



Objektorientierte und Funktionale Programmierung

SS 2014

3 Objektorientierter Entwurf mit UML und Java





Klausurtermine

- Montag, 04.08.14, 10-12 Uhr, Audimax oder Turnhalle
- Freitag, 19.09.14, 10-12 Uhr, Audimax oder Turnhalle



Anmeldezeitraum

- Der Anmeldezeitraum für die Klausuren läuft vom **xx.06.14 bis xx.xx.14 (KW xx-xx)**

3 Objektorientierter Entwurf mit UML und Java ...



Lernziele

- Vorgehensweise beim objektorientierten Entwurf kennen
- Detaillierteres Wissen über UML Klassendiagramme
- Klassen in Java definieren können

Literatur

- **[Ba05], LE 11, 12**
- [Ba99], LE 4, 5
- [BK03], Kap. 1, 2



Aufgabe des objektorientierten Entwurfs

- Entwicklung eines Lösungskonzepts unter Berücksichtigung der Rahmenbedingungen
 - dazu notwendig: Definition einer (fachlichen und technischen) Anwendungsarchitektur
 - Strukturierung in Komponenten, Schichten, etc.
- In EI II im Vordergrund: Verfeinerung des Klassendiagramms
 - OOD-Modell enthält alle Klassen, Attribute und Operationen des späteren Programms
 - mit syntaktisch korrekten Namen
 - das fertige OOD-Modell kann direkt (auch automatisch) in einen Programm**rahmen** übersetzt werden
 - enthält noch keine Implementierung der Operationen



Verfeinerung der Klassendiagramme beim Entwurf

- Festlegung von Datentypen der Attribute
- Genauere Spezifikation von Operationen
 - Parameter und Ergebnisse, sowie deren Datentypen
- Definition der Sichtbarkeit von Attributen und Operationen
 - Umsetzung des Geheimnisprinzips
- Verfeinerte Spezifikation von Assoziationen
- Objektverwaltung
- Einführung von abstrakten Operationen und Schnittstellen
- Strukturierung mit Hilfe von Paketen

3 Objektorientierter Entwurf mit UML und Java ...



Anmerkung zur Vorgehensweise der Vorlesung

- Für jedes Konzept des OOD-Klassendiagramms wird auch die
- Umsetzung in die Programmiersprache Java gezeigt
 - obwohl diese Umsetzung nicht mehr zum Entwurf gehört

Java

- Objektorientierte Programmiersprache
- 1996 von Sun Microsystems entwickelt
- Stark an C++ angelehnt, aber einfacher
- Wird vor allem bei Internet-Anwendungen eingesetzt



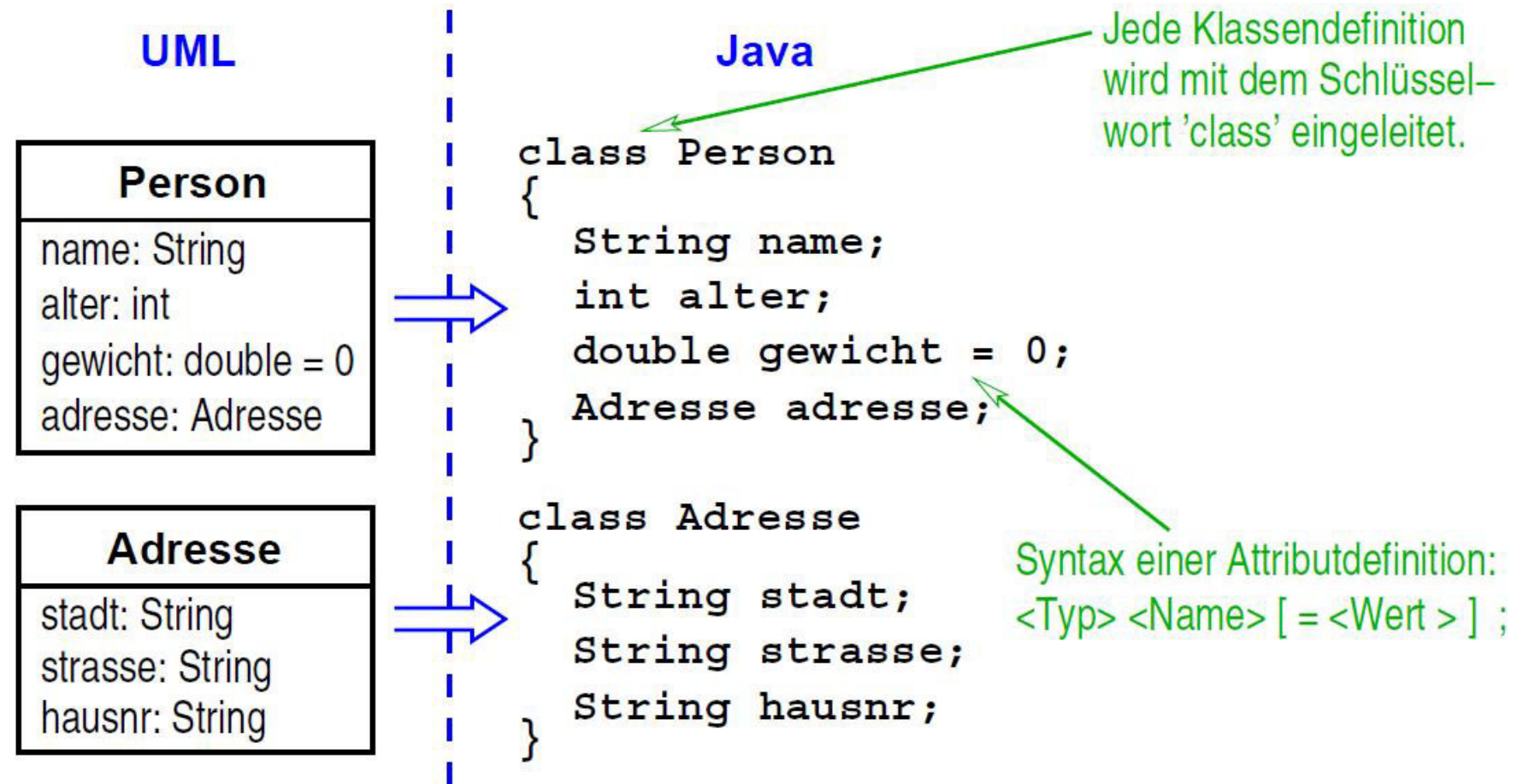
3.1 Attribute

- Zur Programmierung: jedes Attribut muß einen festen Datentyp besitzen
 - das Attribut kann nur Werte dieses Typs besitzen

- Als Attributtypen sind möglich:
 - vordefinierte, primitive Datentypen
 - in Java u.a.:
 - int: ganze Zahl, z.B. 0, 42, -17
 - double: Gleitkommazahl, z.B. .1, 3.1415, -17e-2
 - String: Zeichenkette, z.B. "Hallo"
 - eigene Datentypen
 - in Java definiert jede Klasse auch einen Datentyp

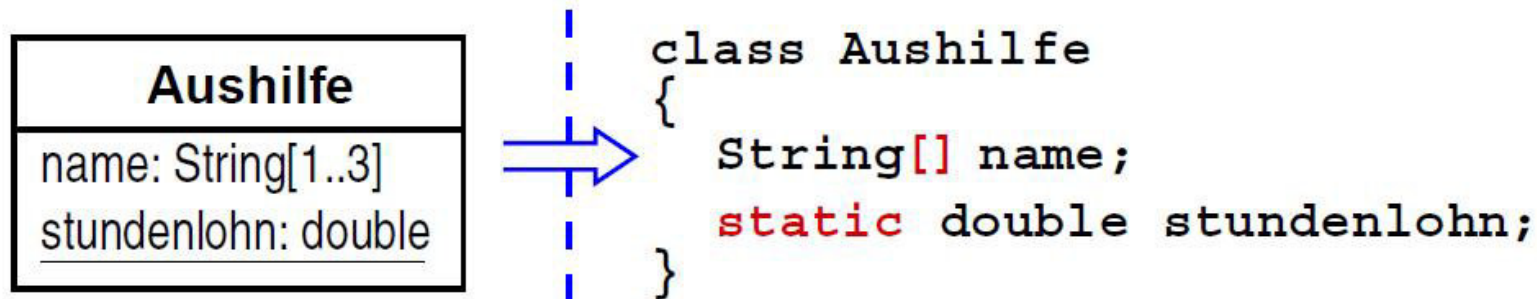
3.1 Attribute ...

Umsetzung in Java: einfache Attribute



3.1 Attribute ...

Umsetzung in Java: Multiplizitäten, Klassenattribute

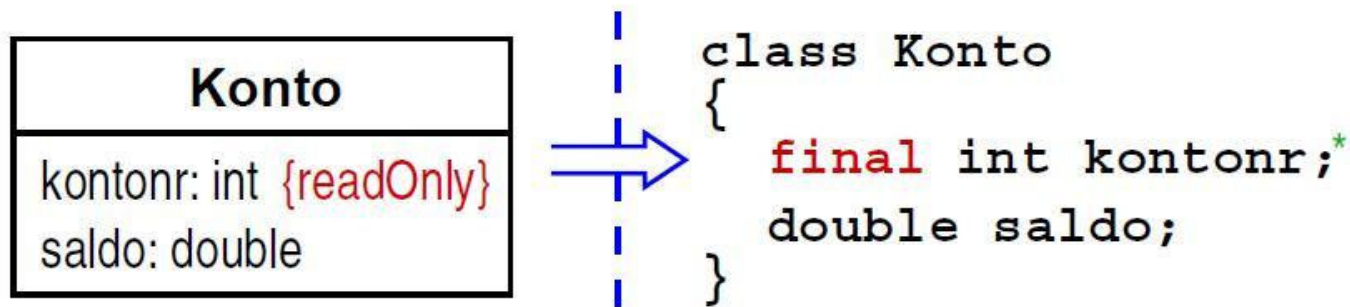


- Attribute mit Multiplizität ungleich 1 werden in Java durch **Felder** (**Arrays**) dargestellt (siehe später)
 - in Java erfolgt bei der Deklaration eines Feldes keine Angabe, wieviel Elemente es enthalten kann
- Klassenattribute werden durch das vorgestellte Schlüsselwort **static** gekennzeichnet

3.1 Attribute ...

Unveränderliche Attribute

- Attribute können als nicht veränderbar gekennzeichnet werden
 - ihre Werte bleiben nach der Erzeugung (Initialisierung) des Objekts konstant
 - z.B. die Kontonummer eines Kontos
- Darstellung in UML und Java:



* vgl. Abschnitt 3.4!



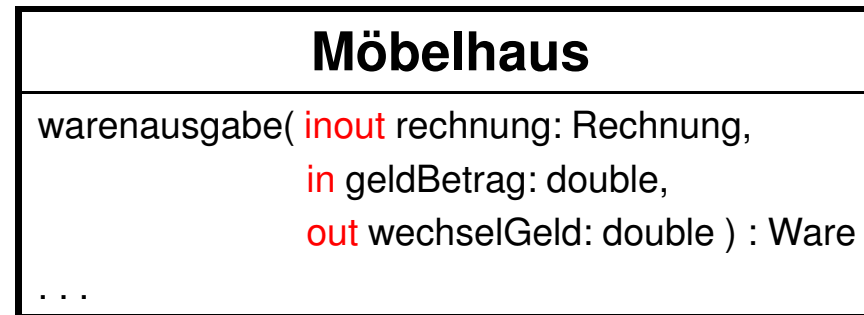
3.2 Operationen

- Für jede Operation einer Klasse muß festgelegt werden:
 - Name der Operation
 - Liste aller Parameter (ggf. leer) mit folgenden Angaben:
 - Name des Parameters
 - Parameter-Typ
 - analog zu Attributen, ggf. auch mit Multiplizität
 - Richtung des Datenflusses:
 - Eingabewert, Ausgabewert oder beides
 - Falls die Operation einen Ergebniswert liefert: Typ des Ergebniswerts
- Der Name bildet zusammen mit der Liste der Parameter(typen) die **Signatur** einer Operation

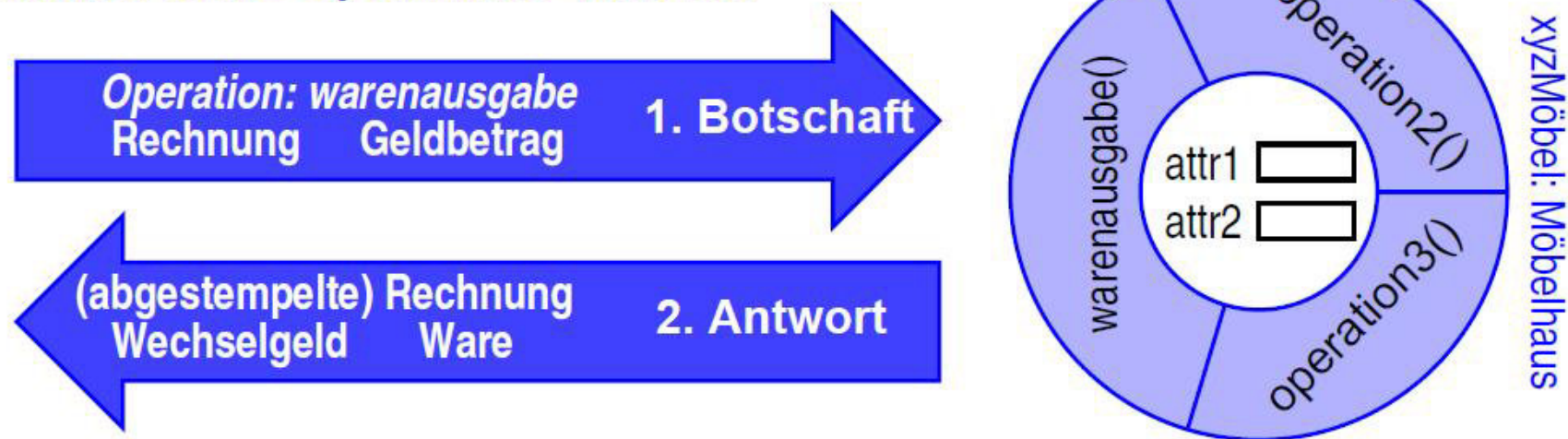
3.2 Operationen ...

