `hasWon()`-Methode muss also die Default-Methode aus `IGambler` überschreiben



Dicer (Würfelspieler) gewinnt nur, wenn er eine "6" würfelt. Seine `hasWon()`-Methode muss ebenfalls die aus `IGambler` überschreiben!

# Interfaces – Anwendungsbeispiel 2 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 2 – Interface *IGambler*

```
1 public interface IGambler {  
2  
3     String getNameOfGame(); // Name des Spiels  
4  
5     // Wenn nichts anderes implementiert wird,  
6     // gewinnt der Spieler immer!  
7     default boolean hasWon() {  
8         return true;  
9     } // end of method implementation  
10 } // end interface
```

# Interfaces – Anwendungsbeispiel 2 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 2 – Klasse Winner

```
1 public class Winner implements IGambler {  
2  
3     private final String nameOfGame =  
4         "Winner takes all!";  
5  
6     @Override  
7     public String getNameOfGame() {  
8         return nameOfGame;  
9     } // end method  
10  
11    // hasWon () muss nicht neu implementiert werden!  
12 } // end class
```

# Interfaces – Anwendungsbeispiel 2 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 2 – Klasse Loser

```
1 public class Loser implements IGambler {  
2  
3     @Override  
4     public String getNameOfGame() {  
5         return this.getClass().getName();  
6     }  
7  
8     // Loser verliert immer! Neuimplementation!  
9     @Override  
10    public boolean hasWon() { return false; }  
11 } // end class
```

# Interfaces – Anwendungsbeispiel 2 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 2 – Klasse Dicer – Teil 1

```
1 import java.util.Random;  
2  
3 public class Dicer implements IGambler {  
4  
5     private int value; // gewuerfelter Wert  
6  
7     @Override  
8     public String getNameOfGame() {  
9         return this.getClass().getName();  
10    } // end method getNameOfGame()
```

# Interfaces – Anwendungsbeispiel 2 – Java-Darstellung

## Interfaces – Anwendungsbeispiel 2 – Klasse Dicer – Teil 2

```
11 @Override
12 public boolean hasWon() {
13
14     Random r = new Random(); // Wuerfel erzeugen
15
16     // Einmal wuerfeln: nextInt() liefert hier eine
17     // Ganzzahl zwischen 0 und kleiner 6 -- daher +1
18     value = r.nextInt(6) + 1;
19
20     // Gewonnen, wenn eine "6" gewuerfelt
21     return (value==6);
22 } // end method hasWon()
23 } // end class
```

# Interfaces – Anwendungsbeispiel 2 zu Default-Implementierung

## Zusammenfassung

- Das Interface *IGambler* modelliert das Gewinnverhalten eines Spielers
- Die Methode *hasWon()* ist hier als Default-Implementierung konzipiert:
  - Wenn nichts anderes in der implementierenden Klasse steht, dann liefert die Default-Implementierung immer *true* zurück.
  - Der Spieler gewinnt in diesem Fall also immer.
- Die Klasse *Winner* implementiert das Interface und nutzt die Default-Implementierung von *hasWon()*.
- Die Klassen *Loser* und *Dicer* (Würfelspieler) haben jeweils eine eigene Implementierung von *hasWon()*.

# Interfaces – Anwendungsbeispiel 2 zu Default-Implementierung

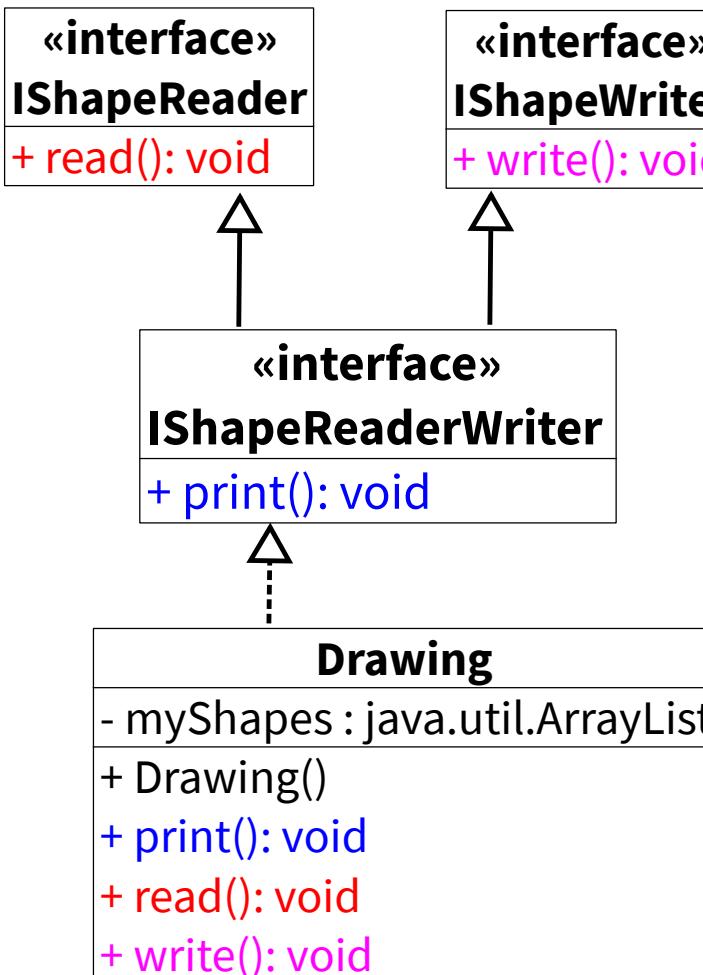
## Übung

Schreiben Sie eine Klasse *TestGame*, welche

- jeweils einen Spieler für jede der 3 Klassen *Winner*, *Loser* und *Dicer* erzeugt.
- Lassen Sie jeden Spieler 10 Runden spielen und geben Sie aus, ob er gewonnen hat oder nicht!

# Interfaces – mehrstufige Vererbungshierarchien – UML

## Interfaces können von anderen Interfaces Vorgaben erben!



Das Interface *IShapeReaderWriter* erbt von zwei weiteren Interfaces.

Die Klasse Drawing implementiert das Interface *IShapeReaderWriter*

Sie muss alle Methoden des Interfaces implementieren -- auch die ererbten.

# Interfaces – mehrstufige Vererbungshierarchien – Java

## Elterninterface *IShapeReader*

```
1 public interface IShapeReader {  
2  
3     // Shape einlesen (zB von Tastatur oder Datei)  
4     public void read();  
5 }
```

# Interfaces – mehrstufige Vererbungshierarchien – Java

## Elterninterface *IShapeWriter*

```
1 public interface IShapeWriter {  
2  
3     // Shape ausgeben (zB auf Konsole oder Datei)  
4     public void write();  
5 }
```

# Interfaces – mehrstufige Vererbungshierarchien – Java

## Kindinterface *IShapeReaderWriter*

```
1 public interface IShapeReaderWriter
2     extends IShapeReader, IShapeWriter {
3
4     public void print(); // nur fuer Ausgabe auf Bildschirm
5 }
```

→ erbt Vorgaben von 2 Elterninterfaces

→ gibt selbst eine eigene Methode *print()* vor.

# Interfaces – Zusammenfassung

Wir merken uns:

- Interfaces sind rein abstrakte Vorgaben von Methodenschnittstellen für abgeleitete Klassen.
- Wenn eine Klasse von einem Interface „erbt“, so sagt wir: Die Klasse **implementiert das Interface**, denn: Sie implementiert die Methoden, die das Interface vorgibt.
- Eine Klasse kann mehrere Interfaces implementieren.

# Interfaces – Zusammenfassung

Wir merken uns:

- Eine Implementierungsbeziehung ist eine **IST**-Beziehung.
- In unserem Shape-Beispiel also: Eine *Drawing* **IST** auch ein *IShapeReader* und ein *IShapeWriter*
- Ein Interface kann auch von anderen Interfaces erben – dies geschieht mit dem Schlüsselwort **extends**.

