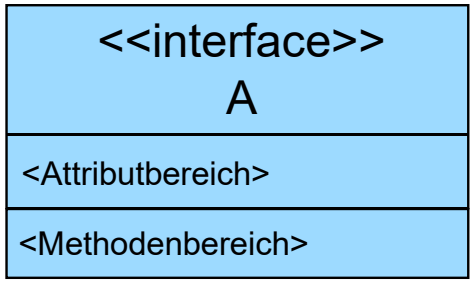
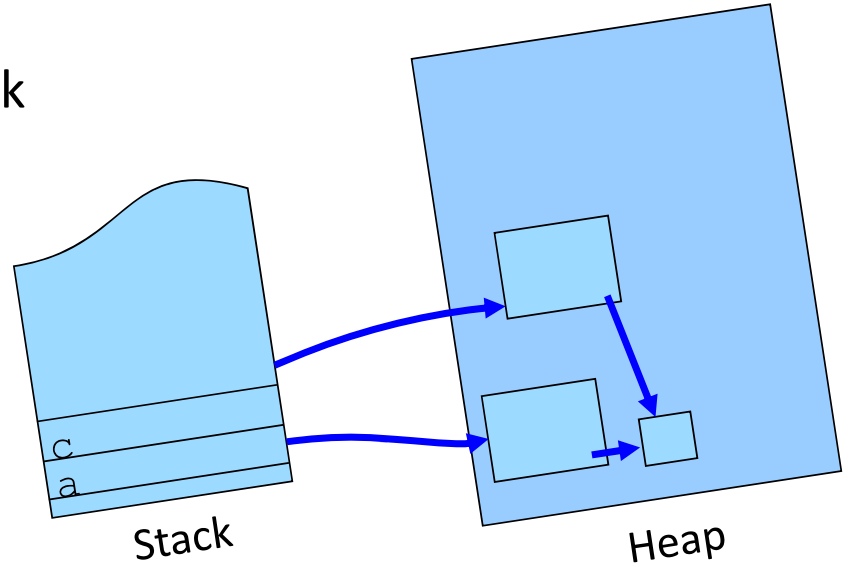
Vorlesung Programmieren
Thema 8: Objektorientierung (III)

Olaf Herden
Fakultät Technik
Studiengang Informatik



this

super



Stand: 01/2024

Gliederung

- Operatoren `this/super/instanceof`
- Parameterübergabe
- Beziehungen zwischen Klassen
- Schnittstellen
- Polymorphie
- Kopieren von Objekten
- Klassenbibliotheken

this-Operator (I): Motivation (I)

- Beispiel:

```
(1) public class Mitarbeiter{
(2)
(3)   private int persNr;
(4)   private String name;
(5)
(6)   public Mitarbeiter(String name, int persNr){
(7)      ...
(8)   }
(9) }
```

- Namenskonflikt in Rumpf des Konstruktors: Attribute `persNr` und `name` sowohl Instanz- als auch Übergabeattribute in Parameterliste
- Es muss gelöst werden:
  - Welches Attribut ist bei Verwendung von `persNr` bzw. `name` im Rumpf des Konstruktors gemeint
  - Wie werden die anderen Attribute mit Bezeichnung `persNr` bzw. `name` angesprochen

this-Operator (II): Motivation (II)

- Beispiel:

```
( 1) public class Mitarbeiter{
( 2)
( 3)   private int persNr;
( 4)   private String name;
( 5)   private String ort;
( 6)
( 7)   public Mitarbeiter(String name, int persNr){...}
( 8)   public Mitarbeiter(String name, int persNr, String ort){
( 9)     ...
(10)   }
(11) }
```

- Zweiter Konstruktor Obermenge des ersten
- Wünschenswert: Zweiter Konstruktor ruft ersten auf
- Anweisung `Mitarbeiter(name,persNr);` scheitert

this-Operator (III): Lösung (I)

- Abhilfe in beiden Beispielen: this-Operator, zeigt auf aktuelles Objekt
- Beispiel 1: mittels this Ansprechen der Objektvariablen

```
( 1) public class Mitarbeiter{
( 2)
( 3)   private int persNr;
( 4)   private String name;
( 5)
( 6)   public Mitarbeiter(String name, int persNr){
( 7)     this.persNr = persNr;
( 8)     this.name   = name;
( 9)   }
(10) }
```

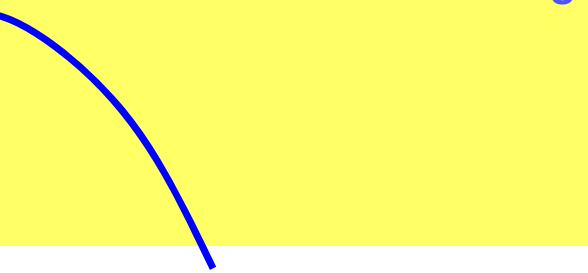
Ansprechen der Objektvariablen

Ansprechen der lokalen Variablen

this-Operator (IV): Lösung (II)

- Beispiel 2: mittels this Aufruf des ersten Kontruktors aus dem zweiten heraus

```
( 1) public class Mitarbeiter{
( 2)
( 3)   private int persNr;
( 4)   private String name;
( 5)   private String ort;
( 6)
( 7)   public Mitarbeiter(String name, int persNr){...}
( 8)   public Mitarbeiter(String name, int persNr,
( 9)                                      String ort){
(10)     this(name, persNr);
(11)     this.ort = ort;
(12)   }
(13) }
```



super-Operator (I): Motivation

- Beispiel:

```
(1) public class Mitarbeiter{
(2)   ...
(3)   public void drucken(){
(4)     System.out.println(persNr);
(5)     System.out.println(name);
(6)     ...
(7)   }
(8) }
```

```
(1) public class Arbeiter extends Mitarbeiter{
(2)   private float stundensatz;
(3)   public void drucken(){
(4)     System.out.println(persNr);
(5)     System.out.println(name);
(6)     ...
(7)     System.out.println(stundensatz);
(8)  }
(9) }
```