Zur Richtung von Parametern

UML– Darstellung:



Ablauf eines Operations-Aufrufs:



3.2 Operationen ...

Umsetzung in Java: einfache Operationen und Parameter

Operation ohne Rückgabewert

Methodenkopf: definiert den Signatur der Operation

Kreis
zeichnen() verschieben(in dx: int, in dy: int) skalieren(in faktor: double) berechneFlaeche(): double

Methodenrumpf: definiert die Programmcode für die Implementierung der Operation

class Kreis

```

{
    void zeichnen() {
        ... // Code der Operation 'zeichnen'
    }
    void verschieben(int dx, int dy) {
        ... // Code der Operation 'verschieben'
    }
    void skalieren(double faktor) {
        ... // Code der Operation 'skalieren'
    }
    double berechneFlaeche() {
        ... // Code der Operation 'berechneFlaeche'
    }
}

```

3.2 Operationen ...



Anmerkungen

- In UML ist die Angabe einer Parameterliste optional
 - wenn eine Liste angegeben wird, muß sie mindestens die Namen der Parameter enthalten
- In Java muß die Parameterliste immer angegeben werden (falls die Operation Parameter hat)
 - für jeden Parameter muß Name und Typ definiert werden
 - es gibt keine Angabe der Richtung
 - wir betrachten vorerst nur Eingabe-Parameter
 - mehr dazu später!
- Operationen ohne Ergebnis werden in Java durch den Ergebnis"typ" void gekennzeichnet



3.2 Operationen ...

Umsetzung in Java: Multiplizitäten, Klassenoperationen

Aushilfe
erhoeheLohn(in betrag: double) setzeAdresse(in adresse: String[1..5])

```
class Aushilfe
{
    Static void
    erhoeheLohn(double betrag) {
        ...
    }
    void setzeAdresse(String [ ]
                       adresse) {
        ...
    }
}
```

- Multiplizitäten bei Parametern werden analog zu Attributen umgesetzt
- Klassenoperationen werden durch das vorgestellte Schlüsselwort **static** gekennzeichnet

3.2 Operationen ...



Überladen von Operationen

- Eine Klasse darf mehrere Operationen mit dem gleichen Namen besitzen, falls sich deren Parameterlisten unterscheiden:
 - in der Anzahl der Parameter
 - oder in mindestens einem Parametertyp
- Man spricht dann von einer **überladenen Operation**
- Operationen mit gleicher Signatur, die sich nur im Ergebnistyp unterscheiden, sind nicht erlaubt
 - ein Objekt kann sonst beim Empfang einer Botschaft die auszuführende Operation nicht eindeutig bestimmen

3.2 Operationen ...



Beispiele zum Überladen

Zulässig:

```
class Ware {  
    // Bezahlen per Überweisung  
    void bezahlen(double betrag,  
                  Konto konto) {  
        ...  
    }  
  
    // Bezahlen per Kreditkarte  
    void bezahlen(double betrag,  
                  Kreditkarte karte) {  
        ...  
    }  
}
```

Unzulässig:

```
class KursVerwaltung {  
    // Suche Kurs, Ergebnis: KursNr.  
    int suche(String stichwort) {  
        ...  
    }  
  
    // Suche Kurs, Ergebnis: Titel  
    String suche(String stichwort) {  
        ...  
    }  
}
```



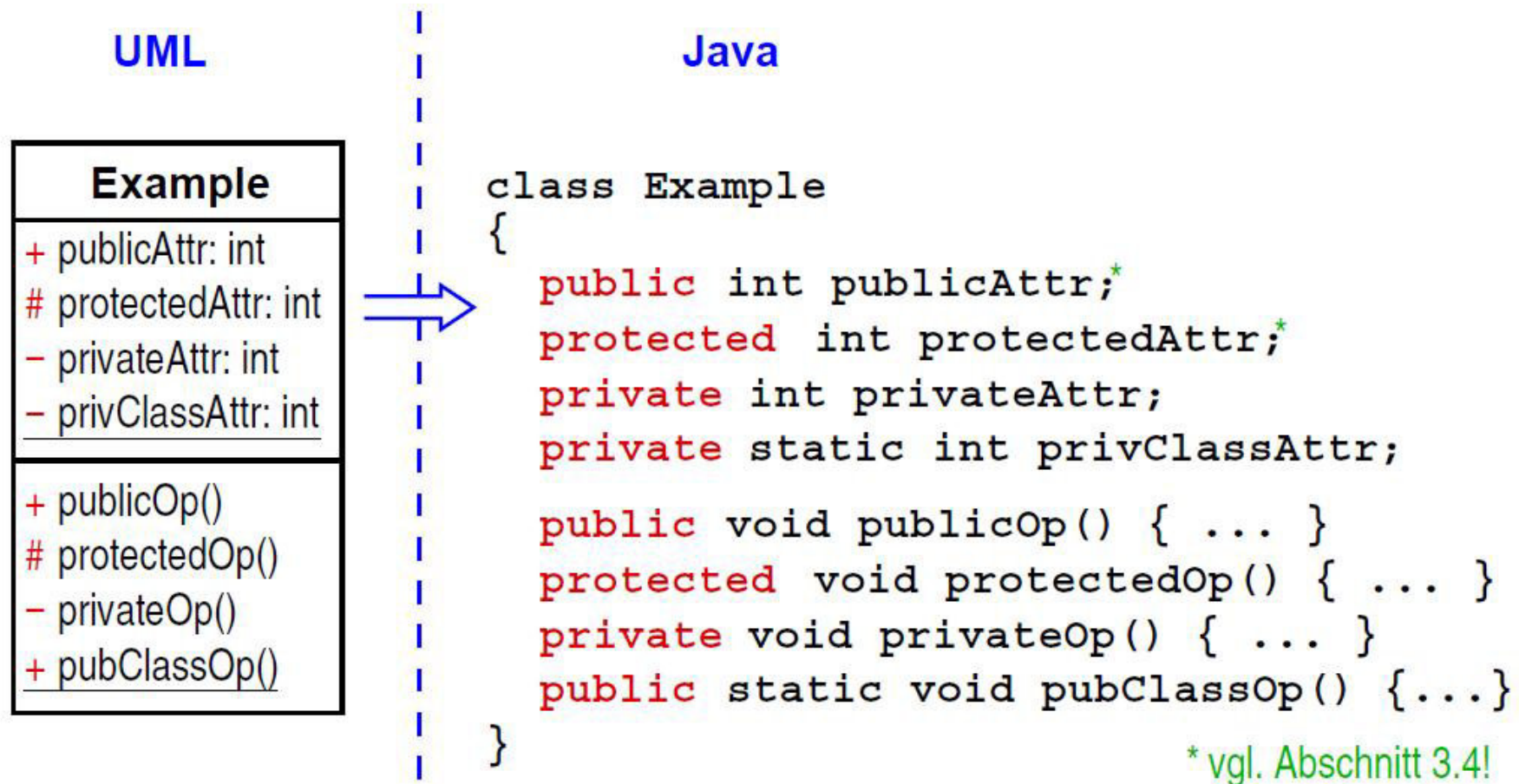
3.3 Sichtbarkeit von Attributen und Operationen

- UML und Java erlauben die Festlegung, welche Attribute und Operationen einer Klasse "von Außen" sichtbar sein sollen
 - zur Realisierung von Geheimnisprinzip und Datenkapselung
- Wir unterscheiden (zunächst) drei Sichtbarkeiten:
 - **public** (öffentlich): sichtbar für alle Klassen
 - auf Attribut kann von allen Klassen aus zugegriffen werden
 - Operation kann von allen Klassen aufgerufen werden
 - **private** (privat): sichtbar nur innerhalb der Klasse
 - kein Zugriff/Aufruf durch andere Klassen möglich
 - **protected** (geschützt): sichtbar nur innerhalb der Klasse und ihren Unterklassen

3.3 Sichtbarkeit von Attributen und Operationen ...



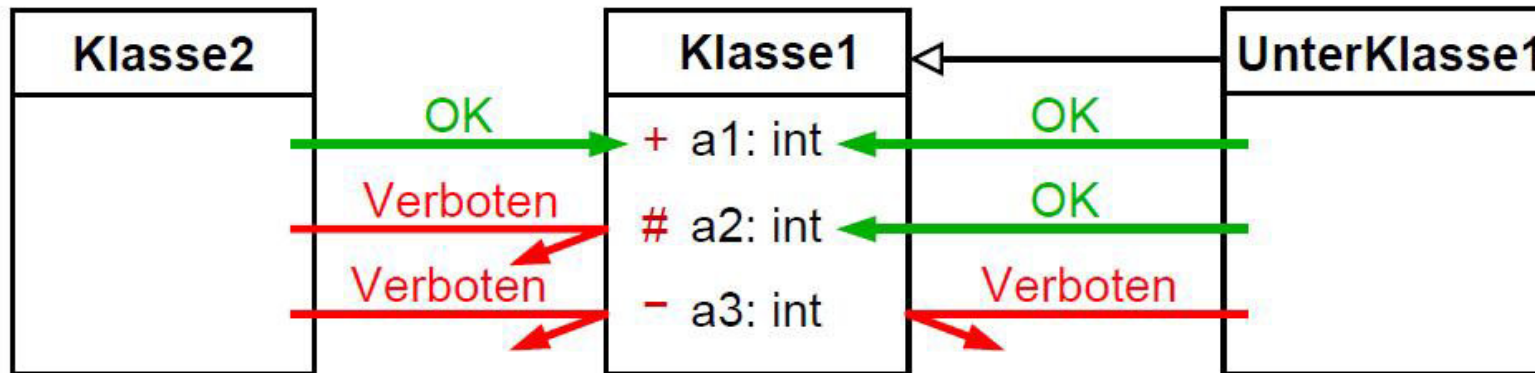
Spezifikation der Sichtbarkeit in UML und Java



3.3 Sichtbarkeit von Attributen und Operationen ...



Auswirkung auf Zugriffsversuche zwischen Klassen

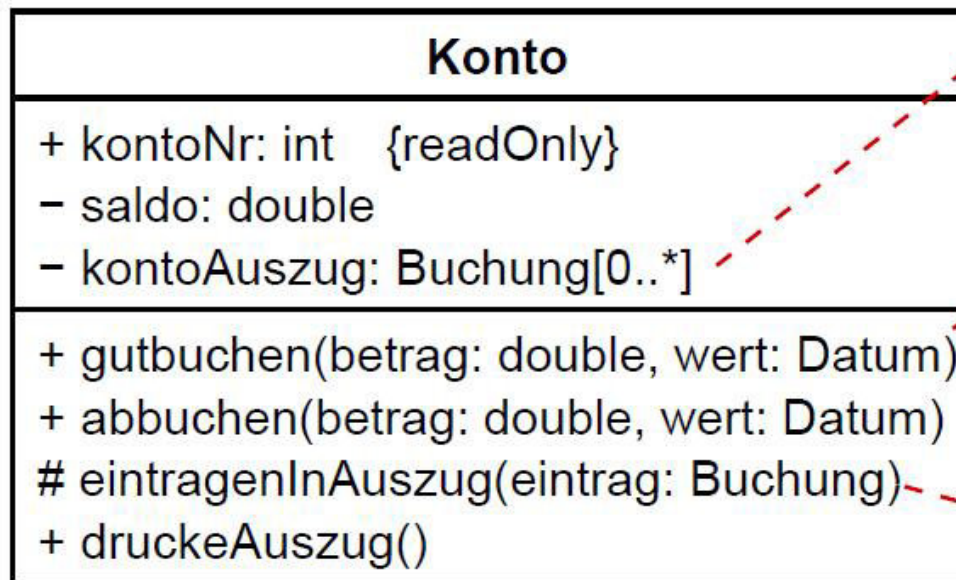


- Anmerkungen zu Sichtbarkeiten in Java:
 - die Bedeutung von *protected* ist etwas anders als in UML
 - obiges Diagramm gilt daher nur, wenn die Klassen in verschiedenen Paketen definiert wurden (=> 3.9)
 - die Angabe *public/protected/private* kann auch entfallen
 - dies definiert eine spezielle Sichtbarkeit (=> 3.9)

3.3 Sichtbarkeit von Attributen und Operationen ...



Beispiel: Kontoverwaltung



Die Buchungen sind nach Wertstellungsdatum sortiert

Ruft eintragenInAuszug() auf, um Eintrag im Kontoauszug zu machen

Unterklassen können beliebig eigene Einträge in den Kontoauszug machen

Anmerkung: hier wird auch die UML–Notation für Kommentare gezeigt



3.4 Get- und Set-Methoden

- In der **Implementierung** einer Klasse sollten Attribute immer *private* sein
 - Wahrung des Geheimnisprinzips: kein direkter Zugriff
- Konvention: Zugriff auf Attribute von außen nur über Get- und Set-Methoden, z.B.:

```
private String name;           // Attribut
public String getName();       // Get-Methode
public void setName(String aName); // Set-Methode
```

- Vorteil: Kapselung der Zugriffe
 - feste Schnittstelle nach außen, unabhängig von konkreter Speicherung bzw. Darstellung der Daten
 - Get- und Set-Methoden können Prüfungen vornehmen