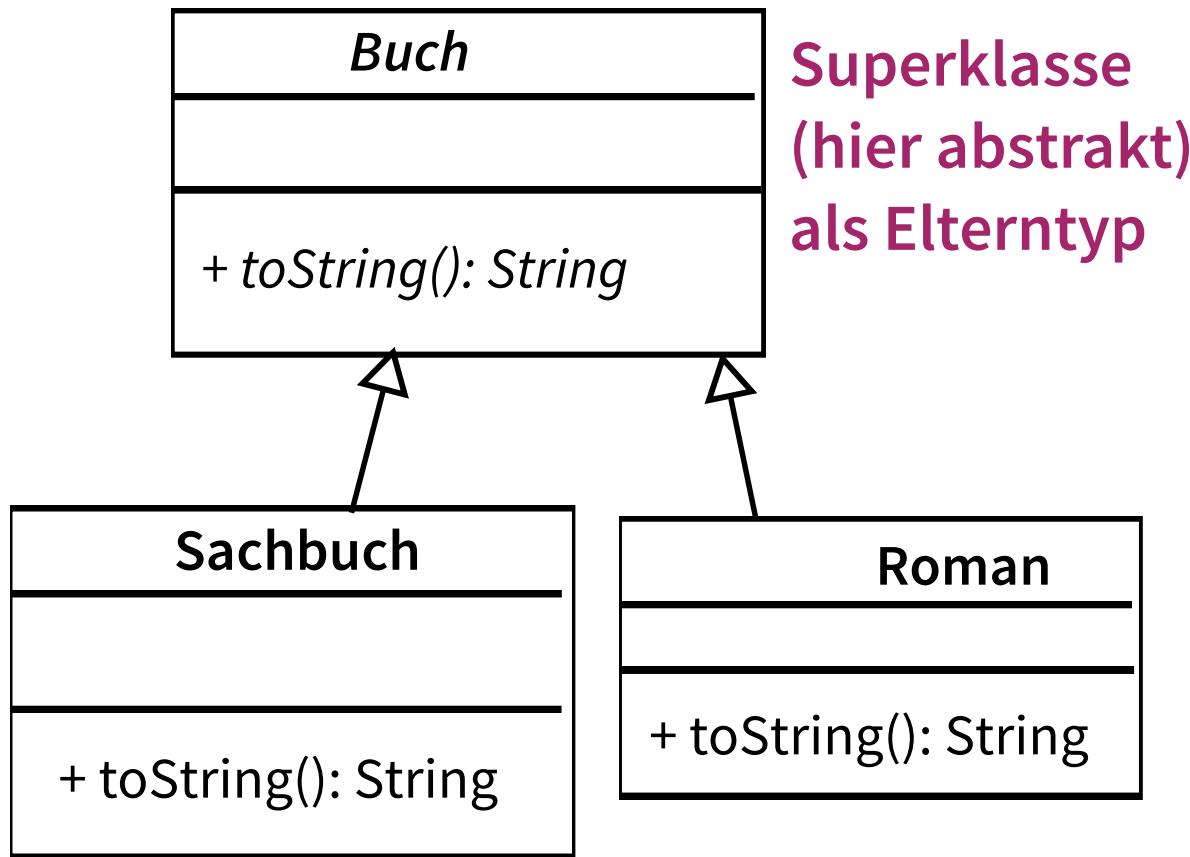
# Auswirkungen von Interfaces auf Polymorphie

Interfaces sind Supertypen!

- Ein Interface kann genau so als Supertyp verwendet werden wie eine Klasse.
- Das bedeutet: Wir können Variablen vom Typ eines Interfaces anlegen.
- Die Objekterzeugung erfolgt dann – ähnlich wie bei einer abstrakten Superklasse – mit einer der implementierenden Klassen.
- Auf den nächsten Folien folgt ein kleines Beispiel ...

# Auswirkungen von Interfaces auf Polymorphie

## Beispiel 1: Abstrakte Klasse als Elterntyp



# Auswirkungen von Interfaces auf Polymorphie

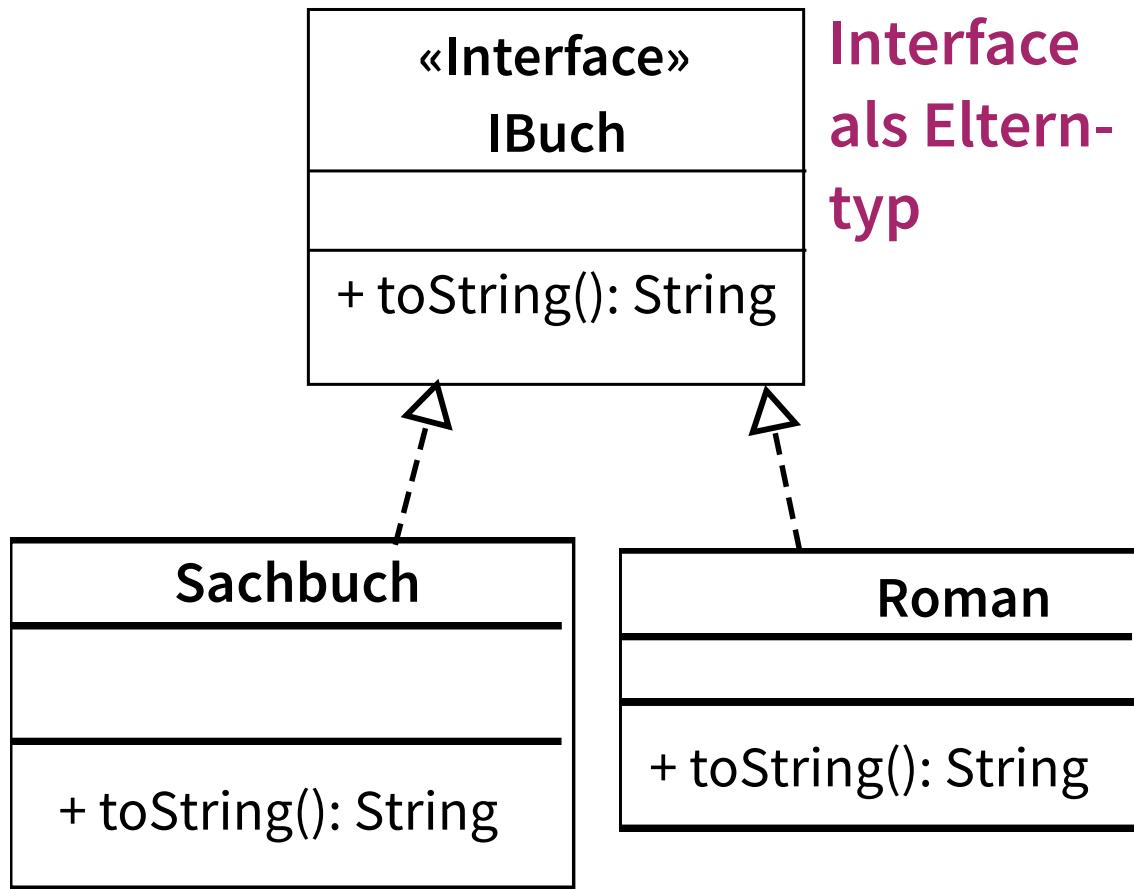
## Beispiel 1: Abstrakte Klasse als Elterntyp

Hier ist folgendes, wie gewohnt, möglich:

```
1 // Liste anlegen -- typisiert auf Elternklasse
2 ArrayList<Buch> liste = new ArrayList<Buch>();
3
4 // Der Liste ein Element vom Typ Kindklasse hinzufuegen
5 liste.add(new Sachbuch());
```

# Auswirkungen von Interfaces auf Polymorphie

## Beispiel 2: Interface als Elterntyp



# Auswirkungen von Interfaces auf Polymorphie

## Beispiel 2: Interface als Elterntyp

Hier ist folgendes, wie gewohnt, möglich:

```
1 // Liste anlegen -- typisiert auf Interface
2 ArrayList<IBuch> liste = new ArrayList<IBuch>();
3
4 // Der Liste ein Element vom Typ einer der
5 // implementierenden Klassen hinzufuegen
6 liste.add(new Sachbuch());
```

# Auswirkungen von Interfaces auf Polymorphie

Wir merken uns: Interfaces sind Supertypen!