- Methode `drucken()` in Subklasse: Wiederholung gleichnamiger Methode der Superklasse
- Wünschenswert: Aufruf Methode der Superklasse in Subklasse

super-Operator (II): Lösung

- Abhilfe: Operator `super`, verweist auf Superklasse
- Lösung:

```
(1) public class Arbeiter extends Mitarbeiter{
(2)   private float stundensatz;
(3)   public void drucken(){
(4)     super.drucken();
(5)     System.out.println(stundensatz);
(6)   }
(7) }
```

Aufrufen Methode `drucken()` der Superklasse

super-Operator (III): Konstruktor

- Aufruf Konstruktor der Superklasse durch Operator `super`
- Beispiel:

```
(1) public class Mitarbeiter{
(2)   ...
(3)   public Mitarbeiter(String name,int persNr,String ort){
(4)     this(name,persNr);
(5)     this.ort=ort;
(6)   }
(7) }
```

```
(1) public class Arbeiter extends Mitarbeiter{
(2)   ...
(3)   public Arbeiter(String name,int persNr,String ort,
(4)                                          int stundenSatz){
(5)   super(name, persNr, ort);
(6)     this.stundenSatz = stundenSatz;
(7)   }
(8) }
```

Aufruf Konstruktor der Superklasse

`super`-Operator (IV): Anmerkungen

- Regeln für `super` in Verbindung mit Konstruktoren:
  - Jede Subklasse muss einen Konstruktor definieren, der einen Konstruktor der Superklasse mittels `super` als erste Anweisung des Konstruktorrumpfes aufruft
  - Ausnahme: Wenn die Superklasse einen parameterlosen Konstruktor besitzt, wird dieser automatisch aufgerufen

- Anmerkung: Mehrstufiger Aufruf `super.super.<methode>` ist verboten

`instanceof`-Operator (I)

- Feststellen Zugehörigkeit eines Objektes zu Klasse
- Syntax: `<Objektname> instanceof <Klassenname>`
- Semantik:
  - Liefert `true`, falls Objekt Instanz der Klasse (oder einer Superklasse) ist, ansonsten `false`
- Beispiel: Klassen `Mitarbeiter` und `Arbeiter` wie in letzten Beispielen

instanceof-Operator (II)

```
( 1) public class Anwendung{
( 2)
( 3)   public static void main(String[] args){
( 4)
( 5)     Mitarbeiter meier   = new Mitarbeiter();
( 6)     Arbeiter    mueller = new Arbeiter();
( 7)
( 8)     System.out.println(meier instanceof Mitarbeiter);
( 9)     System.out.println(meier instanceof Arbeiter);
(10)     System.out.println(mueller instanceof Mitarbeiter);
(11)     System.out.println(mueller instanceof Arbeiter);
(12)   }
(13) }
```

führt zur Ausgabe

```
true
false
true
true
```

Übersicht

- Operatoren `this/super/instanceof`
- Parameterübergabe
- Beziehungen zwischen Klassen
- Schnittstellen
- Polymorphie
- Kopieren von Objekten
- Klassenbibliotheken

Arten

- Call by Value: Wertübergabe
  - Übergebener Wert wird kopiert
  - Änderungen in Methode bleiben für aufrufenden Wert folgenlos
- Call by Reference: Referenzübergabe
  - Übergebener Wert wird nicht kopiert
  - Stattdessen Referenz (ein Verweis) auf diesen Wert
  - Änderungen in Methode verändern aufrufenden Wert
- In Java:
  - Primitive Typen (z.B. `int` oder `float`): Übergabe mittels Call by Value
  - Objekte: Übergabe mittels Call by Reference

Realisierung in Java (I): Beispiel

```
( 1) public class Ueber{
( 2)   static void veraendern(int i, Mitarbeiter m){
( 3)     i = i + 5;
( 4)     m.setPersNr(4712);
( 5)   }
( 6)
( 7)   public static void main(String[] args){
( 8)     int j = 7;
( 9)     Mitarbeiter meier = new Mitarbeiter("Meier",4711);
(10)     veraendern(j,meier);
(11)     System.out.println(j);
(12)     System.out.println(meier.getPersNr());
(13)   }
(14) }
```