3.4 Get- und Set-Methoden ...

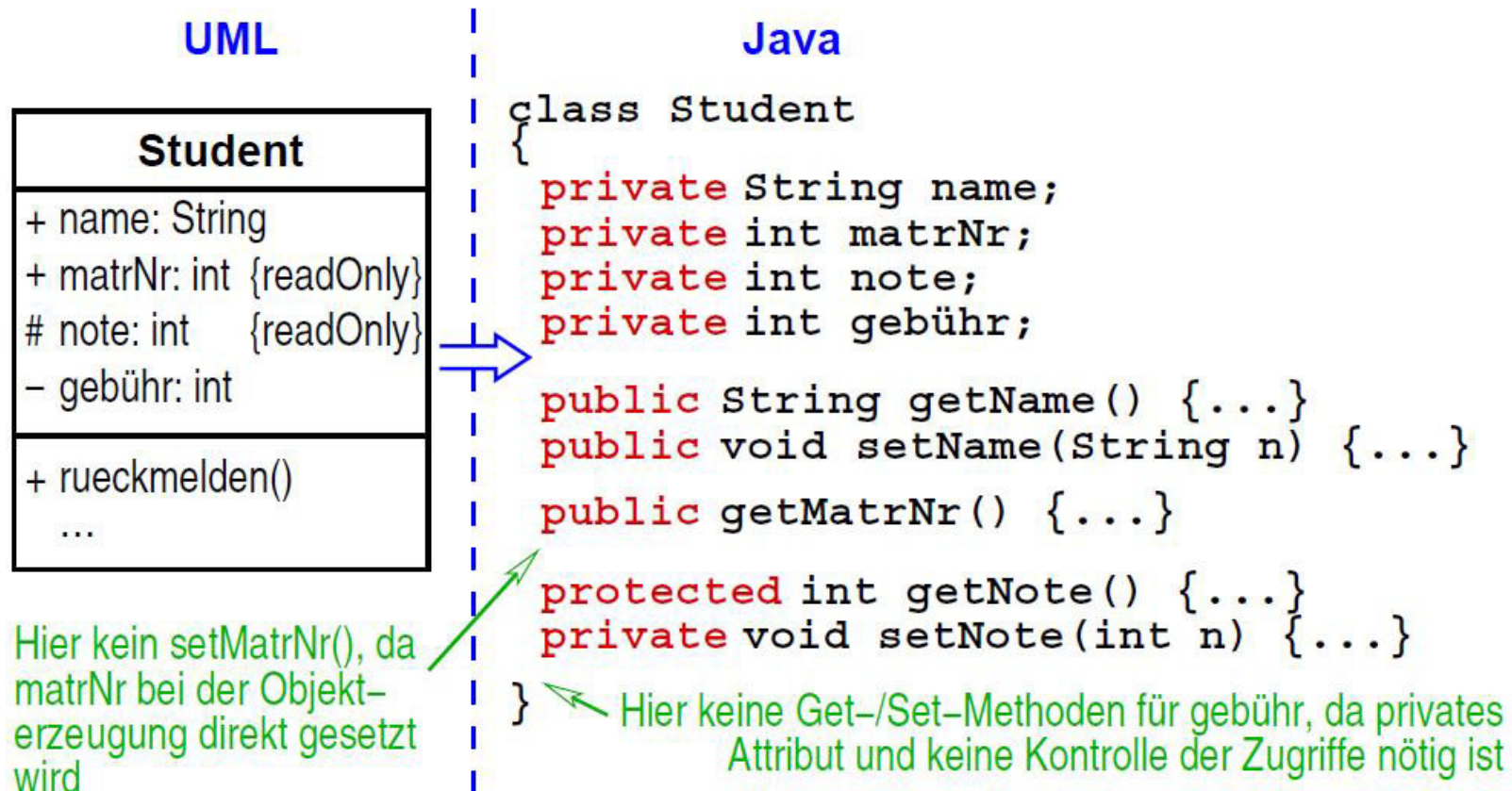
Hinweise zu Get- und Set-Methoden

- Namenskonvention beachten: `getXxx()`, `setXxx()`
- Get- und Set-Methoden werden i.d.R. nicht im Klassendiagramm dargestellt
 - nur das Attribut wird gezeigt
- Die Sichtbarkeit des Attributs im Klassendiagramm bestimmt die Sichtbarkeit der Get- und Set-Methoden im Java-Code
 - im Java-Code ist das Attribut selbst immer `private`
- Bei `readOnly`-Attributen:
 - Set-Methode ist `private` bzw. kann auch ganz fehlen
- Get- und Set-Methoden mit Verstand verwenden!
 - nur dort, wo es auch sinnvoll ist ...



3.4 Get- und Set-Methoden ...

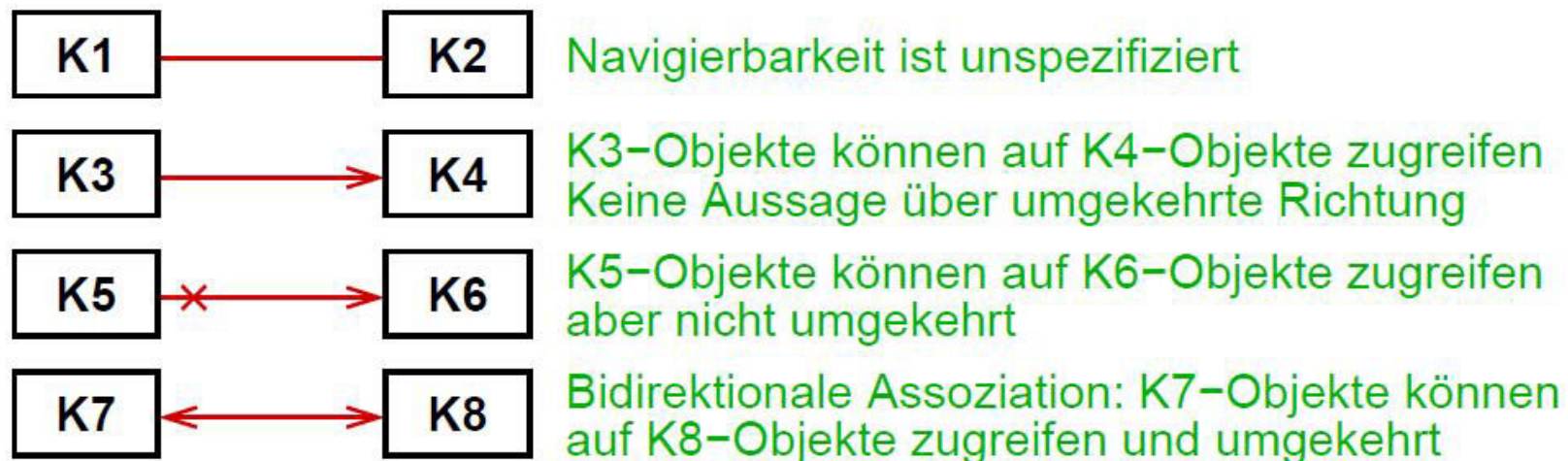
Beispiel: eine Studentenklasse





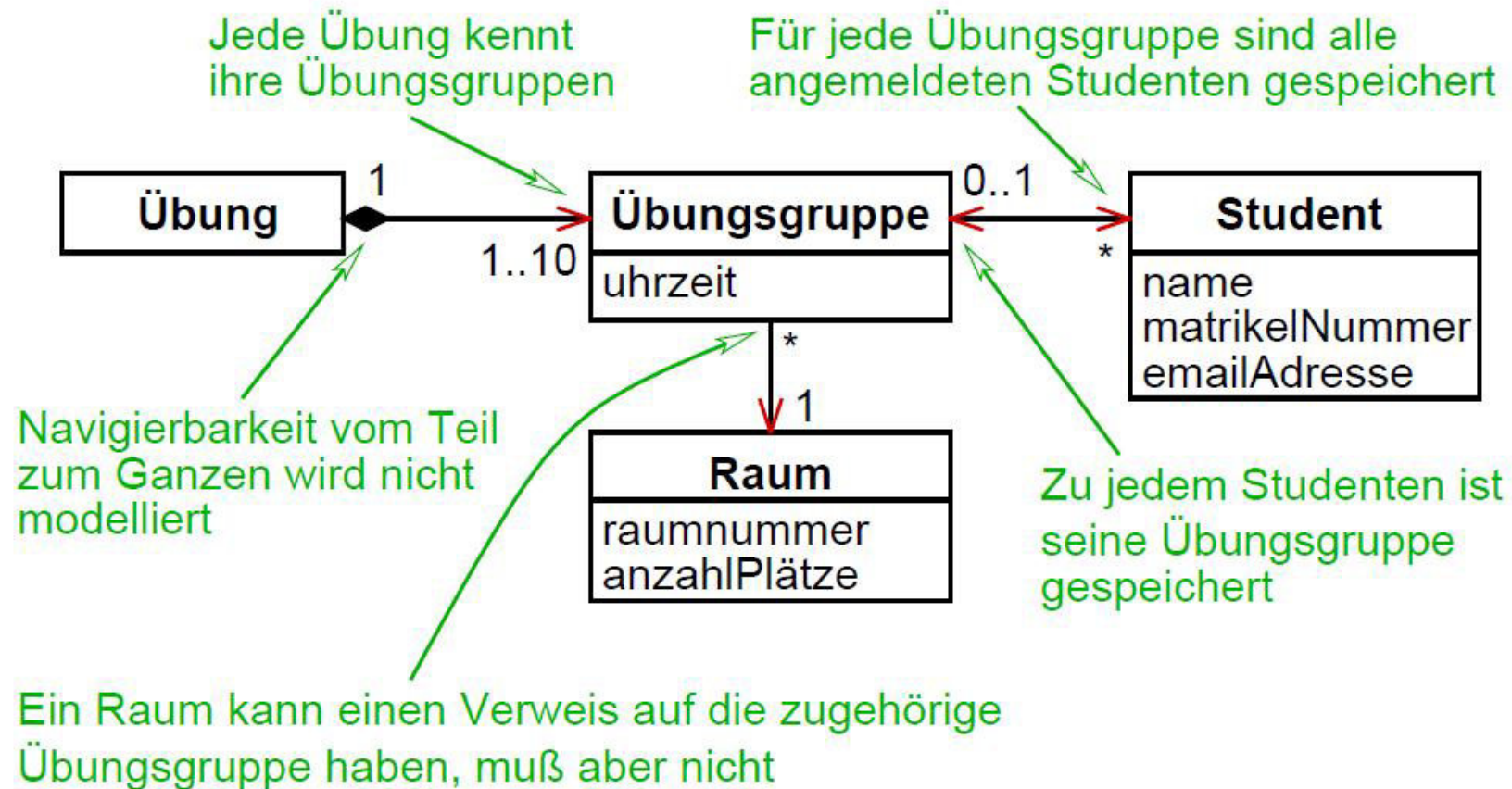
3.5 Assoziationen

- Im Entwurf wird zusätzlich die **Navigierbarkeit** modelliert:
 - Assoziation von A nach B navigierbar => Objekte von A können auf Objekte von B zugreifen (aber nicht notwendigerweise umgekehrt)
- Darstellung in UML:



3.5 Assoziationen ...

Beispiele zur Navigierbarkeit





3.5 Assoziationen ...

Realisierung von Assoziationen

- Was bedeutet die Navigierbarkeit für die Realisierung von Assoziationen?
 - **K1 - > K2** bedeutet, daß das K1-Objekt das K2-Objekt (bzw. die K2-Objekte) "kennen" muß
 - d.h., das K1-Objekt muß eine **Referenz** (auch **Verweis**, **Zeiger**) auf das K2-Objekt speichern
- In der Programmierung ist eine Referenz ein spezieller Wert, über den ein Objekt eindeutig angesprochen werden kann
 - z.B. Adresse des Objekts im Speicher des Rechners
 - Referenzen können wie normale Werte benutzt werden:
 - Speicherung in Attributen
 - Übergabe als Parameter / Ergebnis von Operationen



3.5 Assoziationen ...

Referenzen in Java

- In Java ist jeder Klassenname auch ein gültiger Datentyp
 - z.B. zur Deklaration von Attributen oder Parametern
- Beispiel:

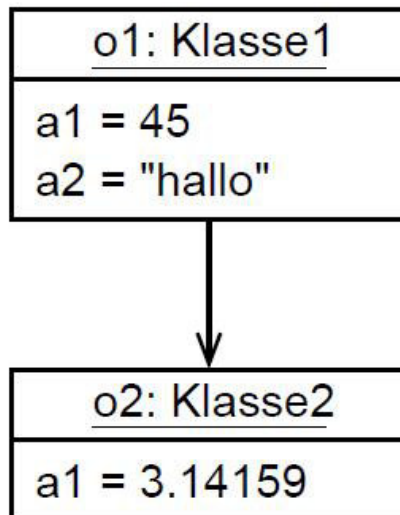
```
class übungsgruppe {  
    Raum übungsraum;  
    ...  
}
```
- Der Datentyp ist dabei immer eine **Referenz** auf die angegebene Klasse (genauer: auf ein Objekt dieser Klasse)
 - im Beispiel: das Attribut **übungsraum** speichert eine Referenz auf ein Raum-Objekt, nicht das Objekt selbst!
- Anmerkung: Java unterscheidet (im Gegensatz zu C++) die Begriffe "Referenz" und "Zeiger" nicht