- Ein Interface kann genau so als Supertyp verwendet werden wie eine Klasse.
- Das bedeutet: Wir können Variablen vom Typ eines Interfaces anlegen.
- Die Objekterzeugung erfolgt dann – ähnlich wie bei einer abstrakten Superklasse – mit einer der implementierenden Klassen.
- Auch Listen oder Arrays können auf ein Interface typisiert werden.
- Sie werden dann aber zur Laufzeit mit Objekten von Klassen befüllt, die dieses Interface implementieren!

# Der instanceof-Operator

## Problem bei polymorphen Objekten

- Häufig ist eine Referenzvariable auf den Elterntyp typisiert:  
Elternklasse variable;
- Zur Laufzeit erhält die Variable aber eine Referenz auf ein Objekt einer Kindklasse:  
`variable = new Kindklasse1();`  
(beispielsweise beim Einlesen aus einer Datei)
- Manchmal muss später festgestellt werden, zu welcher (Kind-)Klasse das Objekt einer solchen Referenz gehört.
- Hierzu stellt Java den Operator instanceof bereit.

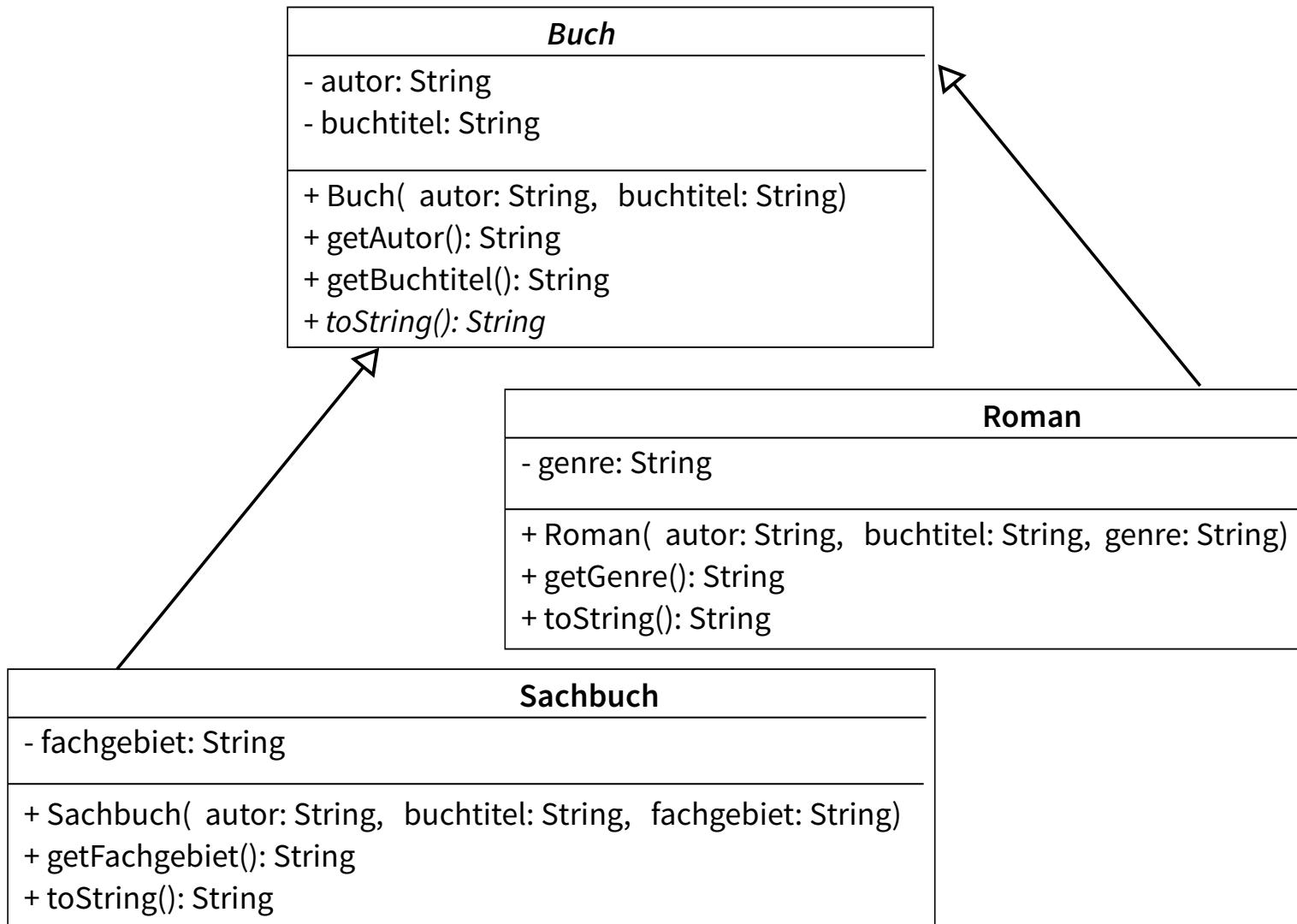
# Der instanceof-Operator

## Anwendung des instanceof-Operators

```
1 if (variable instanceof Kindklasse1) {  
2     // Aktion fuer ein Objekt von Kindklasse1  
3 }  
4 else if(variable instanceof Kindklasse2) {  
5     // Aktion fuer ein Objekt von Kindklasse2  
6 }  
7 else ...
```

# Der instanceof-Operator

## Anwendungsbeispiel: eine kleine Klassenhierarchie



# Der instanceof-Operator

Anwendungsbeispiel: Klasse VerarbeiteBuch

```
1 import java.util.Scanner;  
2 public class VerarbeiteBuch {  
3  
4     public void ausgabe( Buch b ) {
```

Formaler Parameter vom Typ der Elternklasse

# Der instanceof-Operator

## Anwendungsbeispiel: Klasse VerarbeiteBuch

```
5 // Ausgabe Überschrift
6 if (b instanceof Roman) {
7     System.out.println("Roman: ");
8 } // end if
9 else if (b instanceof Sachbuch) {
10    System.out.println("Sachbuch");
11 } // end else
12
13 // Ausgabe der Objektdaten als String
14 System.out.println(b.toString());
15 } // end method
16 } // end class
```

**Überprüfung: Welches Objekt steckt hinter der Referenzvariablen b?**

## Der instanceof-Operator

Anwendungsbeispiel: Klasse VerarbeiteBuch - Alternative Ausgabe

```
13     // Ausgabe der Objektdaten als String geht auch
14         // einfacher (Weil toString() die Standard-Kon
15             // Methode ist)
16     System.out.println(b);
17 } // end method
18 } // end class
```

**Überprüfung: Welches Objekt steckt hinter der Referenzvariablen b?**

# Der instanceof-Operator

Wir merken uns:

- Manchmal muss später festgestellt werden, zu welcher (Kind-)Klasse ein Objekt einer Elternklassen-Referenz gehört.
- Hierzu stellt Java den Operator `instanceof` bereit.
- Der Operator gibt einen boolean-Wert zurück: `true`, falls der `instanceof`-Vergleich zutrifft, `false` sonst.

Fakultät Informatik

# Programmierung 2

Exceptions

Vorwort

Einstieg in Java

Einstieg in die OOP: Klassen und Objekte/Instanzen

Die Klasse String

Graphische Darstellung von Klassen mit der UML – Teil 1

Vererbungsbeziehungen

Exceptions

## Was sind Exceptions?

Exceptions selbst erzeugen und werfen – throw und throws

Exceptions fangen – try – catch – finally

## Einstieg Streams

# Was sind Exceptions?

In Objektorientierten Programmiersprachen sind Exceptions ...