führt zur Ausgabe

7

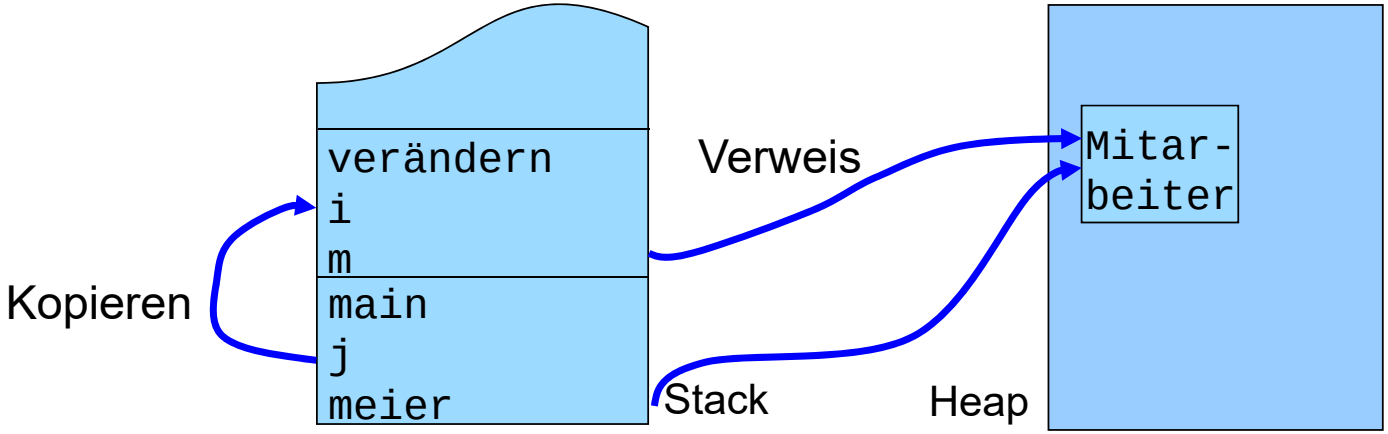
4712

Call by Value: Addition in Methode `veraendern` ohne Auswirkung auf Aufrufparameter `j`

Call by Reference: Manipulation in Methode `veraendern` verändert Wert von Objekt `meier`

Realisierung in Java (II): Erläuterung

- In `main`-Methode Anlegen Variable primitiven Typs (`j` vom Typ `int`) und Objekt (`meier` als Instanz der Klasse `Mitarbeiter`)
- Aufruf Methode `veraendern`:
  - Wert Variable `j` wird nach `i` kopiert
  - Anlegen Verweis auf `meier` für `m`



  - Manipulation beider Variablen in Methode `veraendern`
  - Änderungen an `m` ändern auch Übergabeparameter

Übersicht

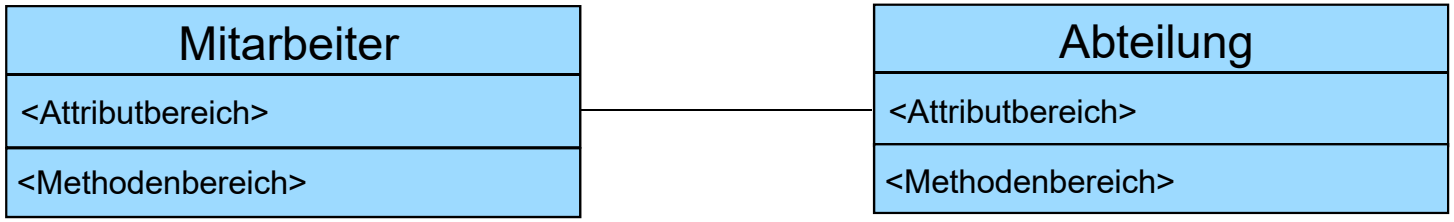
- Operatoren `this/super/instanceof`
- Parameterübergabe
- Beziehungen zwischen Klassen
- Schnittstellen
- Polymorphie
- Kopieren von Objekten
- Klassenbibliotheken

Beziehungen

- Bisher: Klassen im Super-/Subklasse-Verhältnis
- Andere Beziehungen zwischen Klassen denkbar
- Beispiel:

- Klassen haben Bezug zueinander:
  - Mitarbeiter gehören zu einer Abteilung
  - Abteilungen haben Mitarbeiter
- Beziehung nichtals Vererbung darstellbar, weder spezialisiert Mitarbeiter eine Abteilung noch umgekehrt
- Allgemeinerer Beziehungstyp: Assoziation
- Darstellung: Einfache Kante zwischen Klassen

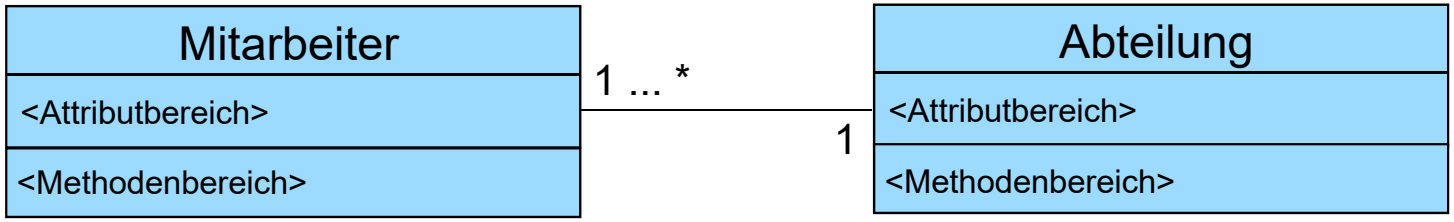
Assoziationen (I)



- Damit: Beziehungstyp Assoziation dargestellt
- Weitere Angaben wünschenswert:
  - Wie viele Mitarbeiter kann eine Abteilung haben?
  - Zu wie vielen Abteilungen kann ein Mitarbeiter gehören?
- Dazu: Markierung Assoziationskante mit Multiplizitäten (Kardinalitäten), geben Anzahl an Assoziation teilnehmender Objekte an
- Notation: Angabe Mindest- und Maximalzahl, getrennt durch „...“ (Bsp.: 3 ... 7)
- Regeln:
  - Wenn Mindestzahl = Maximalzahl $\Rightarrow$ Wert schrieben (Bsp.: 5 ... 5 entspricht 5)
  - Wenn Maximalzahl unbestimmt $\Rightarrow$ Verwende Symbol „*“ (Bsp.: 1 ... *)