3.5 Assoziationen ...

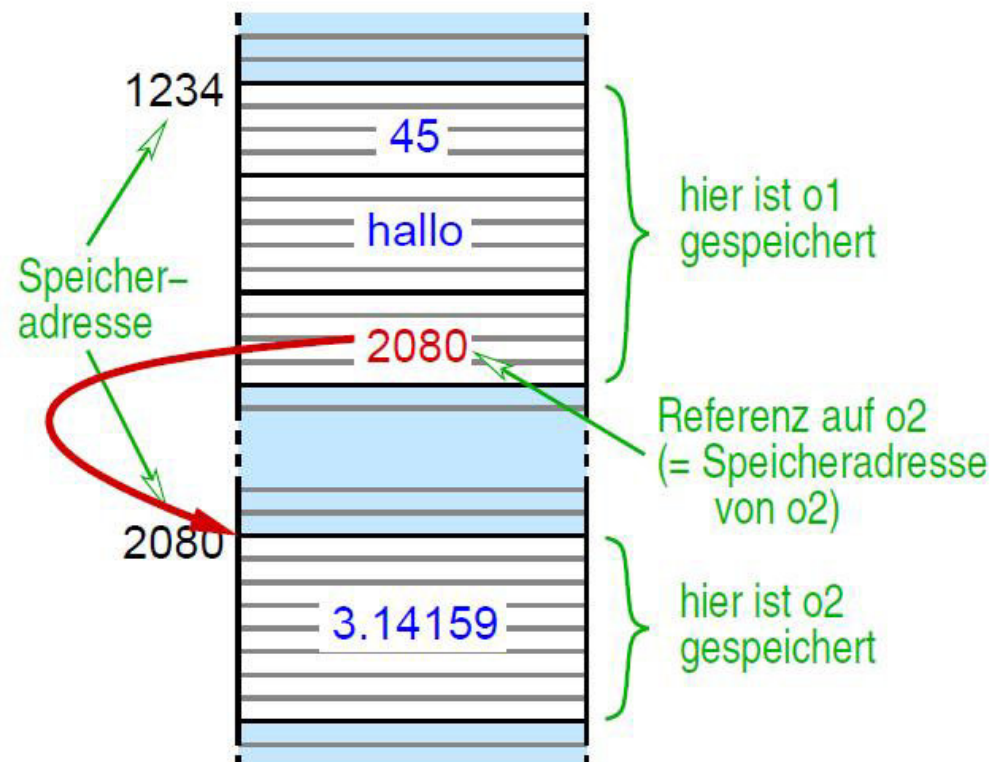


Speicherung von Objekten im Rechner (Modellvorstellung)

UML Objektdiagramm



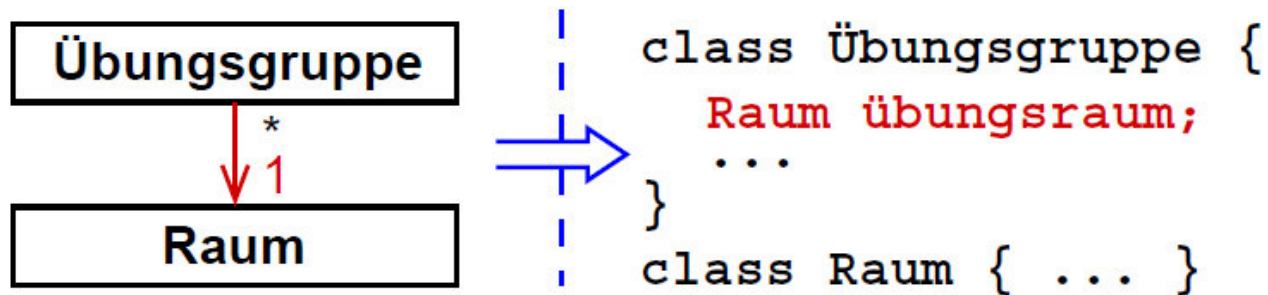
Speicherung im Rechner



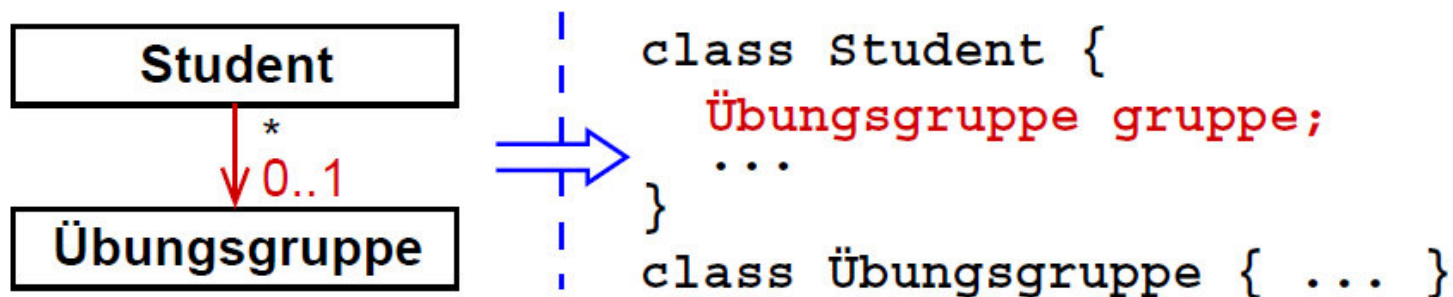
3.5 Assoziationen ...

Umsetzung von Assoziationen in Java

➔ Muß-Assoziation (Navigation in eine Richtung)



➔ Kann-Assoziation (Multiplizität 0..1, einseitige Navigation)

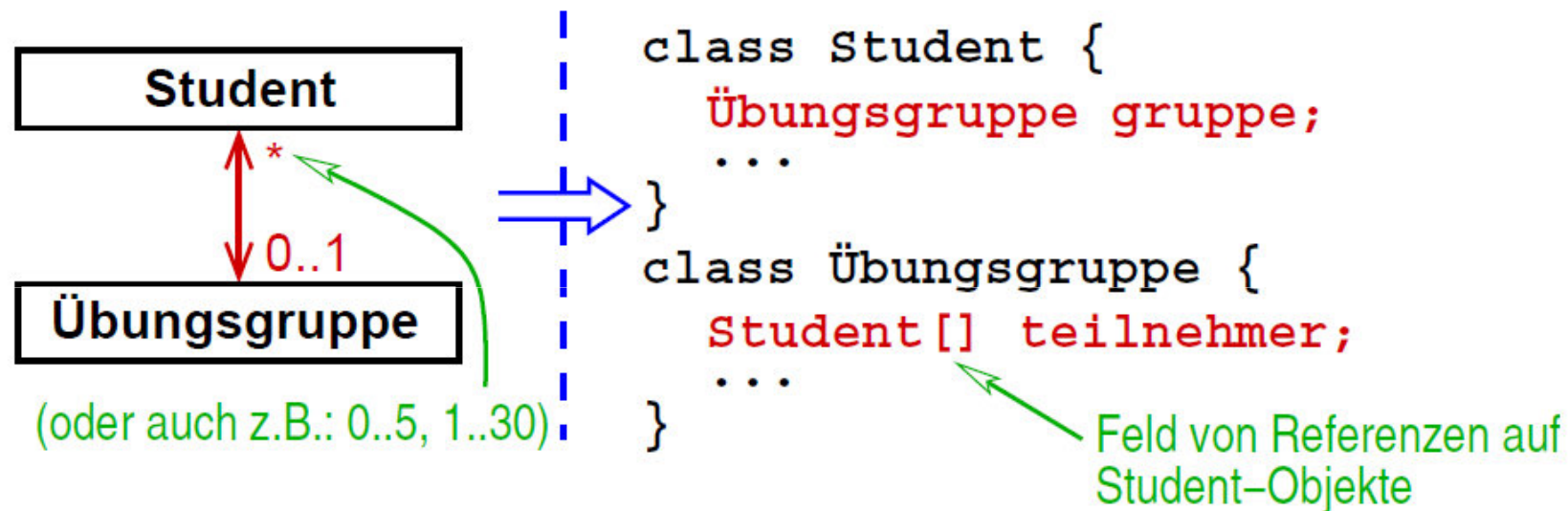


➔ spezieller Referenz-Wert `null` zeigt auf kein Objekt

3.5 Assoziationen ...

Umsetzung von Assoziationen in Java ...

- Assoziation mit Multiplizität größer 1, beidseitige Navigation

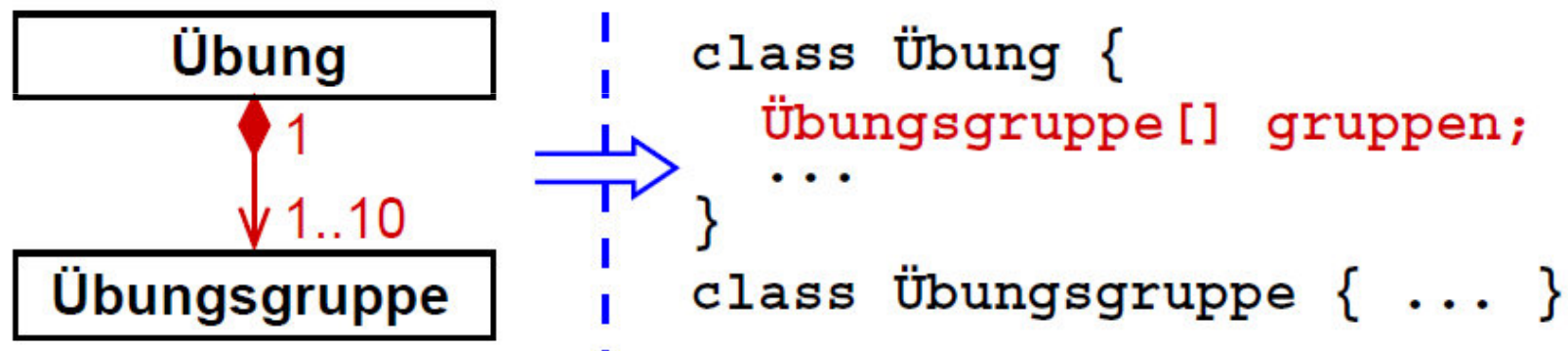


- wie viele Elemente ein Feld enthalten kann wird erst bei der Initialisierung des Feldes festgelegt (siehe später)
- statt eines Feldes können auch *Container*-Klassen aus der Java-Laufzeitbibliothek verwendet werden (siehe später)

3.5 Assoziationen ...

Umsetzung von Aggregation und Komposition in Java

- Wie bei einfachen Assoziationen
 - Aggregat/Komposit-Objekt enthält i.a. Feld von Referenzen auf die Teil-Objekte:

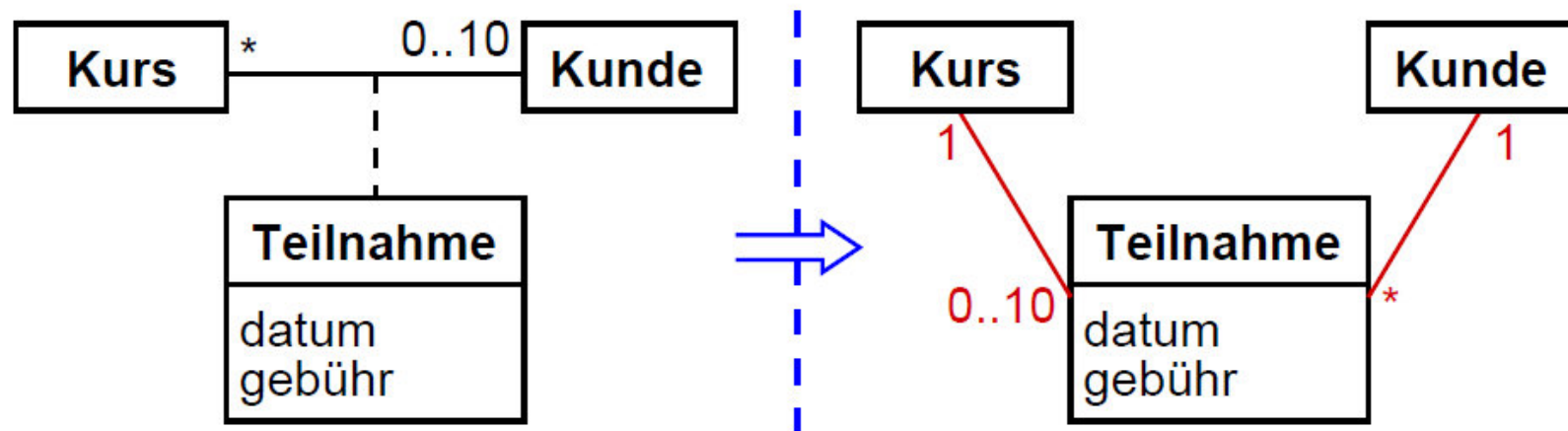


- Bei der Komposition muß ggf. die Lebensdauer-Verwaltung der Teil-Objekte berücksichtigt werden

3.5 Assoziationen ...

Auflösung von Assoziationsklassen

- Assoziationsklassen können durch Koordinator-Klassen ersetzt werden
 - ihre Haupt-Aufgabe ist, sich zu merken, wer wen kennt
- Beispiel:

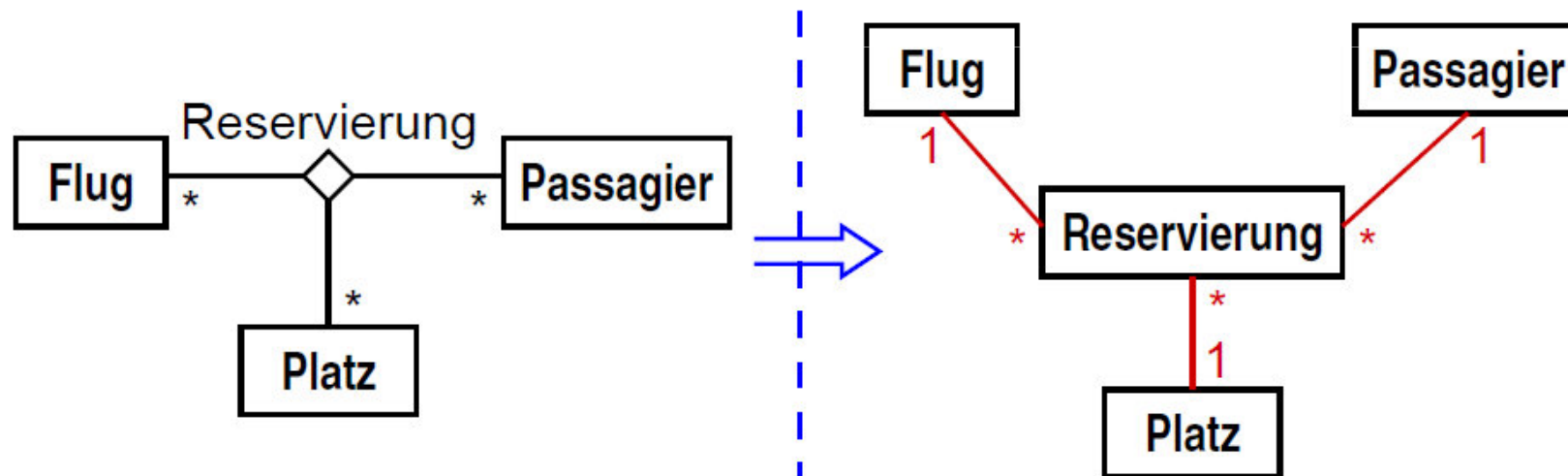


3.5 Assoziationen ...



Auflösung von mehrgliedrigen Assoziationen

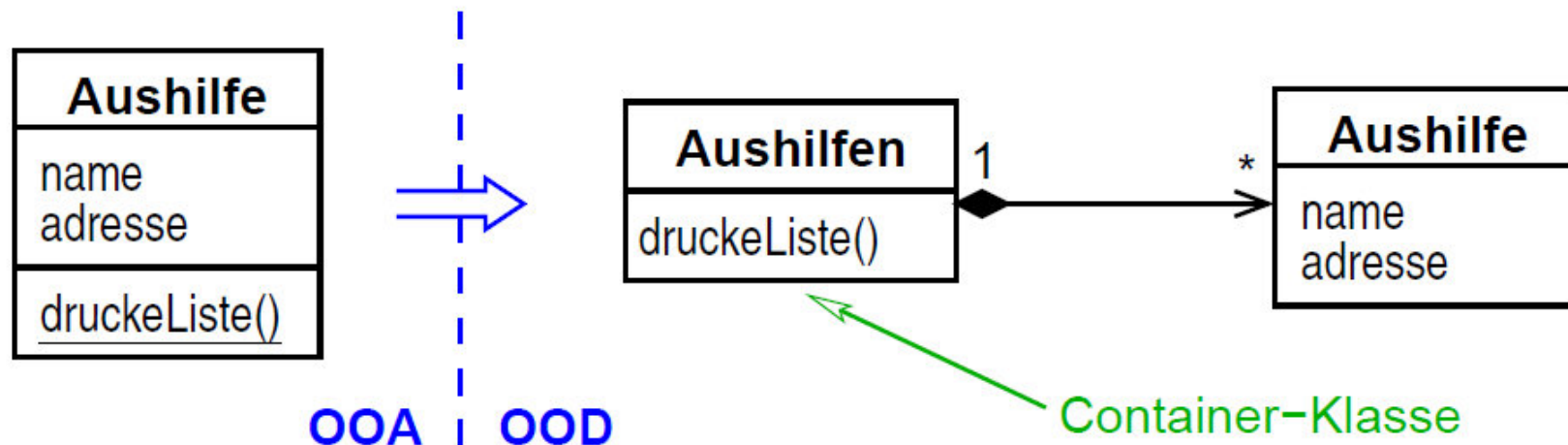
- Mehrgliedrige Assoziationen werden ebenfalls durch Koordinator-Klassen realisiert
- Beispiel:





3.6 Objektverwaltung

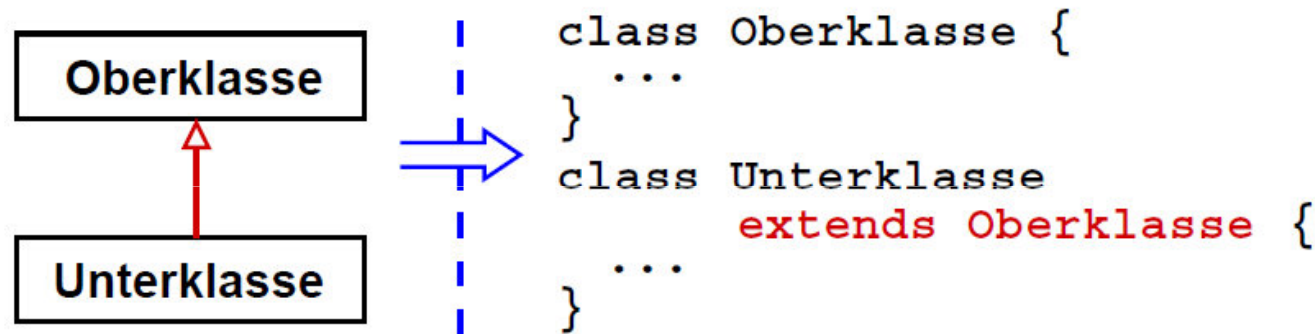
- In der Analysephase besitzt jede Klasse inhärent eine Objektverwaltung
- Im Entwurf ist diese (falls notwendig!) explizit zu modellieren
 - als **Container-Klasse**: verwaltet Menge von Objekten einer anderen Klasse
- Beispiel:



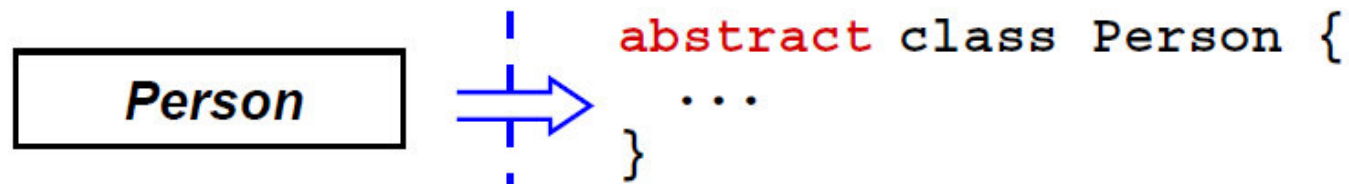


3.7 Generalisierung (Vererbung)

- Anmerkung: keine spezielle Verfeinerung beim Entwurf, hier soll nur die Umsetzung in Java gezeigt werden!



- Umsetzung der Generalisierungsbeziehung in Java:





3.7 Generalisierung (Vererbung) ...

Überschreiben von Methoden

- Beim Überschreiben einer ererbten Methode müssen (in Java 2) Signatur und Ergebnistyp exakt übereinstimmen!

Richtig:

```
class Ober {  
    void op(int p) {...}  
}  
  
class Unter extends Ober {  
    // überschreibt op()  
    // aus Basisklasse  
    void op(int p) { ... }  
}
```

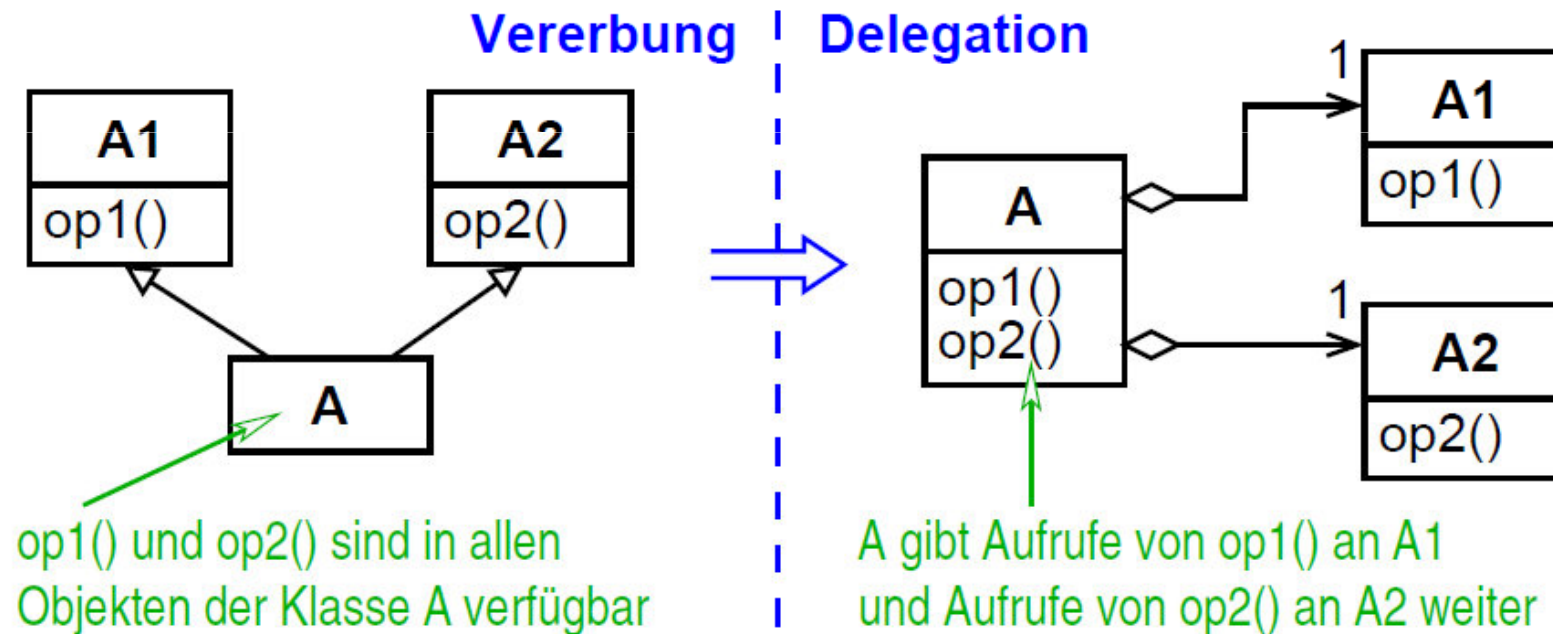
Falsch:

```
class Ober {  
    void op(int p) {...}  
}  
  
class Unter extends Ober {  
    // neue Operation!  
    void op(double p) { ... }  
    // Fehler!  
    int op(int p) { ... }  
}
```

3.7 Generalisierung (Vererbung) ...

Mehrfachvererbung und Delegation

- Java erlaubt im Gegensatz zur UML keine Mehrfachvererbung
- Behelf: **Delegation**
 - Objekt gibt Aufrufe von Methoden an andere Objekte weiter

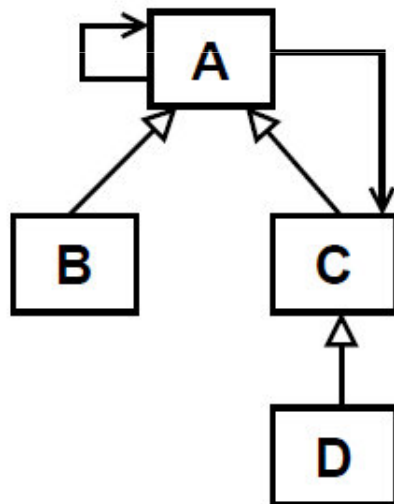


3.7 Generalisierung (Vererbung) ...

Referenzen und Generalisierung

- Referenzen können außer auf Objekte der im Typ angegebenen Klasse auch auf Objekte einer **Unterklass**e dieser Klasse zeigen
- Beispiel:

Klassendiagramm



Java-Code

```

class A {
    A ref1;
    C ref2;
}
  
```

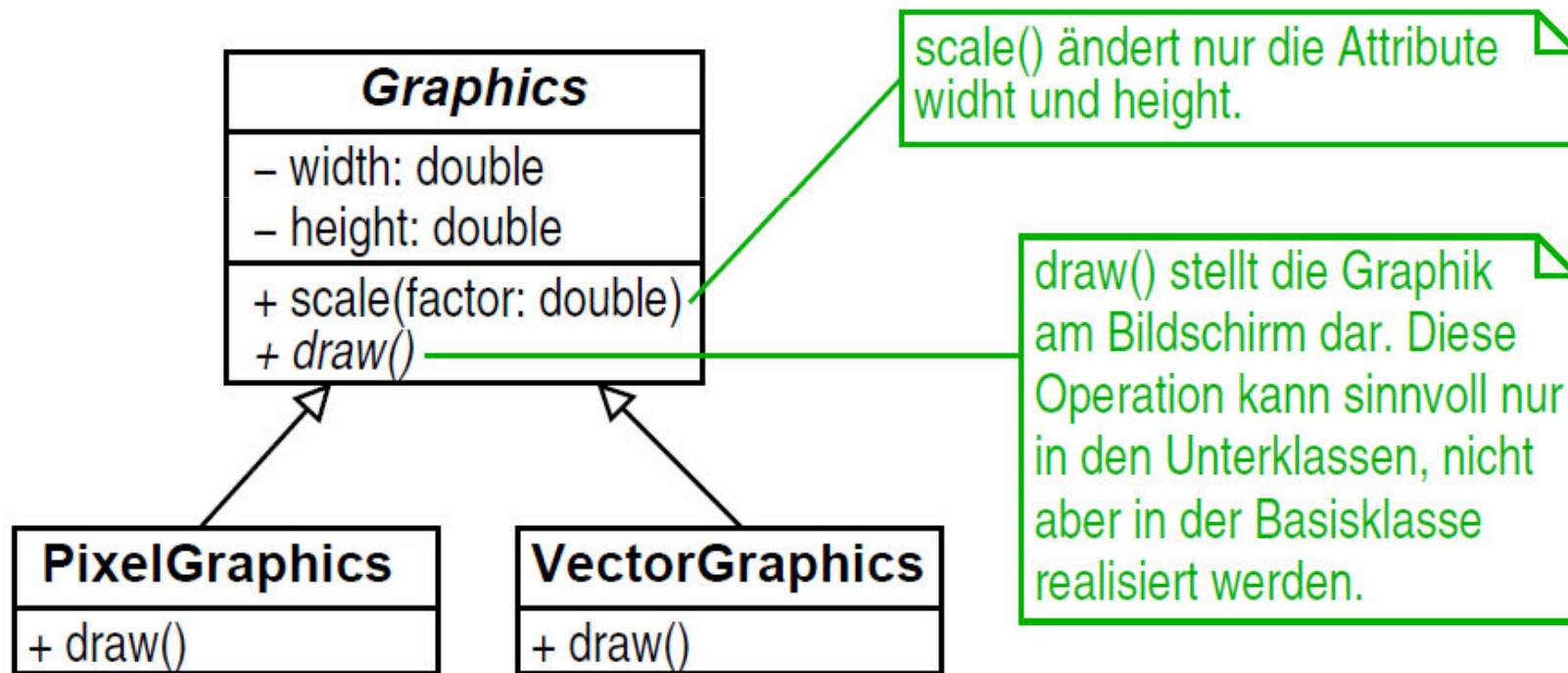
Kann auf Objekte der Klassen
A, B, C und D verweisen

Kann auf Objekte der Klassen
C und D verweisen, nicht aber
auf Objekte der Klassen A und B!