- ... Objekte spezieller Klassen, die in Fehler- oder Ausnahmebehandlungs-Situationen des Programms erzeugt werden.
- Sie
  - enthalten Informationen darüber, **wo im Code** der Fehler / die Ausnahmesituation entstanden ist, und
  - können sehr schnell an die oberste Aufruf-Ebene des Programm-Codes durchgereicht werden.
- Sie erlauben es den Programmcode aufzuteilen in
  - korrekt abgelaufenen Standard-Code (**try**-Klausel)
  - Code, der nur im Fehlerfall zum Auffangen des Fehlers durchlaufen wird (**catch**-Klausel)
  - Code, der **immer** durchlaufen wird, gleichgültig, ob ein Fehler aufgetreten ist oder nicht (**finally**-Klausel)

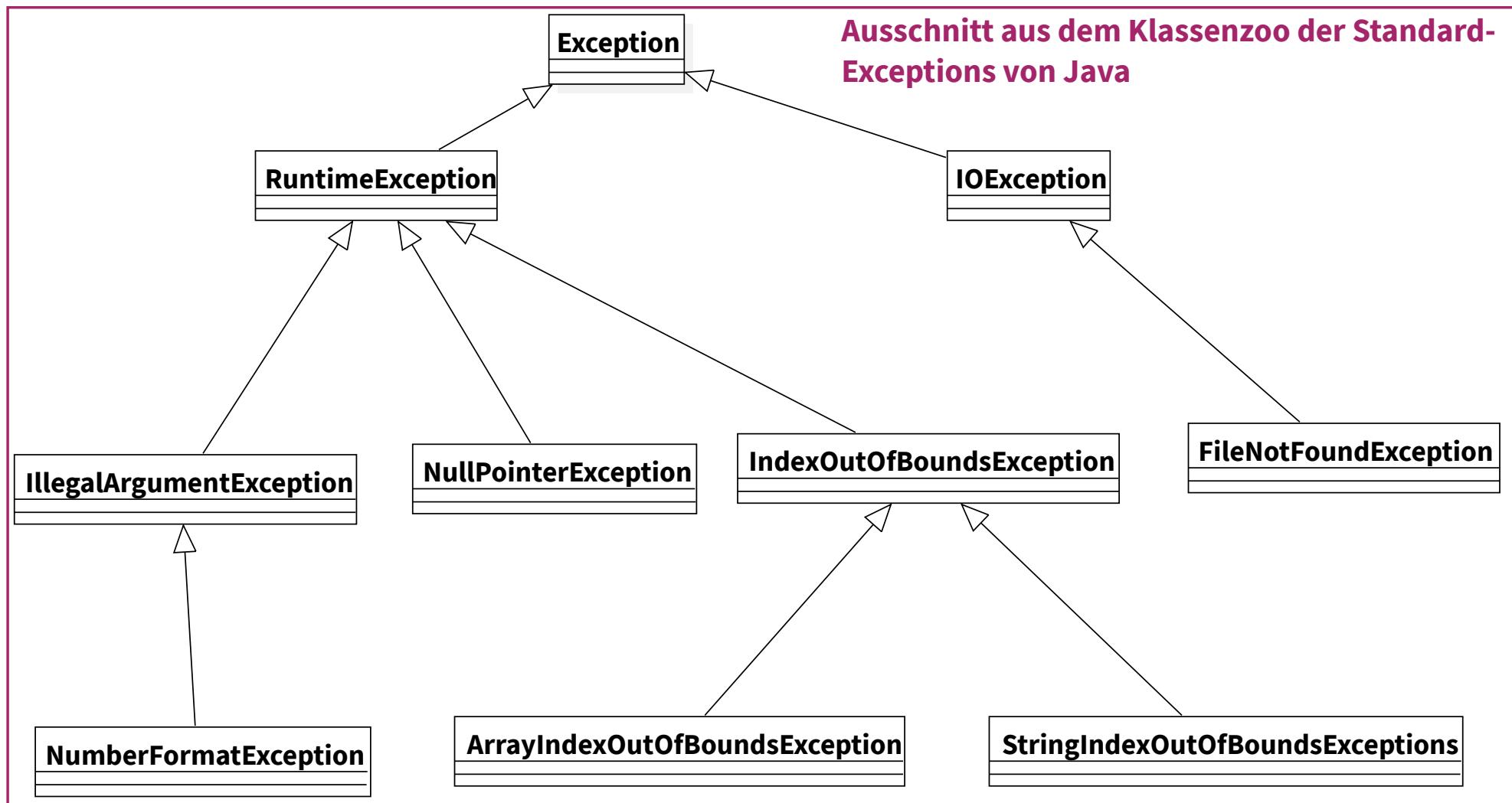
# Was sind Exceptions?

Exceptions können in Java ...

- ... als Objekte vorgefertigter Exception-Klassen selbst erzeugt werden, oder ...
- ... als Objekte eigener, selbst definierter Exception-Klassen erzeugt werden.
- Ein Exception-Objekt an den Aufrufer einer Methode weiterreichen heißt in der OOP auch „**Eine Exception werfen**“.
- Dies geschieht in Java mit `throw`.
- Eine Java-Methode kann auch ein in ihr entstandenes Exception-Objekt einfach an ihren Aufrufer weiterreichen. Dies wird an ihrem Kopf kenntlich gemacht mit der Klausel `throws`

# Was sind Exceptions?

Einige sehr bekannte, vorgefertigte Exception-Klassen in Java



# Exceptions selbst erzeugen und werfen – throw

## Syntax von throw

```
1 // Exception-Objekt erzeugen: Dem Konstruktor wird eine
2 // mögliche Fehlermeldung mit übergeben.
3 ExceptionType1 myException =
4     new ExceptionType1("Exception-Nachricht ...");
5
6 // Exception werfen
7 throw myException;
```

# Methoden, in denen Exceptions geworfen werden, ... ... benötigen die throws-Klausel

## Syntax von throws

```
1 returnType methodName(typ1 param1, typ2 param2,...)  
2           throws ExceptionType1,ExceptionType2,...  
3 {  
4     // Code der Methode ...  
5     ExceptionType1 ex1 = new ExceptionType1("Fehler 1");  
6     throw ex1;  
7     // weiterer Code ... weiterer throw ...  
8     // ggf. return-Statement  
9 }
```

# Exceptions selbst erzeugen und werfen

## Unterschied throw und throws

- `throw` wirft ein konkretes Exception-Objekt innerhalb des Methodenrumpfes
- `throws` ist dagegen eine Deklaration am Methodenkopf: Sie macht kenntlich, dass innerhalb dieser Methode bestimmte Typen von Exceptions geworfen werden können.

# Exceptions selbst erzeugen und werfen – Beispiel

## Klasse Zahlenformat – Teil 1

```
1 public class ZahlenFormat {  
2  
3     public int returnZahlWert(char ziffer)  
4             throws NumberFormatException {  
5         // gib entsprechenden int-Wert  
6         // zwischen 0 und 9 zurueck  
7         if( ziffer >= '0' && ziffer <= '9') {  
8             return ziffer - '0';  
9         }  
}
```

Methode gibt den Zahlenwert einer übergebenen Ziffer zurück

# Exceptions selbst erzeugen und werfen – Beispiel

## Klasse Zahlenformat – Teil 2

```
10      // ansonsten wirf Exception-Objekt
11  else {
12      NumberFormatException myExcept =
13          new NumberFormatException(
14              "Keine Ziffer!");
15      throw myExcept;
16  } // end else
17 } // end method
18 } // end class
```

Falls das übergebene Zeichen keine Ziffer war: Methode wirft Exception vom Typ *NumberFormatException*

## Exceptions selbst erzeugen und werfen – Beispiel

### main()-Klasse MainEx1 – Teil 1

```
1 import java.util.Scanner;  
2  
3 public class MainEx1 {  
4  
5     public static void main(String[] args)  
6         throws NumberFormatException {  
7  
8     Scanner tastatur = new Scanner(System.in);  
9  
10    ZahlenFormat worker = new ZahlenFormat();
```

Da die main()-Methode später eine Methode aufruft, die eine *NumberFormatException* werfen kann, so muss sie diese ebenfalls verarbeiten – hier mit throws.