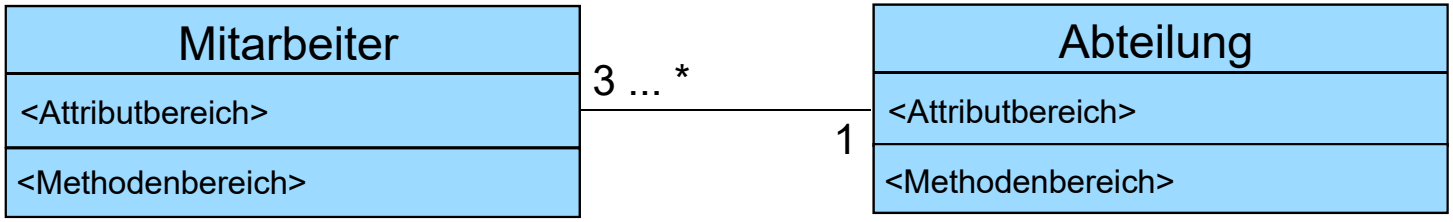
Assoziationen (II)

- Beispiel 1:



  - „Eine Abteilung hat mindestens eine bis beliebig viele Mitarbeiter“
  - „Ein Mitarbeiter gehört zu genau einer Abteilung“

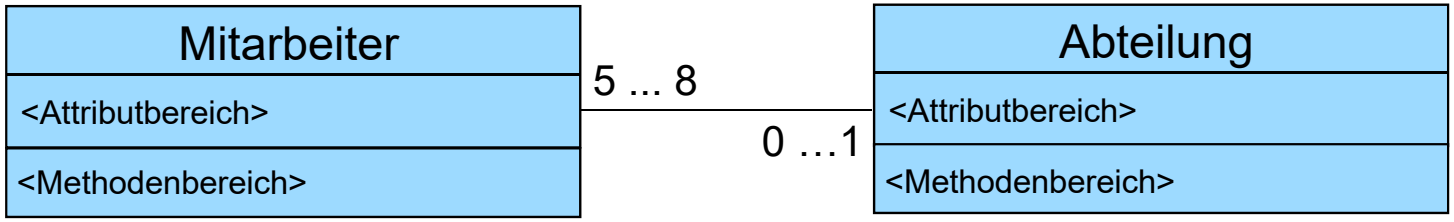
- Beispiel 2:



  - „Eine Abteilung hat mindestens drei bis beliebig viele Mitarbeiter“
  - „Ein Mitarbeiter gehört zu genau einer Abteilung“

Assoziationen (III)

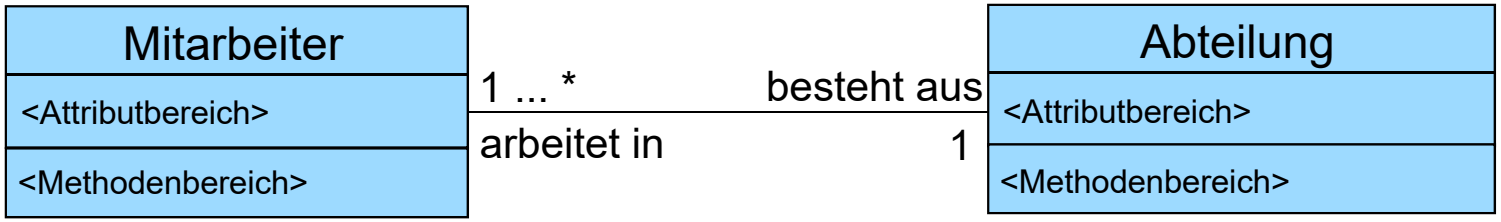
- Beispiel 3:



  - „ Eine Abteilung hat mindestens fünf, aber höchstens acht Mitarbeiter“
  - „Ein Mitarbeiter gehört zu einer oder keiner Abteilung“

Assoziationen (IV): Rollen

- Manchmal wünschenswert: Angabe Art der Assoziation, d.h. welche Rolle Objekte in Assoziation spielen, z.B.
  - „Eine Abteilung besteht aus Mitarbeitern“
  - „Ein Mitarbeiter arbeitet in einer Abteilung“
- UML: Ergänzung Assoziationen um Rollen, z.B.



- Anmerkung: Häufig Rollenbezeichnungen offensichtlich oder sind allgemein (z.B. hat, besteht aus, ...) $\Rightarrow$ Weglassen

Assoziationen (V): Java

- Aufnehmen Elemente der assoziierten Klasse in Klasse
- Beispiel:

```
(1) public class Mitarbeiter{
(2)    ...
(3)    private Abteilung abt;
(4)    ...
(5) }
```

Assoziation mit Multiplizität 1: Eintragen Attribut der anderen Klasse

```
(1) public class Abteilung{
(2)    ...
(3)    private Mitarbeiter[] mit;
(4)    ...
(5) }
```

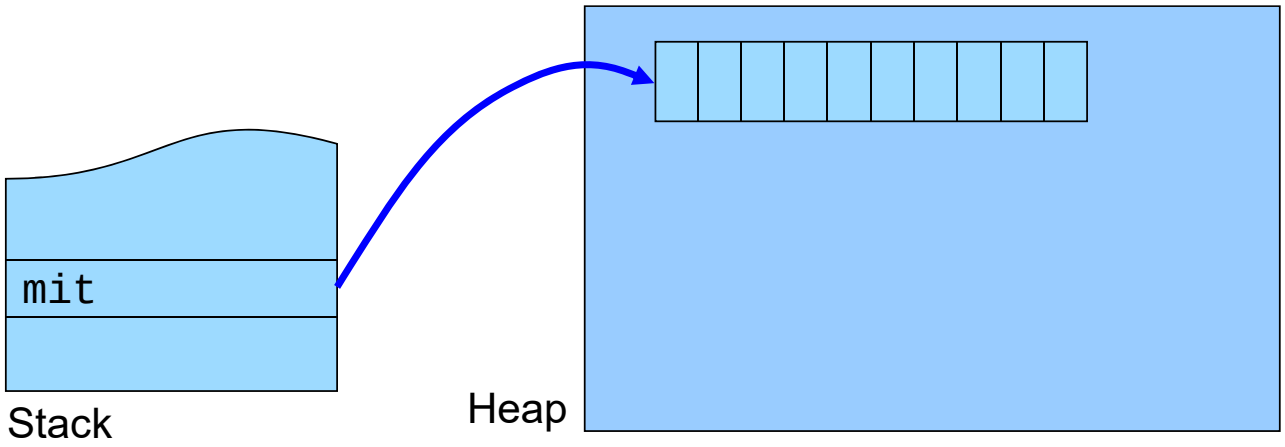
Assoziation mit höherer Multiplizität: Verwendung komplexer Datentyp (z.B. Feld) zur Speicherung mehrerer Mitarbeiter bei einer Abteilung