3.8 Abstrakte Operationen und Schnittstellen

- Zur Motivation: ein Beispiel zur Generalisierung
 - eine Graphik in einem Textdokument kann eine Pixelgraphik oder eine Vektorgraphik sein:

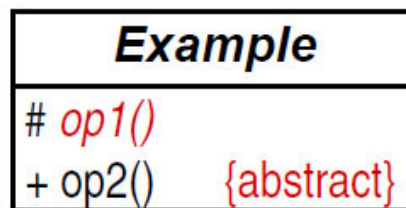


3.8 Abstrakte Operationen und Schnittstellen ...



3.8.1 Abstrakte Operationen

- Eine **abstrakte Operation** einer Klasse wird von der Klasse nur deklariert, nicht aber implementiert
 - die Klasse legt nur Signatur und Ergebnistyp fest
 - die Implementierung muß in einer Unterklasse durch überschreiben der ererbten Operation erfolgen (=> Folie 105, 162)
- Abstrakte Operationen dürfen nur in abstrakten Klassen auftreten
- Darstellung abstrakter Operationen in UML und Java:



Kursivschrift oder
Zusatz {abstract}

```
abstract class Example {  
    protected abstract void op1() ;  
    public abstract void op2() ;  
}
```

Abstrakte Operationen besitzen keinen Rumpf,
d.h. keine Implementierung.



3.8.1 Abstrakte Operationen ...

Umsetzung des Beispiels (Kap.3, Folie 40) in Java

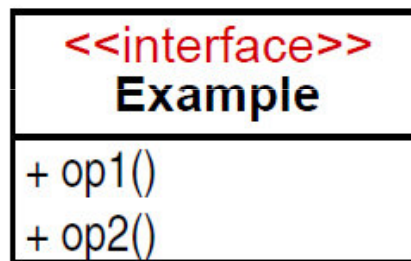
```
abstract class Graphics {  
    private double width;  
    private double height;  
    public void scale( double factor) { ... }  
    public abstract void draw();  
}  
class PixelGraphics extends Graphics {  
    ...  
    public void draw() { ... }  
}  
class VectorGraphics extends Graphics {  
    ...  
    public void draw() { ... }  
}
```

3.8 Abstrakte Operationen und Schnittstellen ...



3.8.2 Schnittstellen (*Interfaces*)

- Eine **Schnittstelle** beschreibt eine Menge von Signaturen (inkl. Ergebnistyp) von Operationen
 - oder: eine Schnittstelle ist eine (abstrakte) Klasse, die **nur** abstrakte Operationen enthält
- Darstellung in UML und Java:



```
interface Example {
    public void op1() ;
    public void op2() ;
}
```

- Die Operationen einer Schnittstelle sind
 - immer abstrakt⇒spezielle Kennzeichnung kann entfallen
 - immer öffentlich⇒in Java kann public ggf. entfallen (= > 3.9)



3.8.2 Schnittstellen (*Interfaces*) ...

Motivation

- Schnittstellen definieren "Dienstleistungen" für aufrufende Klassen, sagen aber nichts über deren Implementierung aus
 - funktionale Abstraktionen, die festlegen **was** implementiert werden soll, aber nicht **wie**
- Schnittstellen realisieren damit das Geheimnisprinzip in der stärksten Form
 - Java-Klassen verhindern über Sichtbarkeiten zwar den Zugriff auf Interna der Klasse, ein Programmierer kann diese aber trotzdem im Java-Code der Klasse lesen
 - der Java-Code einer Schnittstelle enthält nur die öffentlich sichtbaren Definitionen, nicht die Implementierung



3.8.2 Schnittstellen (*Interfaces*) ...

Beispiel: Datenstruktur Keller

Als Java-Klasse

```
class Keller {  
    private int[] stack;  
    private int sp = -1;  
    public void push(int i) {  
        stack[++sp] = i;  
    }  
    public int pop() {  
        return stack[sp--];  
    }  
    public int top() {  
        return stack[sp];  
    }  
}
```

Als Java-Schnittstelle

```
interface Keller {  
    public void push(int i);  
    public int pop();  
    public int top();  
}
```

Java-Code enthält keinerlei
Implementierungsdetails

Implementierung ist nicht von anderen
Klassen aus zugreifbar, aber trotzdem
für Programmierer lesbar



3.8.2 Schnittstellen (*Interfaces*) ...

Schnittstellen und Klassen

- Schnittstellen sind von ihrer Struktur und Verwendung her praktisch identisch mit (abstrakten) Klassen
 - sie können wie abstrakte Klassen nicht instanziiert werden
 - Referenzen auf Schnittstellen (und auch abstrakte Klassen) sind aber möglich
 - sie können auf Objekte zeigen, die die Schnittstelle implementieren
- Klassen können von Schnittstellen "erben"
 - "vererbt" werden nur die Signaturen der Operationen
 - die Klassen müssen diese Operationen selbst implementieren
 - man spricht in diesem Fall von einer **Implementierungs-Beziehung** statt von Generalisierung



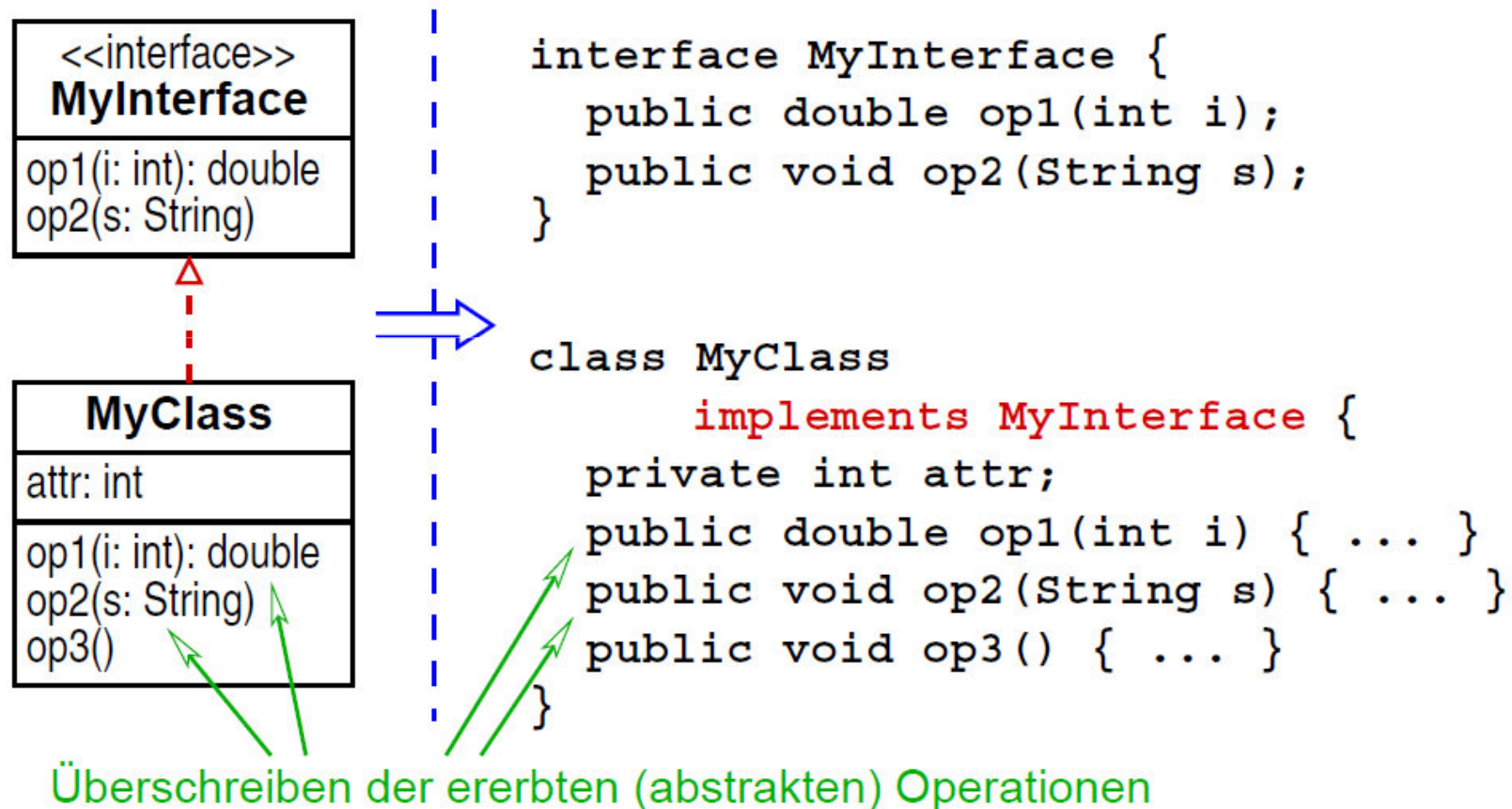
3.8.2 Schnittstellen (*Interfaces*) ...

Die Implementierungs-Beziehung

- Sie besagt, daß eine Klasse die Operationen einer Schnittstelle implementiert
 - die Klasse erbt die abstrakten Operationen
 - d.h. deren Signaturen (incl. Ergebnistyp)
 - diese müssen dann geeignet überschrieben werden
 - werden nicht alle Operationen überschrieben (d.h. implementiert), so bleibt die erbende Klasse abstrakt
 - die überschreibenden Operationen müssen öffentlich sein
 - die Klasse kann zusätzlich weitere Operationen und Attribute definieren
- Eine Klasse kann mehrere Schnittstellen implementieren
 - eine Art Mehrfachvererbung, auch in Java erlaubt

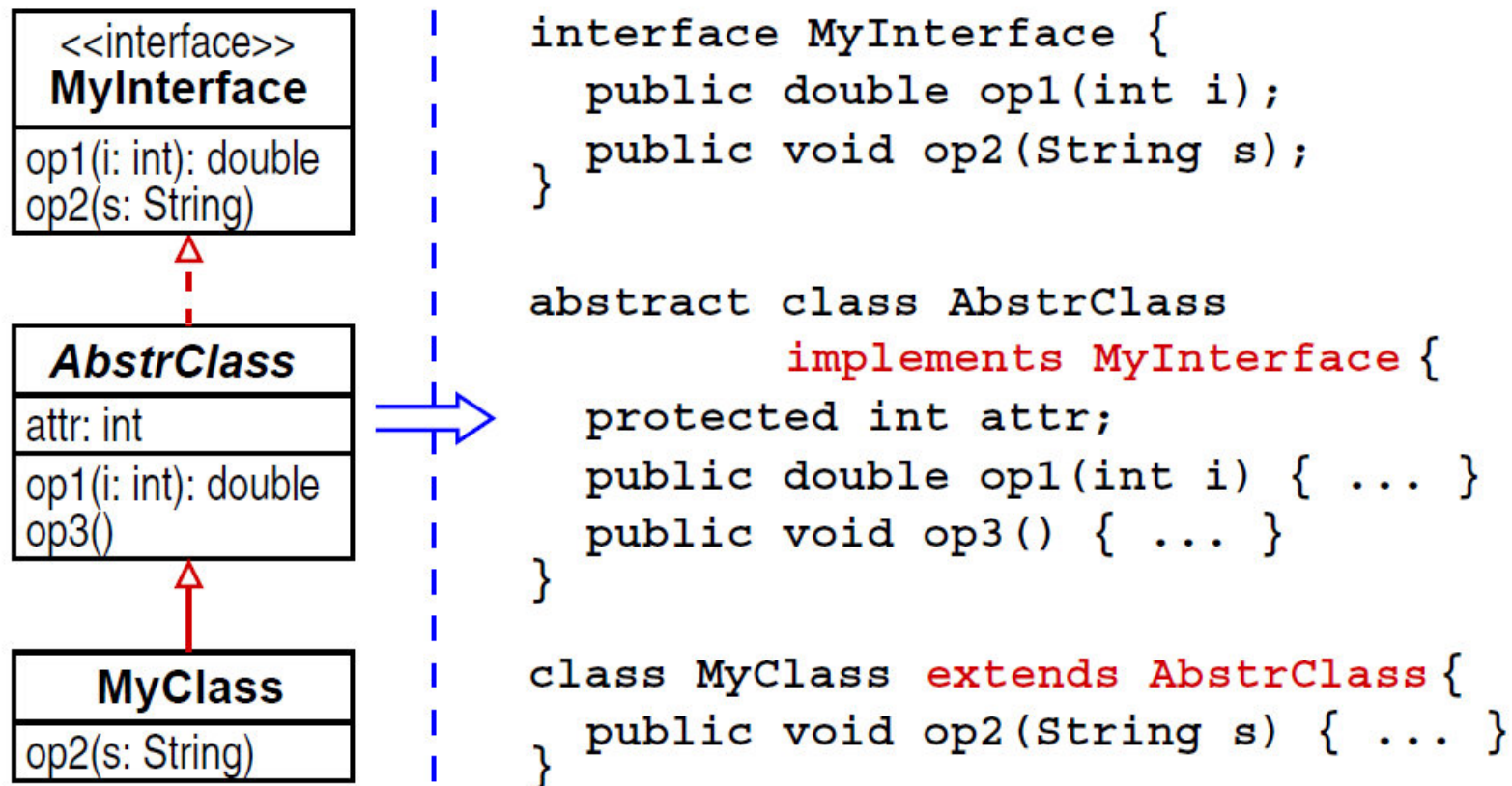
3.8.2 Schnittstellen (*Interfaces*) ...