# Exceptions selbst erzeugen und werfen – Beispiel

## main()-Klasse MainEx1 – Teil 2

```
11 // Eingabe anfordern
12 System.out.print(
13     "Bitte Ziffer zwischen 0-9 eingeben: ");
14
15 // Tastatureingabe entgegennehmen
16 String s = tastatur.next();
17 // Erstes Zeichen aus Eingabe abfragen
18 char c = s.charAt(0);
```

Hier erfolgt die Nutzereingabe. Ihr erstes Zeichen wird abgefragt.

## Exceptions selbst erzeugen und werfen – Beispiel

### main()-Klasse MainEx1 – Teil 3

```
20 // Versuche, Zeichen in Zahl zu konvertieren
21 int wert = worker.returnZahlWert(c); ←Dieser Methodenaufruf
22 // Rueckmeldung an Nutzer ausgeben
23 System.out.println(
24     "Eingabe erfolgte mit Wert: " + wert);
25
26 } // end main
27 } // end class
```

**Dieser Methodenaufruf kann eine NumberFormatException werfen!**

# Exceptions selbst erzeugen und werfen – Beispiel

Compilierung und Start des Programms mit ...

```
javac ZahlenFormat.java
```

```
javac MainEx1.java
```

```
java MainEx1
```

... oder innerhalb von Eclipse ...

# Exceptions selbst erzeugen und werfen – Beispiel

Erster Start: Eingabe der Ziffer 5

Bitte Ziffer zwischen 0-9 eingeben: 5

Eingabe erfolgte mit Wert: 5

# Exceptions selbst erzeugen und werfen – Beispiel

Zweiter Start: Eingabe des Zeichens W

Bitte Ziffer zwischen 0-9 eingeben: W

Exception in thread "main"

```
java.lang.NumberFormatException: Keine Ziffer!  
at ZahlenFormat.returnZahlWert(ZahlenFormat.java:15)  
at MainEx1.main(MainEx1.java:22)
```

Bei Eingabe einer Nicht-Ziffer: Die letzte Ausgabe von **System.out.println()** erfolgt nicht mehr, da die Exception den Programmablauf vorher abbricht!

# Exceptions fangen try – catch – finally

## Problem bei der Verarbeitung mit throw und throws

- Wenn Exceptions bis auf die Ebene von *main()* hinauf geworfen werden, so wird ggf. der Programm-Ablauf unerwünscht unterbrochen.
- Daher will man manchmal Exceptions „an Ort und Stelle“ verarbeiten:
  - ① Code-Statements „probieren“ (try), bis eine Exception eintritt
  - ② Exception an „Ort und Stelle“ fangen und verarbeiten (catch)
  - ③ Gleichgültig ob wir in (1) oder (2) landen: Rest-Code durchführen (finally)

# Methoden, in denen Exceptions geworfen werden, ... ... benötigen nicht immer die throws-Klausel!!!

## Syntax von try – catch – finally

```
1 returnType methodName(typ1 param1, typ2 param2,...) {  
2     try {  
3         // Code-Statement 1; ... Code-Statement N;  
4     }  
5     catch(ExceptionType1 |  
6             ExceptionType2 | ... | ExceptionTypeN e){  
7         // Abfang-Statement1; ...  
8     }  
9     finally {  
10        // Schluss-Statement1; ...  
11    } // end finally  
12 } // end method
```

**Bei methoden-interner Verarbeitung von Exceptions mit try-catch können wir auf die throws-Klausel am Methodenkopf verzichten!**

# Exceptions fangen – Beispiel mit neuer *main()*-Klasse

## Klasse MainEx2 – Teil 1 – Start *main()*

```
1 import java.util.Scanner;  
  
2  
3 public class MainEx2 {  
  
4  
5     public static void main(String[] args) {  
6         // Arbeitsobjekte erstellen  
7         Scanner tastatur = new Scanner(System.in);  
8         ZahlenFormat worker = new ZahlenFormat();  
9         char c='@';  
10        int wert=-1;
```

Methode *main()* legt zuerst die benötigten Arbeitsobjekte an.

# Exceptions fangen – Beispiel mit neuer main()-Klasse

## Klasse MainEx2 – Teil 2 – try-Block

```
11  try{  
12      // Eingabe anfordern  
13      System.out.print(  
14          "Bitte Ziffer zwischen 0-9 eingeben: ");  
15  
16      // Tastatureingabe entgegennehmen  
17      String s = tastatur.next();  
18      // Erstes Zeichen aus Eingabe abfragen  
19      c = s.charAt(0);  
20  
21      // Versuche, Zeichen in Zahl zu konvertieren  
22      wert = worker.returnZahlWert(c);  
23  } // end try
```

try-Block „probiert“ die gewünschten Statements ...

# Exceptions fangen – Beispiel mit neuer *main()*-Klasse

## Klasse MainEx2 – Teil 3 – catch (Multiple Exception catch)

```
24     catch(NumberFormatException |  
25             StringIndexOutOfBoundsException ex){  
26         ex.printStackTrace();  
27     } // end catch
```

# Exceptions fangen – Beispiel mit neuer *main()*-Klasse

## Klasse *MainEx2* – Teil 3 – catch (Multiple Exception catch)

- Der catch-Block wird ausgeführt, wenn im try-Block eine Exception geworfen wurde.
- In diesem Fall wird der try-Block an der Stelle abgebrochen, an der die Exception entstand.
- `ex.printStackTrace()` ist eine von der Klasse *Exception* ererbte Methode. Sie gibt aus, wo die Exception entstand.
- Wenn wir mehrere Exceptions **mit ein und demselben Code** im catch-Block abhandeln wollen, so geben wir sie in seinem Kopf als Veroderung an: `ExceptionType1 | ExceptionType2`. Dies wird als **Multiple Exception Catch** bezeichnet.

# Exceptions fangen – Beispiel mit neuer *main()*-Klasse

## Klasse *MainEx2* – Teil 4 – finally

```
28     finally {
29         System.out.println(
30             "Inhalt von wert lautet nun: " + wert);
31     } // end finally
32 } // end method main()
33 } // end class MainEx2
```

# Exceptions fangen – Beispiel mit neuer *main()*-Klasse

## Klasse MainEx2 – Teil 4 – finally

- Der `finally`-Block ist **optional** – er kann also auch weggelassen werden.
- Wenn er vorhanden ist, wird er **immer** ausgeführt – auch wenn keine Exception geworfen wurde

## Exceptions fangen – Beispiel mit neuer main()-Klasse

### Klasse MainEx2 – Teil 3 – catch (Single Exception catch)

```
24  catch(NumberFormatException ex){  
25      System.out.println("NumberFormatException!");  
26  } // end catch NumberFormatException  
27  catch(StringIndexOutOfBoundsException ex){  
28      System.out.println("StringIndexOutOfBoundsException!");  
29  } // end catch NumberFormatException  
30  catch (Exception ex){ ←  
31      System.out.println("Andere Exception!");  
32  } // end catch Exception e
```

Diese catch-Anweisung der allgemeinen Exception muss als letzter catch erfolgen. Sonst Compilerfehler!!

# Exceptions fangen – Beispiel mit neuer *main()*-Klasse

## Klasse MainEx2 – Teil 3 – catch (Single Exception catch)

- Wenn wir mehrere Exceptions **mit jeweils unterschiedlichem Abfang-Code** im catch-Block abhandeln wollen, so geben wir einzelne getrennte catch-Klauseln an! (**Single Exception Catch**)
  
- Häufig fängt man am Ende einer solchen Folge von catch-Blöcken noch die „allgemeine“ Exception ab:  
`catch (Exception e) {...}`
  
- Jedes XXXException-Objekt **ist** gleichzeitig ein Objekt der Elternklasse *Exception*. So fangen wir alle nicht spezifisch genannten, übrigen Exceptions ab!