Felder von Objekten (I)

- Mit `Mitarbeiter[] mit` wird ein Feld von Objekten angelegt
- Hierbei sind zwei Aufrufe des new-Operators notwendig:

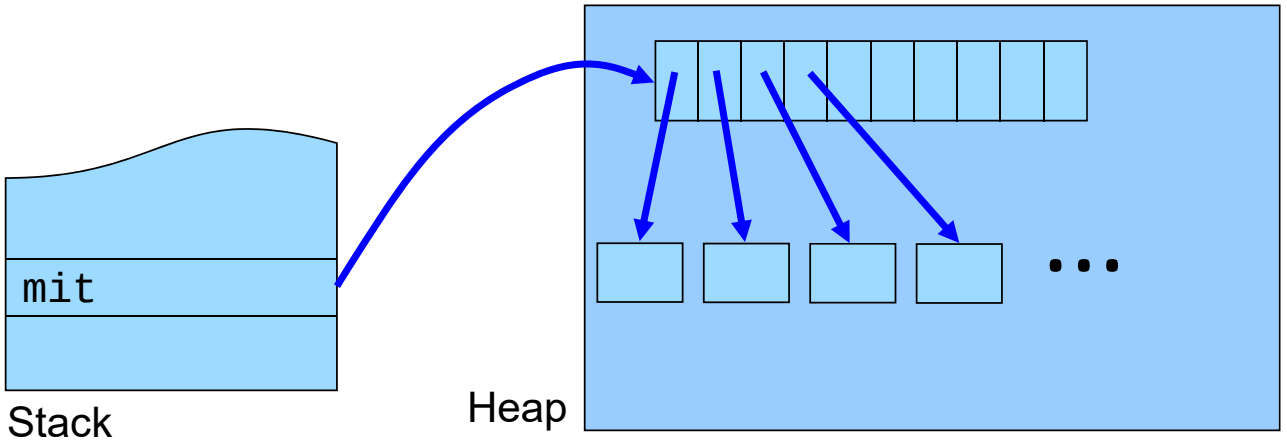
```
(1) Mitarbeiter[] mit = new Mitarbeiter[10];
(2) ...
(3) for(int i=0; i<=9; i++){
(4)   mit[i] = new Mitarbeiter();
(5) }
```

- Bei erstem new-Aufruf (Zeile(1)): Reservierung des Speicherplatz für Feld auf Heap



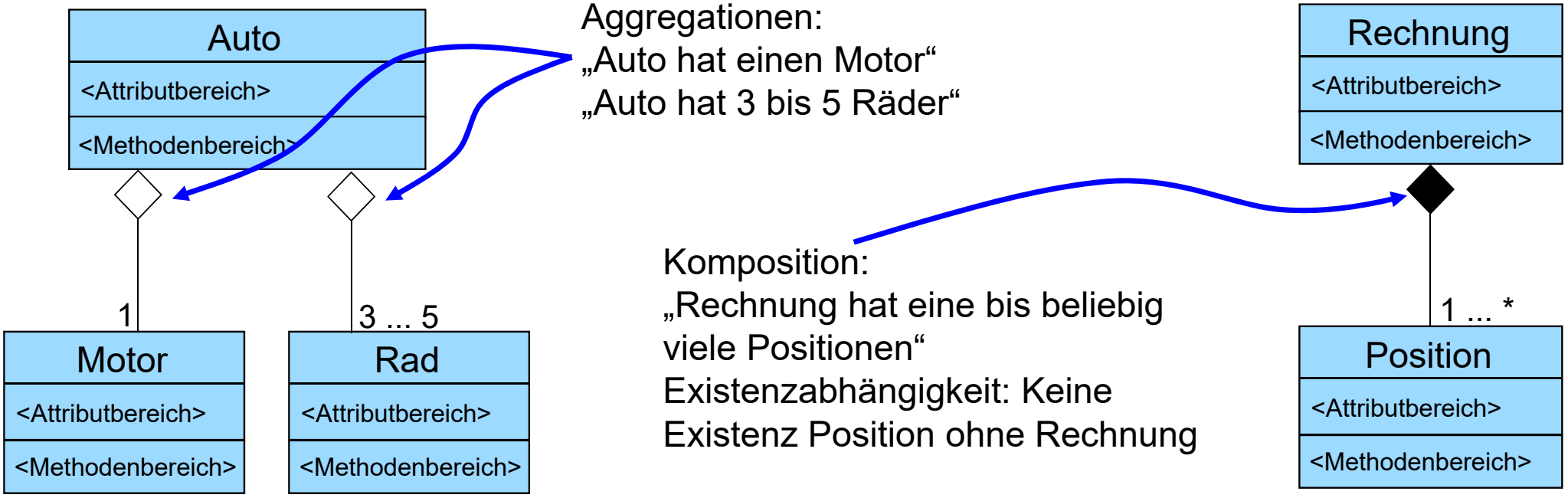
Felder von Objekten (II)