Implementierungs-Beziehung in UML und Java



3.8.2 Schnittstellen (*Interfaces*) ...

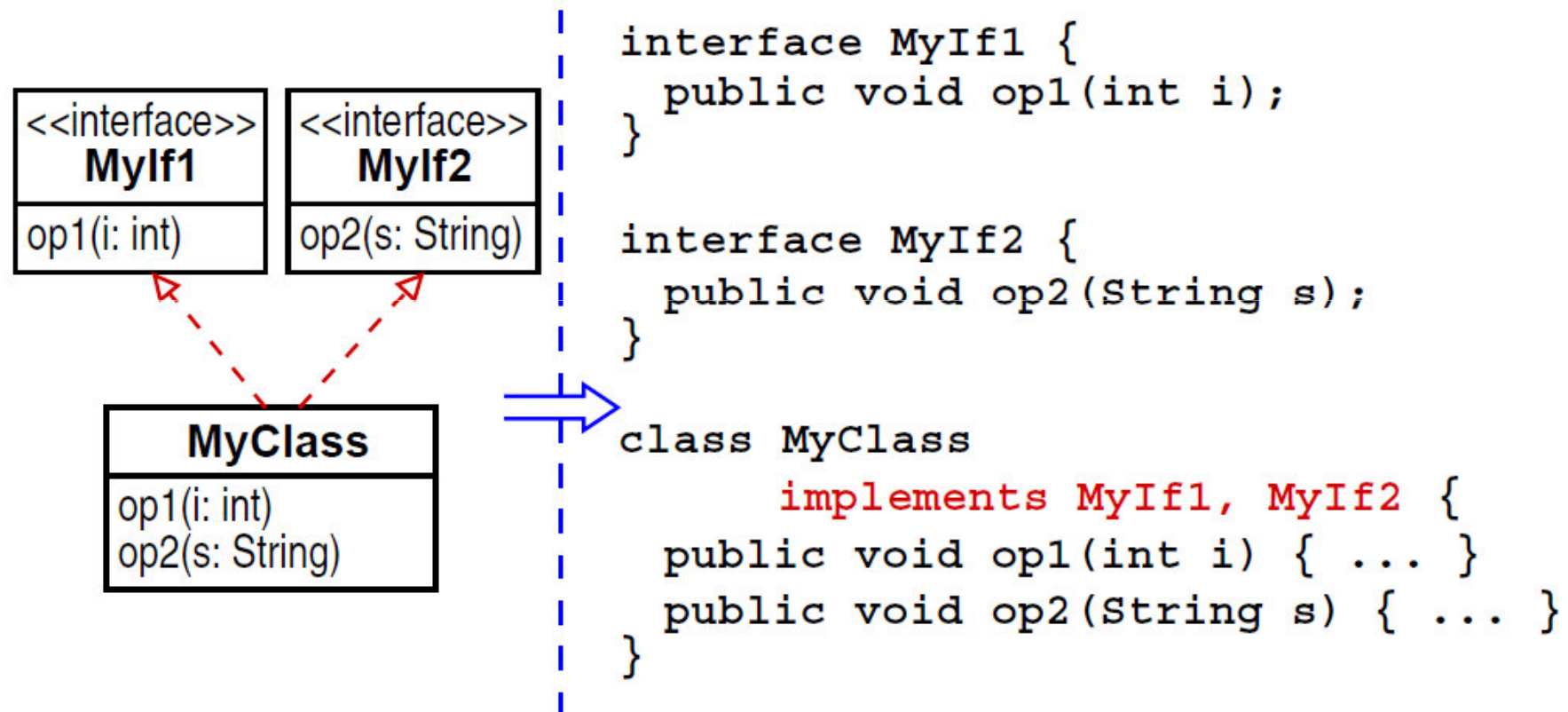
Implementierung einer Schnittstelle über eine abstrakte Klasse





3.8.2 Schnittstellen (*Interfaces*) ...

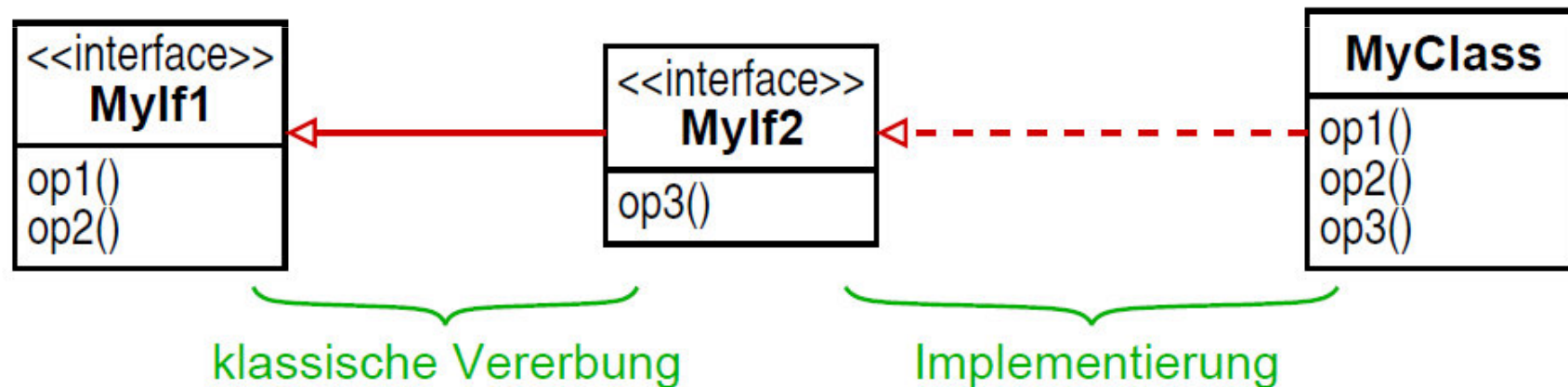
Implementierung mehrerer Schnittstellen



3.8.2 Schnittstellen (*Interfaces*) ...

Schnittstellen und Vererbung

- Schnittstellen können von anderen Schnittstellen erben (analog zu Klassen)
 - Darstellung in UML und Java wie bei normaler Vererbung
- Die implementierende Klasse muß die Operationen "ihrer" Schnittstelle und aller Ober-Schnittstellen implementieren:





3.8.2 Schnittstellen (*Interfaces*) ...

Attribute in Schnittstellen

- UML 2 erlaubt zusätzlich zu abstrakten Operationen auch die Definition von Attributen in einer Schnittstelle
- Java erlaubt in Schnittstellen nur öffentliche, unveränderliche und initialisierte Klassenattribute (d.h. Konstanten)
 - z.B.: *public static final double M_PI = 3.14159;*
 - Konvention: Namen vollständig in Großbuchstaben, mit _ als Trennzeichen
 - *public*, *static* und *final* können auch entfallen (wird dann implizit angenommen)



3.8.2 Schnittstellen (*Interfaces*) ...

Verwendung von Schnittstellen

- Schnittstellen werden häufig verwendet, wenn ein Programm(teil) bestimmte Dienste (Operationen) von nicht näher bekannten Klassen benötigt
- Beispiel: Plugins in einem WWW-Browser
 - Browser nutzt feste Schnittstelle zur einheitlichen Kommunikation mit unterschiedlichen Plugins
- Beispiel: Ereignis-Behandlung in Java
 - Elemente der graphischen Bedienoberfläche nutzen eine feste Schnittstelle zur Weitergabe von Ereignissen an Klassen des Anwendungsprogramms
 - die betroffenen Klassen müssen (u.a.) diese Schnittstelle implementieren



3.8.2 Schnittstellen (*Interfaces*) ...

Praktisches Beispiel

- Beim Drücken der ESC-Taste soll das Fenster einer Anwendung geschlossen werden
- Fenster gibt Tastatur-Ereignisse über die Schnittstelle *KeyListener* weiter:

```
interface KeyListener extends EventListener
{
    public void keyPressed( KeyEvent event);
    public void keyReleased( KeyEvent event);
    public void keyTyped( KeyEvent event);
}
```

- Eine der Anwendungsklassen implementiert diese Schnittstelle
 - hier ist nur *keyPressed* relevant, alle anderen Operationen können leer implementiert werden



3.8.2 Schnittstellen (*Interfaces*) ...

Praktisches Beispiel ...

class MyAppClass implements KeyListener {

```
·  
· // Attribute und Methoden der Anwendungsklasse  
·  
public void keyPressed(KeyEvent event) {  
  
·  
· // Falls gedrückte Taste = ESC-Taste: schließe Fenster  
·  
}  
  
public void keyReleased(KeyEvent event) {}  
public void keyTyped(KeyEvent event) {}  
}
```

Leere Implementierung (keine Anweisungen!)



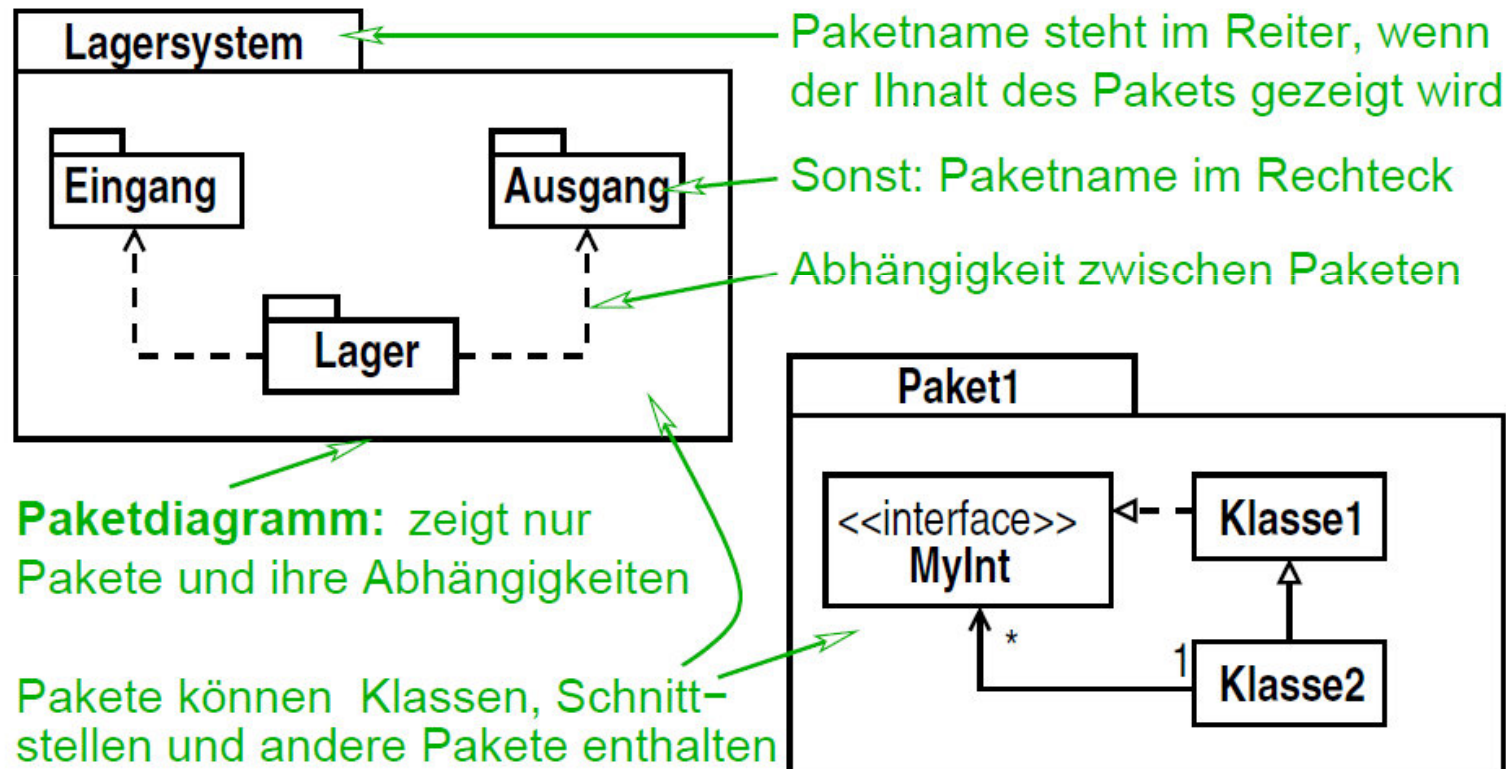
3.9 Pakete

- Große Software-Projekte führen zu einer hohen Zahl von Klassen und Schnittstellen
- Pakete erlauben die Strukturierung von UML-Diagrammen und Java-Code
 - Zusammenfassung logisch zusammengehöriger Klassen und Schnittstellen in einem Paket
 - ein Paket entspricht einem Modul bzw. einer SW-Bibliothek
- Pakete können auch weitere Pakete enthalten
 - hierarchische Strukturierung der Software
- Pakete erlauben verfeinerte Spezifikation von Sichtbarkeiten
- Sie helfen zudem, Namenskonflikte zu vermeiden
 - gleiche Namen in unterschiedlichen Paketen möglich

3.9 Pakete ...

Pakete in UML

➔ Darstellung (Beispiele):





3.9 Pakete ...

Pakete in Java

- Am Anfang einer Java Quellcode-Datei kann die folgende Anweisung stehen:

```
package <Paketname> ;
```

- damit gehören alle in dieser Datei definierten Klassen und Schnittstellen dem genannten Paket an

- Beispiel:

```
package Musikverwaltung;
```

```
class Musikstück {
```

```
...
```

```
}
```

```
class Musikmedium {
```

```
...
```

```
}
```

Beide Klassen liegen im
Paket "Musikverwaltung"



3.9 Pakete ...

Pakete in Java ...