# Zusammenfassung Exceptions

## Klasse MainEx2 – Teil 3 – catch (Single Exception catch)

- Mit `throw` können wir eine selbst erzeugte Exception werfen.
- Eine Methode kann eine in ihr entstandene Exception über die `throws`-Klausel am Methodenkopf weiterwerfen.
- Mit `try-catch` können Exceptions methodenintern abgearbeitet werden. Der Methodenaufrufer bekommt davon nichts mit.
- Es gibt zwei Arten einer `catch`-Klausel:
  - Ein **Single Exception Catch** stellt für jede Art von Exception eine eigene `catch`-Klausel mit eigener Abarbeitung bereit.
  - Ein **Multiple Exception Catch** stellt **eine einzige** `catch`-Klausel für mehrere möglicherweise eintretende Exceptions bereit. Sie werden durch das Veroderungszeichen | voneinander unterschieden.

Fakultät Informatik

# Programmierung 2

Einstieg Streams

Vorwort

Einstieg in Java

Einstieg in die OOP: Klassen und Objekte/Instanzen

Die Klasse String

Graphische Darstellung von Klassen mit der UML – Teil 1

Vererbungsbeziehungen

Exceptions

# Einstieg Streams

Stream-Konzept in Java

Dateizugriff mit einfachen, zeichenorientierten Streams

Klassenzoo der Streams in Java

## Ein- und Ausgabe bisher: Scanner ...

... ermöglichte Tastatur-Eingaben

```
1 import java.util.Scanner;  
2  
3 public class ScannerInput {  
4  
5     public static void main(String[] args) {  
6         Scanner tastatur = new Scanner(System.in);  
7         System.out.print("Bitte eine Eingabe: ");  
8         String input = tastatur.next();  
9         System.out.println("Ihre Eingabe: " + input);  
10    } // end main()  
11 } // end class
```

Nutzereingabe erfolgt über Klasse Scanner ...

# Ein- und Ausgabe neu: Streams

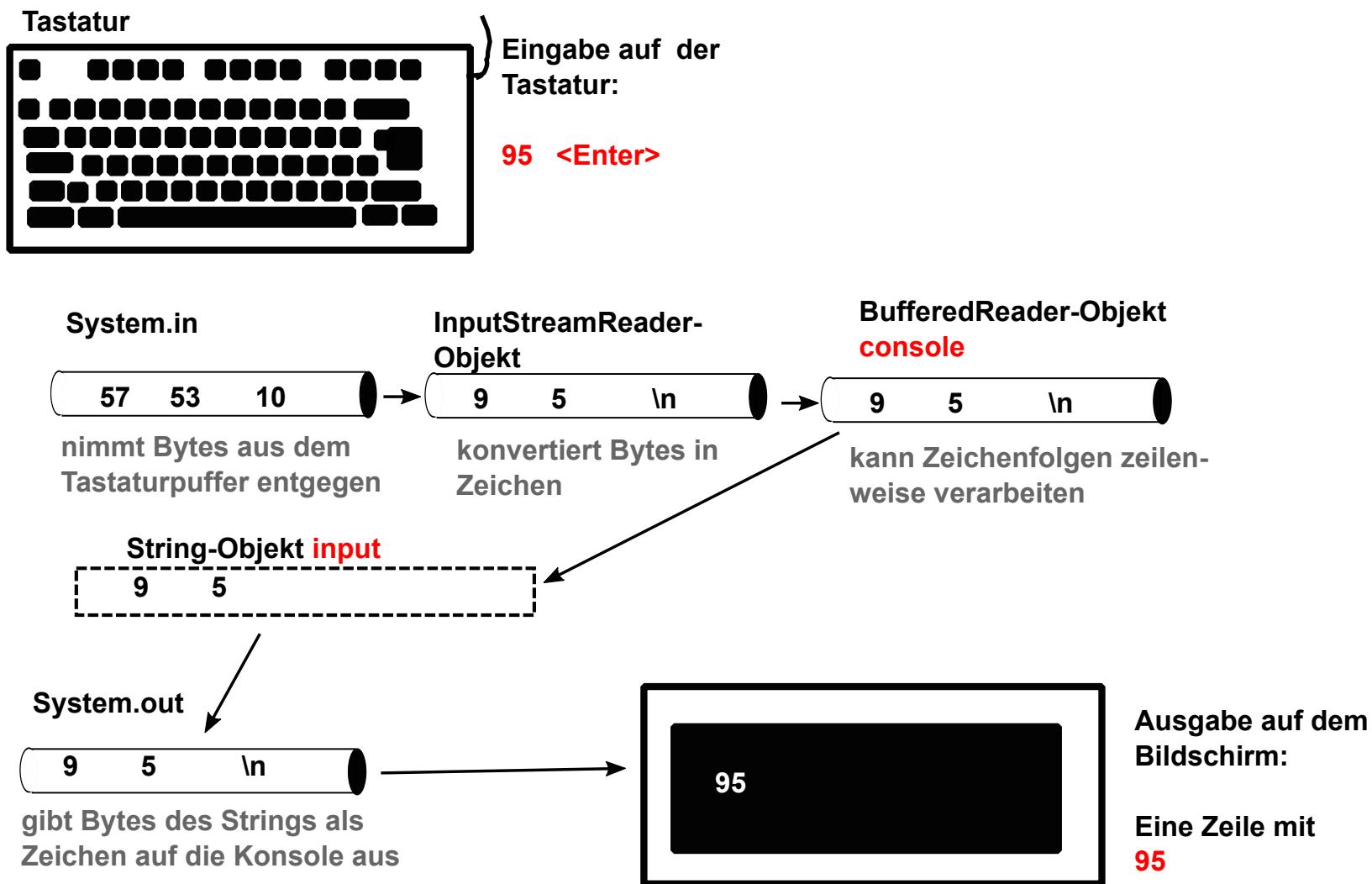
... ermöglichen auch Tastatureingaben ...

```
1 import java.io.*; // Alle Klassen aus java.io
2 public class StreamInput {
3
4     public static void main(String[] args)
5         throws IOException {
6         BufferedReader console =
7             new BufferedReader(new InputStreamReader(System.in));
8         System.out.print("Bitte eine Eingabe: ");
9         String input = console.readLine();
10        System.out.println("Ihre Eingabe: " + input);
11    } // end main()
12 } // end class
```

Nutzereingabe erfolgt über Klasse BufferedReader ...

# Ein- und Ausgabe neu: Streams

## Visualisierung des Stream-Konzepts für die obige Tastatureingabe



# Ein- und Ausgabe neu: Streams

## Abschnittsweise Interpretation des Programms

```
1 import java.io.*;  
2 public class StreamInput {  
3  
4     public static void main(String[] args)  
5         throws IOException {  
6         BufferedReader console =  
7             new BufferedReader(new InputStreamReader(System.in));  
8         System.out.print("Bitte eine Eingabe: ");  
9         String input = console.readLine();  
10        System.out.println("Ihre Eingabe: " + input);  
11    } // end main()  
12 } // end class
```

Einbinden aller Klassen aus Package `java.io`

# Ein- und Ausgabe neu: Streams

## Abschnittsweise Interpretation des Programms

```
1 import java.io.*; // Alle Klassen aus java.io
2 public class StreamInput {
3
4     public static void main(String[] args)
5         throws IOException {
6     // Rest wie vorher ...
```

Methoden der Stream-Klassen nutzen Exceptions – diese müssen weiter verarbeitet werden!

# Ein- und Ausgabe neu: Streams

## Abschnittsweise Interpretation des Programms

```
1 import java.io.*;  
2 public class StreamInput {  
3  
4     public static void main(String[] args)  
5         throws IOException {  
6         BufferedReader console =  
7             new BufferedReader(new InputStreamReader(System.in));  
8         // Rest wie vorher ...
```

Streams werden so verschachtelt, dass

- Daten aus der gewünschten Datenquelle (hier: Tastatur)
- in der gewünschten Weise konvertiert werden (hier: zeichenorientiert mit `InputStreamReader` und zeilenweise mit `BufferedReader`)

# Ein- und Ausgabe neu: Streams

## Abschnittsweise Interpretation des Programms

```
1 import java.io.*; // Alle Klassen aus java.io
2 public class StreamInput {
3
4     public static void main(String[] args)
5         throws IOException {
6         BufferedReader tastatur =
7             new BufferedReader(new InputStreamReader(System.in));
8         System.out.print("Bitte eine Eingabe: ");
9         String input = tastatur.readLine();
10        System.out.println("Ihre Eingabe: " + input);
11    } // end main()
12 } // end class
```

BufferedReader verarbeitet Eingabe zeilenweise ...