- Aufrufen new-Operator in Schleife: Reservierung Speicherplatz für Mitarbeiter-Objekte



Aggregation und Komposition

- Aggregation: Beziehung, die etwas Zusammenfassendes darstellt („hat ein“)
- Komposition: Zusammenfassende Beziehung mit Existenzabhängigkeit
- UML-Notation:

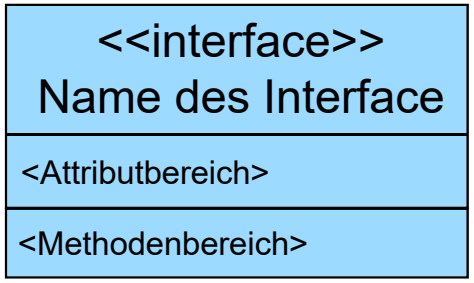


Übersicht

- Operatoren `this/super/instanceof`
- Parameterübergabe
- Beziehungen zwischen Klassen
- Schnittstellen
- Polymorphie
- Kopieren von Objekten
- Klassenbibliotheken

Idee/UML

- Bisher: Abstrakte Klassen, Methoden konnten implementiert sein oder nicht (abstrakte Methoden)
- Idee konsequent fortsetzen: Alle Methoden ohne konkrete Definition (vollständig abstrakte Klasse)
- Diese werden als Schnittstellen (Interfaces) bezeichnet
- UML-Notation:



Java-Notation

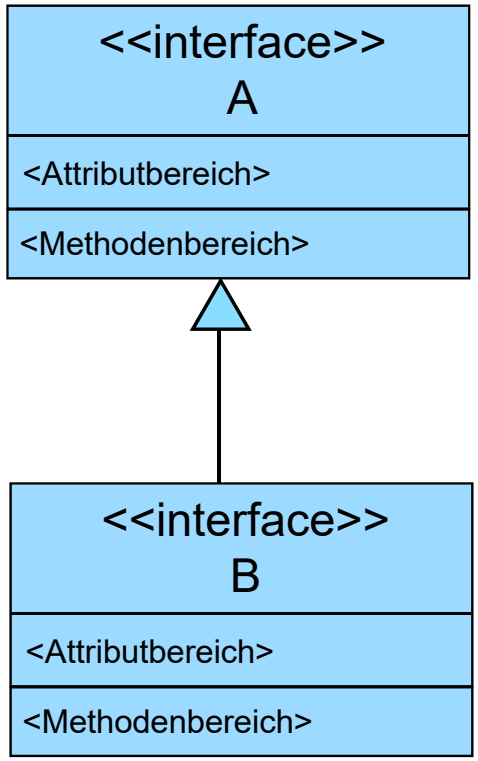
- Java-Syntax: `[public] interface NameDesInterface{ ... }`
- Methodendeklaration in Schnittstellen:
  `[public abstract] <Rückgabetyp> nameDerMethode(<Par.liste>);`
- Wichtig (Unterschied zu abstrakten Methoden in Klassen):
  - Schlüsselwort `abstract` muss nicht angegeben werden, kann aber
  - Methode darf nicht `protected` oder `private` sein
  - Leerer Rumpf wird einfach weggelassen (kein „{}“)
  - Zeile muss mit Semikolon abgeschlossen werden

Attribute

- Attributdeklaration in Schnittstellen:
  `<Datentyp> nameDesAttribut = <wert>;`
- Anmerkungen:
  - Auch ohne Angabe von Modifiern wird jedes Attribut einer Schnittstelle automatisch als `public static final` deklariert
  - Da nur diese Kombination sinnvoll ist, sollte sie auch immer angegeben werden
  - Attribute definieren Konstanten, die allen Klassen zur Verfügung stehen, die die Schnittstelle implementieren
  - Finale Attribute müssen in der Schnittstelle, in der sie deklariert werden, auch initialisiert werden, d.h. z. B. `public static final int i=0;`

Schnittstellen erben Schnittstellen (I)

- Vererbung von Schnittstellen an andere Schnittstellen möglich (analog zu Klassen)
- UML-Notation:



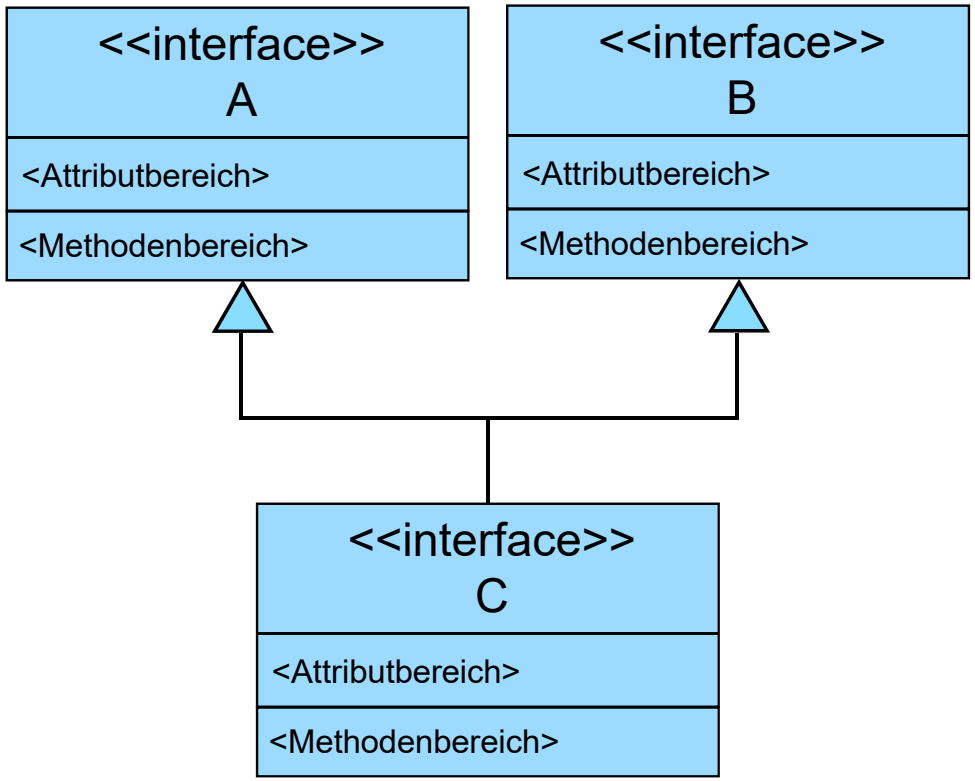
- Java-Notation:

```
(1) public interface A{
(2)   public static final int i = 0;
(3)   public abstract void f();
(4) }
```

```
(1) public interface B extends A{
(2)   public static final int j = 0;
(3)   public void g();
(4) }
```