- Java definiert ein *Default*-Paket
 - in diesem Paket sind alle Klassen und Schnittstellen, die nicht explizit einem Paket zugeordnet wurden

Benennung von Klassen in Paketen

- Klassen in verschiedenen Paketen können denselben Namen besitzen
- Sie werden daher über hierarchische Namen angesprochen:
 - in Java: `Paket.Unterpaket1.Unterpaket2.Klasse`
 - in UML: `Paket::Unterpaket1::Unterpaket2::Klasse`
- Anmerkung: dies gilt genauso für Schnittstellen und Pakete



3.9 Pakete ...

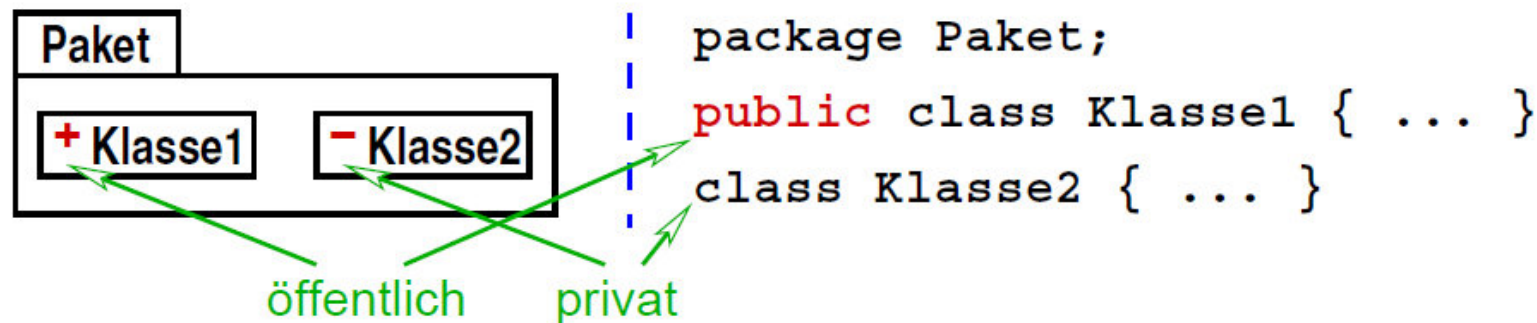
Importieren von Paketen in Java

- Um Klassen (bzw. Schnittstellen) in Java zu nutzen, gibt es verschiedene Möglichkeiten:
 - Angabe des vollen Namens, z.B.
`java.util.Date datum;` // Klasse Date aus Paket java.util
 - Importieren der Klasse:
`import java.util.Date;` // Am Anfang der Programmdatei
...
`Date datum;`
 - Importieren aller Klassen eines Pakets:
`import java.util.*;` // Am Anfang der Programmdatei
- Bei Klassen aus dem eigenen Paket reicht immer der einfache Klassenname aus

3.9 Pakete ...

Pakete und Sichtbarkeiten

- Für die in einem Paket enthaltenen Klassen können Sichtbarkeiten definiert werden:
 - **public**: Klasse ist für alle Pakete sichtbar
 - **private**: Klasse ist nur innerhalb ihres Pakets sichtbar
- Darstellung:

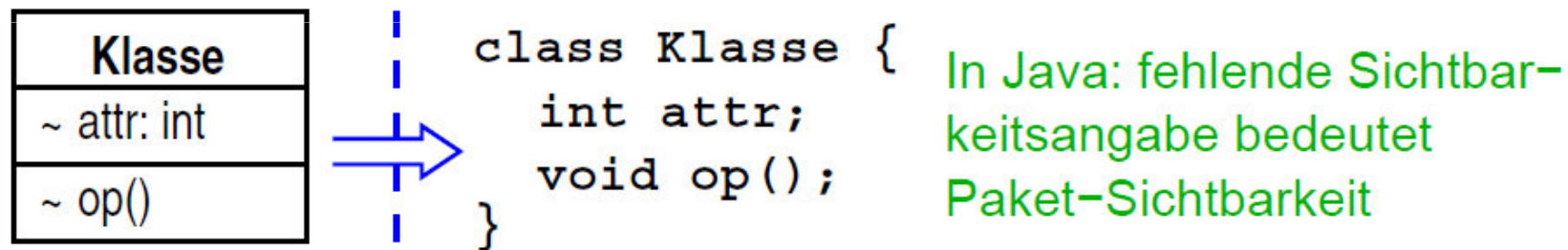


- In einer öffentlichen **Schnittstelle** sind automatisch auch alle Operationen öffentlich (ohne Angabe von public)

3.9 Pakete ...

Pakete und Sichtbarkeiten ...

- Für Attribute und Operationen von Klassen (und Schnittstellen) kann eine weitere Sichtbarkeit definiert werden:
 - **package**: sichtbar in allen Klassen des selben Pakets und nur in diesen
- Darstellung:

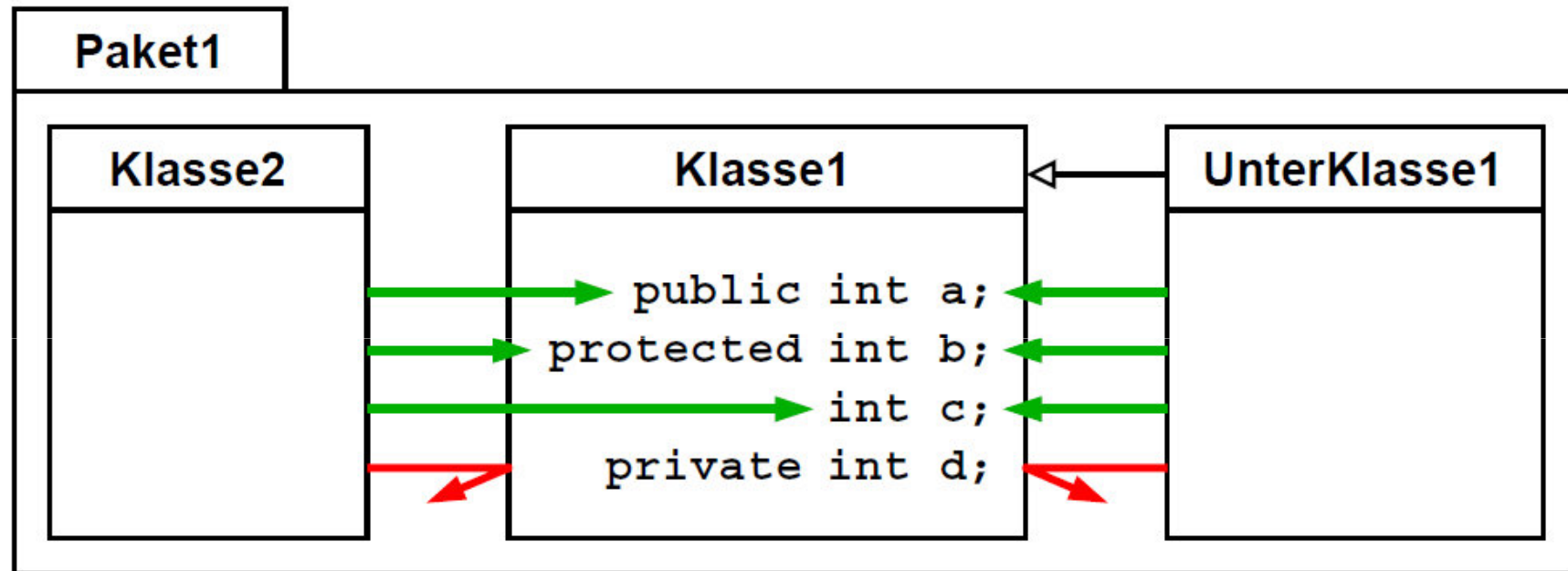


- Java definiert im Zusammenhang mit Paketen auch die Bedeutung von *protected* neu (und anders als UML!):
 - sichtbar in Unterklassen und Klassen des selben Pakets

3.9 Pakete ...



Java: Zugriffe innerhalb des Pakets



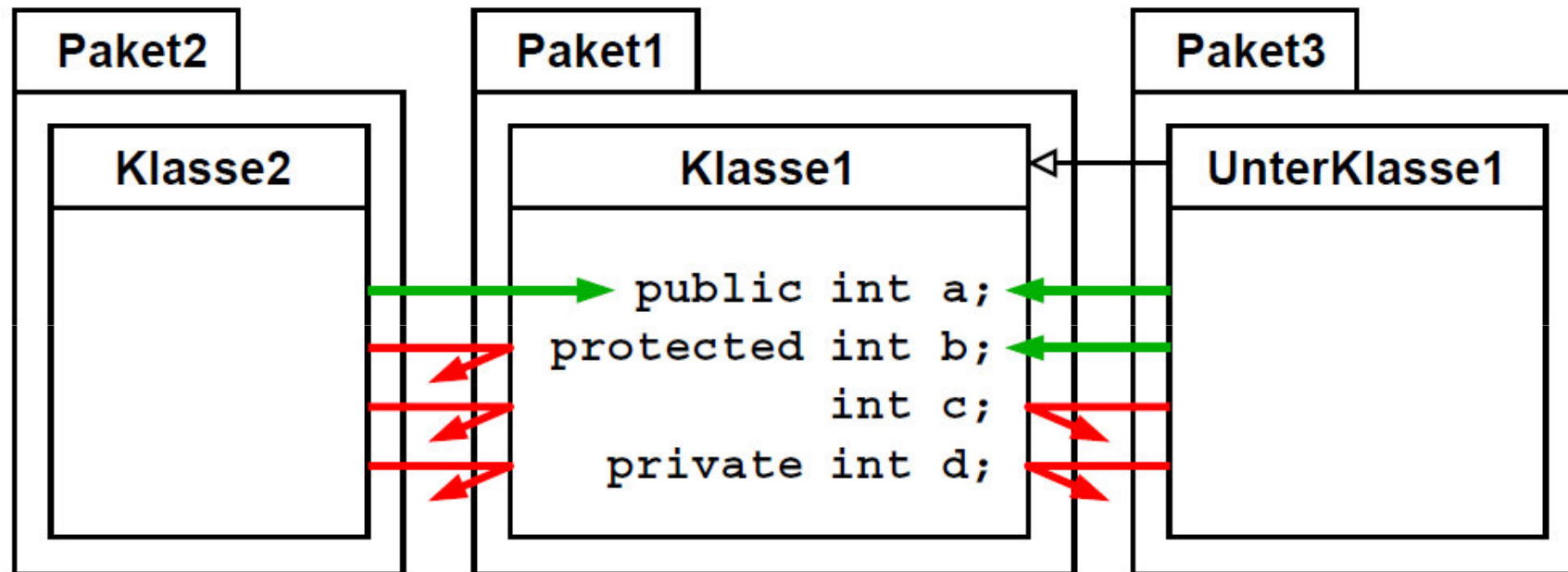
Anmerkung: dies ist kein UML-Diagramm!

➤ Gilt genauso für Methoden

3.9 Pakete ...



Java: Zugriffe innerhalb des Pakets



Anmerkung: dies ist kein UML-Diagramm!

➤ Gilt genauso für Methoden

3.9 Pakete ...



Zusammenfassung der Sichtbarkeitsangaben in Java

Sichtbarkeit	Klasse	Operation	Attribut	Bedeutung
(keine)	●	●	●	nur im Paket zugreifbar
public	●	●	●	für alle zugreifbar
protected		●	●	im eigenen Paket und in Unterklassen zugreifbar
private		●	●	nur in eigener Klasse zugreifbar



3.10 Zusammenfassung

- Aufgabe beim Entwurf (in der OFP): Verfeinerung des Klassendiagramms
 - Typen für Attribute
 - Typen und Parameter von Operationen (Signatur)
 - Sichtbarkeiten: *public*, *protected*, *private*, *package*
 - Assoziationen: Navigierbarkeit und Realisierung durch Referenzen und Koordinator-Klassen
 - Objektverwaltung: Einführen von *Container*–Klassen
 - Einführen von abstrakten Operationen und Schnittstellen
 - Gliederung der Klassen in Pakete