# Ein- und Ausgabe neu: Streams...

... können auch:

- Ein- und Ausgaben aus beliebigen Datenquellen / Datensenken verarbeiten (z.B. Dateien, Netzwerkverbindungen)
- Java unterscheidet zwischen:
  - Ein- und Ausgabe-Streams, sowie
  - zwischen zeichen- und byteorientierten Streams.

# Ein- und Ausgabe neu: Streams

## Vorgehen Nutzung von Streams

- Klassen aus Package `java.io` einbinden mit:

```
import java.io.*;
```

- Stream-Konstruktoren und Stream-Methoden nutzen das Exception-Konzept sehr intensiv. Daher:

- Entweder Ihre stream-nutzende Methode wirft die entsprechenden Exceptions weiter, oder
- Sie fängt sie selbst mit `try-catch`

- Vor der Nutzung in der API nachschlagen, welche Methode welche Exceptions wirft!

# Ein- und Ausgabe neu: Streams

## Vorgehen Nutzung von Streams

- Die Streams so kombinieren, dass beim Lesen:
  - aus der gewünschten Datenquelle gelesen wird, und
  - die Daten in der gewünschten Art und Weise konvertiert werden.
- Die Streams so kombinieren, dass beim Schreiben:
  - auf die gewünschte Datensenke geschrieben wird, und
  - die Daten in der gewünschten Art und Weise konvertiert werden.
- Die Kombination der Streams erfolgt über geeignete Verschachtelung der Konstruktoraufrufe – also z.B.

```
1  BufferedReader datei =  
2      new BufferedReader(new FileReader("abc.txt"));
```

# Dateizugriff mit einfachen, zeichenorientierten Streams

## Datei einlesen: Methode *dateiEinlesen()*

```
1 import java.io.*; // Alle Klassen aus java.io
2 public class Kopierer {
3
4     public String dateiEinlesen (String dateiname)
5         throws IOException,FileNotFoundException{
6         String input="";
7         String zeile;
8
9         // Datei oeffnen
10        BufferedReader dateileser =
11            new BufferedReader( new FileReader(dateiname) );
```

Datenquelle ist nun eine textorientierte Datei.

# Dateizugriff mit einfachen, zeichenorientierten Streams

## Datei einlesen: Methode *dateiEinlesen()*

```
12 // Datei auslesen
13 while ( (zeile=dateileses.readLine()) != null ) {
14     input+=zeile + "\n";
15 } // end while
```

- Methode **readLine()** liefert Zeilen-Inhalt als String
- Zuweisung wird zuerst durchgeführt, erst danach der Vergleich mit **!=**
- Wenn die Datei zu Ende ist, liefert **readLine()** eine **null**-Referenz.
- Methode **readLine()** schneidet Zeilenumbruch "**\n**" ab, daher wird er wieder angefügt

# Dateizugriff mit einfachen, zeichenorientierten Streams

Datei einlesen: Methode *dateiEinlesen()*

```
16 dateileser.close(); // Datei wieder schliessen
17 return input;
18 } // end method dateiEinlesen()
```

Stream wird wieder geschlossen. Damit wird auch die Datei wieder geschlossen.

# Dateizugriff mit einfachen, zeichenorientierten Streams

## Datei schreiben: Methode *dateiSchreiben()*

```
19     public void dateiSchreiben(String dateiname, String inhalt)  
20         throws IOException{  
  
21  
22     // Datei oeffnen  
23     BufferedWriter dateischreiber =  
24         new BufferedWriter( new FileWriter(dateiname) );
```

Datensenke ist nun eine textorientierte Datei

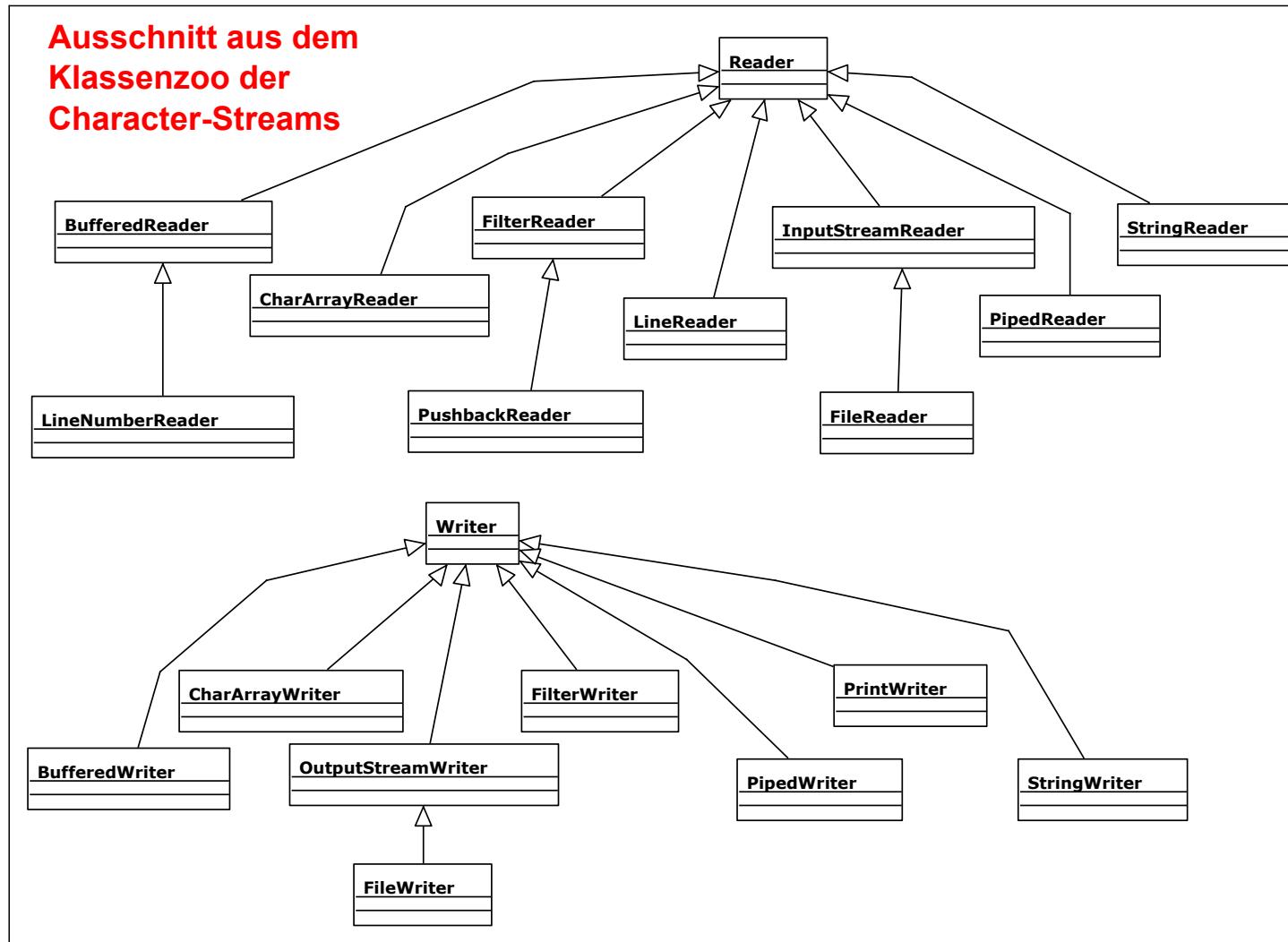
# Dateizugriff mit einfachen, zeichenorientierten Streams