Schnittstellen erben Schnittstellen (I)

- Mehrfachvererbung von Schnittstellen möglich (Unterschied zu Klassen)
- UML-Notation:



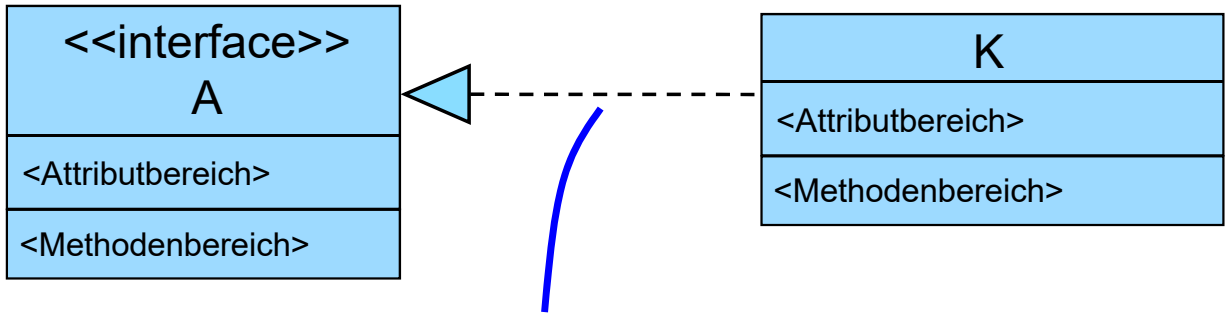
- Java-Notation:

Interface kann von mehreren Interfaces erben

```
(1) public interface C extends A,B{
(2)   public static final int j = 0;
(3)   public void g();
(4) }
```

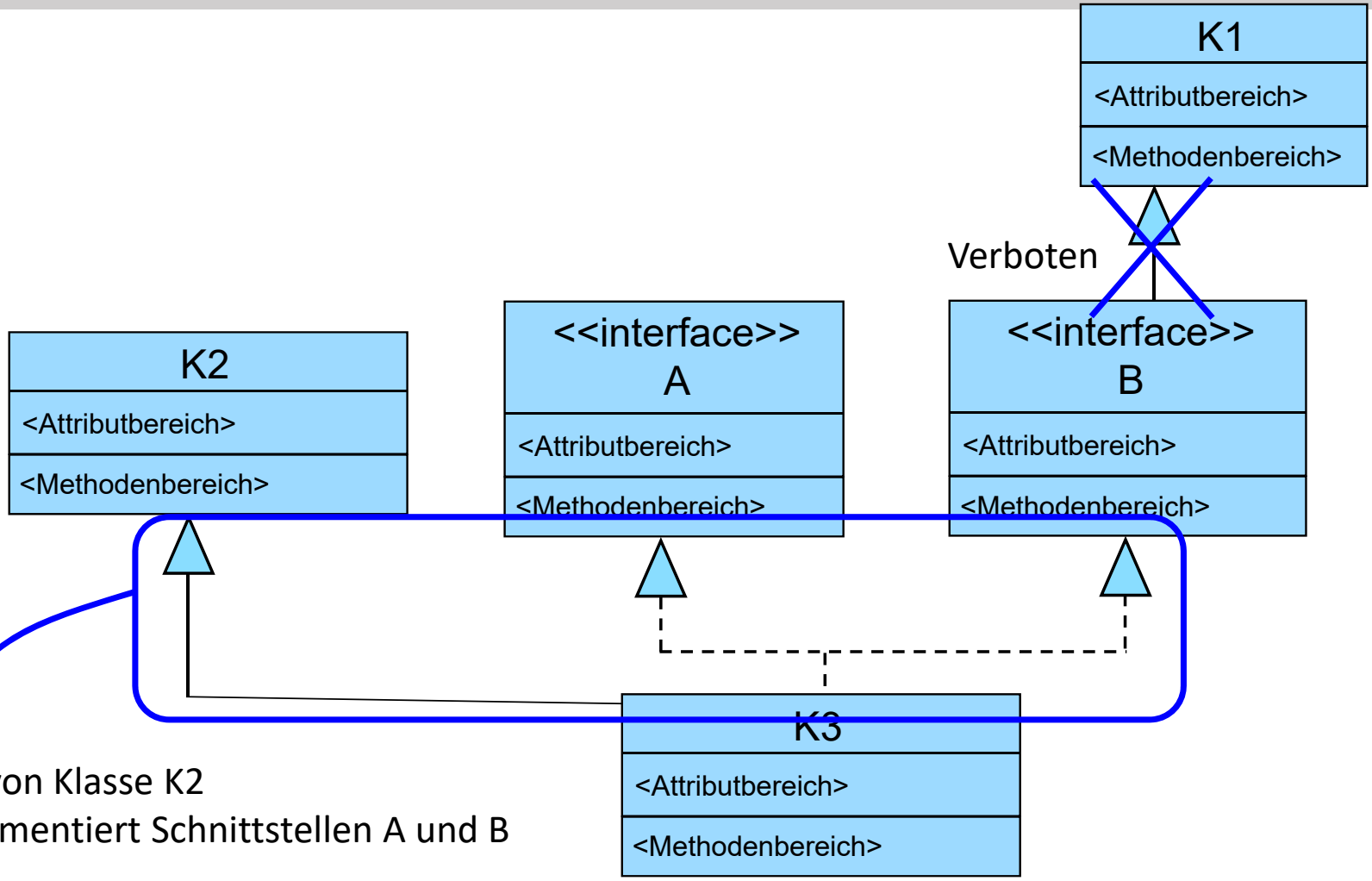
Schnittstellen und Klassen (I)