## Datei schreiben: Methode *dateiSchreiben()*

```
25 // Datei schreiben
26 dateischreiber.write(inhalt) ;
27
28 // Datei wieder schliessen
29 dateischreiber.close();
30 } // end method dateiSchreiben()
```

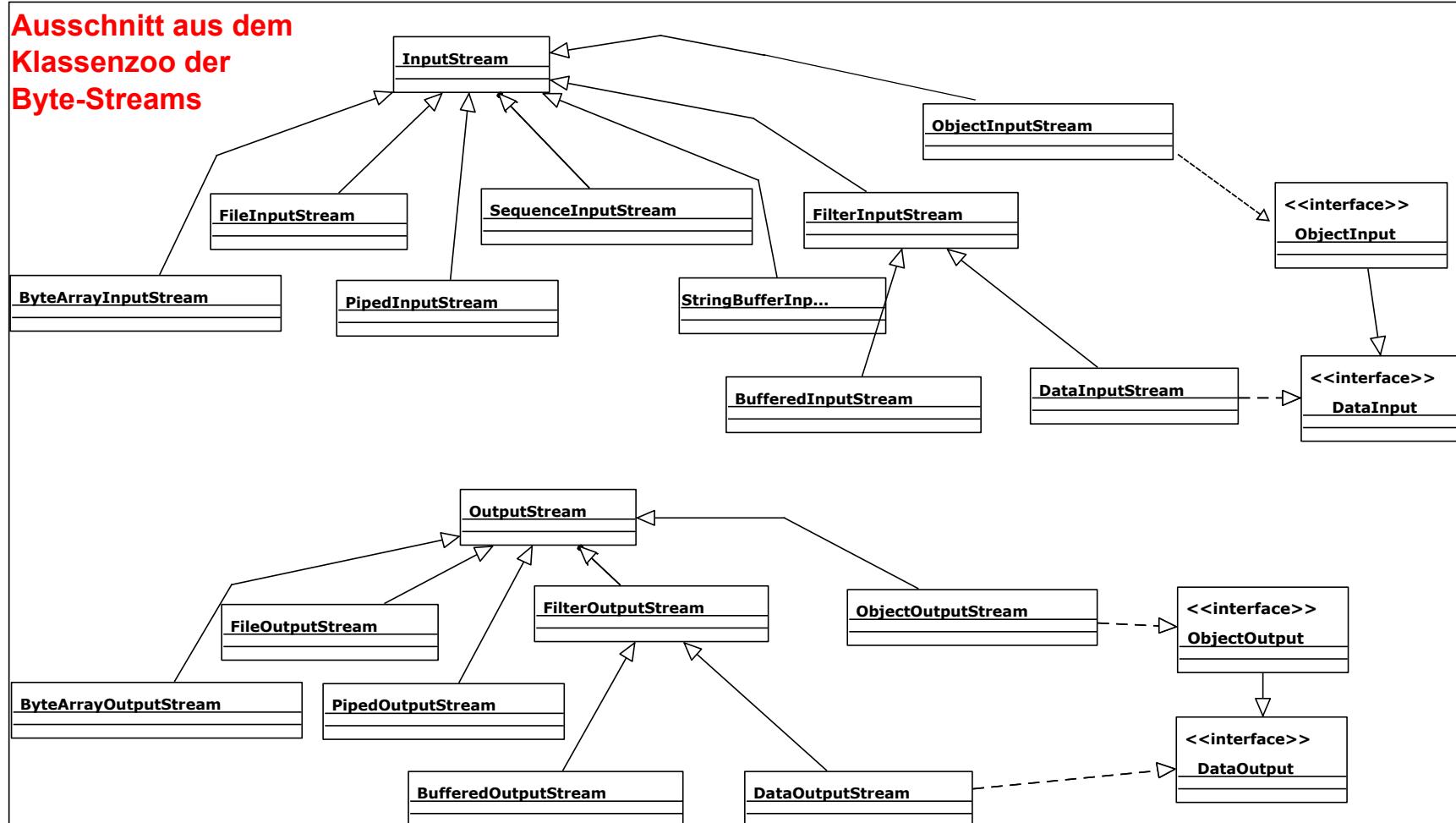
- Mehrzeiliger String wird „auf einen Schlag“ auf die Datei geschrieben.
- Danach wird der Stream (und damit die Datei) wieder geschlossen.

# Zeichenorientierte Streams in Java



→ Für zeichenorientierte Daten (z.B. Textdateien)

# Byteorientierte Streams in Java



→Für byteorientierte Daten (z.B. JPG-Bilder)

---

## Literatur I

- [1] GOLL, J. u. a.: Java als erste Programmiersprache. 8. Teubner-Verlag, 2016
- [2] OESTERREICH., B.: Analyse und Design mit der UML2.5. 11. Oldenbourg-Verlag, 2013
- [3] ORACLE: The Java-Tutorials – Language Basics.  
<http://download.oracle.com/javase/tutorial/java/nutsandbolts/operators.html>, 2018
- [4] ULLENBOOM, Christian: Java ist auch eine Insel. 12. Rheinwerk Verlag, 2020

Fakultät Informatik

# Programmierung 2

Priorität von Operatoren



# Priorität von Operatoren

# Bei mehreren Operatoren in einem Ausdruck:

## Priorität von Operatoren in Java

- sagt aus, in welcher Reihenfolge Operatoren innerhalb eines Ausdrucks ausgewertet werden.
- Schulmathematik z.B.: „Punkt vor Strich“
- **Assoziativität:** sagt aus, ob ein Operator von links nach rechts oder von rechts nach links ausgewertet wird.

# Operatorentabelle

Die Prioritäten sind folgendermaßen zu lesen:

- Priorität 1: höchste Priorität
- Priorität 15: niedrigste Priorität
- frei nach [1], Kapitel 7.

Priorität	Operatoren	Bedeutung	Assoziativität
1	[]	Array-Index	links
	()	Klammerung von Ausdrücken; Klammern beim Methodenaufruf	links
	.	Zugriff von außen auf Objektvariablen oder Methoden	links
	++	Postinkrementoperator	links
	--	Postdekrementoperator	links
2	++	Preinkrementoperator	rechts
	--	Predekrementoperator	rechts
	+ -	Vorzeichen	rechts
	~	Bitweiser Negationsoperator	rechts
	!	Logischer Negationsoperator	rechts
3	(datentyp)	Expliziter Typecast	rechts

Priorität	Operatoren	Bedeutung	Assoziativität
	new	Objekterzeugung	rechts
4	* / %	Multiplikationoperator, Divisionsoperator, Modulooperator	links
5	+ -	Additionsoperator, Subtraktionsoperator	links
	+	Stringverkettungs-Operator	links
6	>> >>> <<	Vorzeichenerhaltender Rechtsshift, Vorzeichenloser Rechtsshift, Linksshift	links
7	< <= > >=	Vergleichsoperatoren Kleiner, Kleinergleich, Größer, Größergleich	links
	instanceof	Typüberprüfung eines Objektes	links

Priorität	Operatoren	Bedeutung	Assoziativität
8	<code>==</code> <code>!=</code>	Gleichheitsoperator Ungleichheitsoperator	links links
9	<code>&amp;</code>	Bitweiser Undoperator	links
10	<code>^</code>	Bitweiser Exklusivoderoperator (ExOr)	links
11	<code> </code>	Bitweiser Oderoperator	links
12	<code>&amp;&amp;</code> <code>&amp;</code>	Logischer Undoperator	links
13	<code>  </code> <code> </code>	Logischer Oderoperator	links
14	<code>? :</code>	Bedingungsoperator	rechts
15	<code>=</code>  <code>+ = - = * = / =</code> <code>% = &amp; =   = ^ =</code> <code>~ = &gt;&gt; = &gt;&gt;&gt;</code> <code>&lt;&lt; =</code>	Einfacher Zuweisungsoperator  Zusammengesetzte Zuweisungsoperatoren	rechts rechts rechts

---

Priorität	Operatoren	Bedeutung	Assoziativität
-----------	------------	-----------	----------------

---