- Schnittstellen können nicht von Klassen erben
- Klassen können von einer oder mehreren Schnittstellen „erben“
- Wird als Implementieren bezeichnet
- Methoden müssen in Klassen implementiert werden
- UML-Notation:



- Klasse K implementiert Interface A
- Implementierung aller Methoden aus A in K

Schnittstellen und Klassen (II): UML Beispiel



- Klasse K3 erbt von Klasse K2
- Klasse K3 implementiert Schnittstellen A und B

Schnittstellen und Klassen (III): Java

- Auch in Java: Klasse kann beliebige Anzahl Schnittstellen implementieren
- Beispiel:

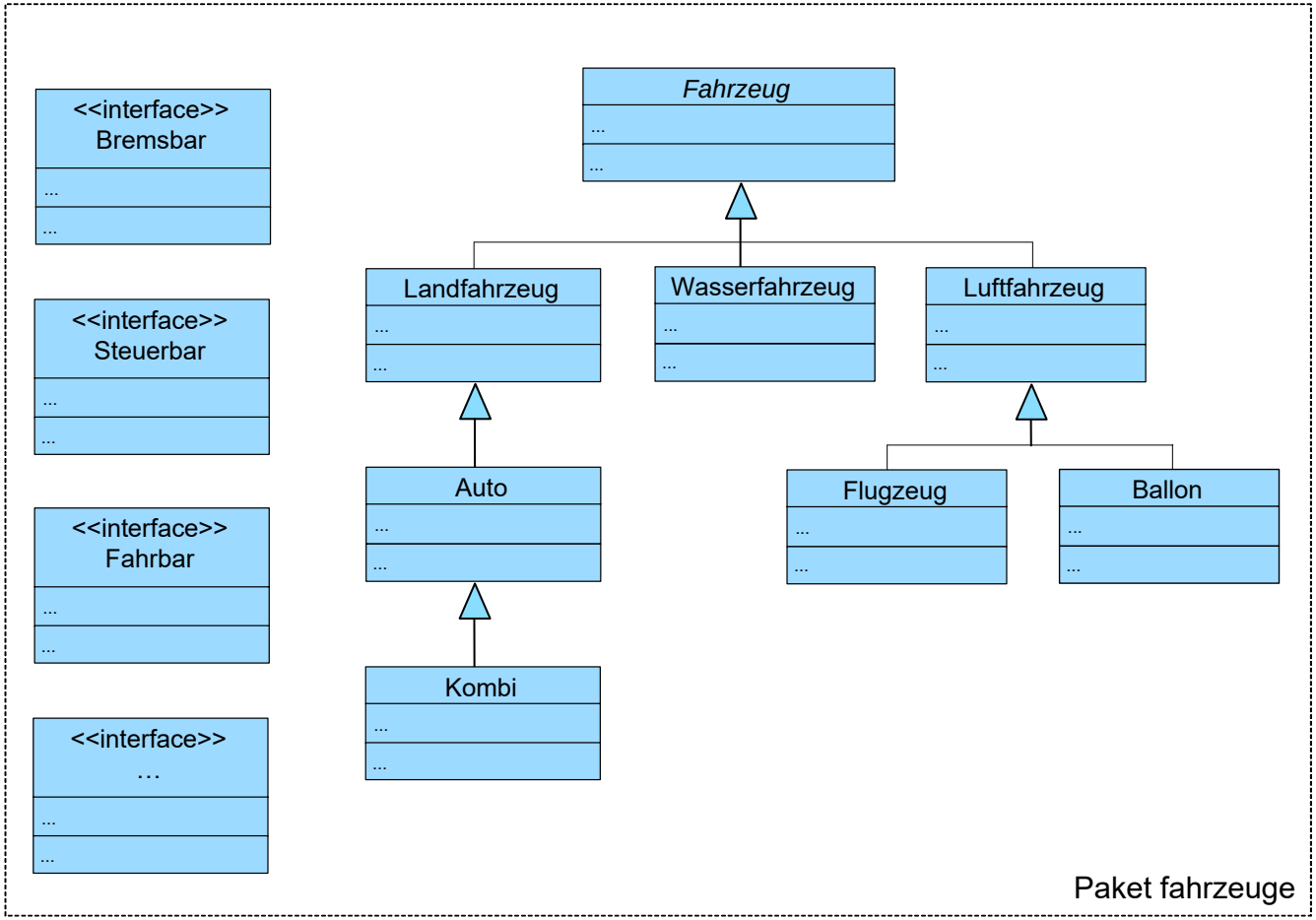
```
(1) public class K3 extends K2 implements A,B {
(2)    ...
(3) }
```

Hier darf nur eine Klasse stehen

Hier können beliebig viele Schnittstellen stehen

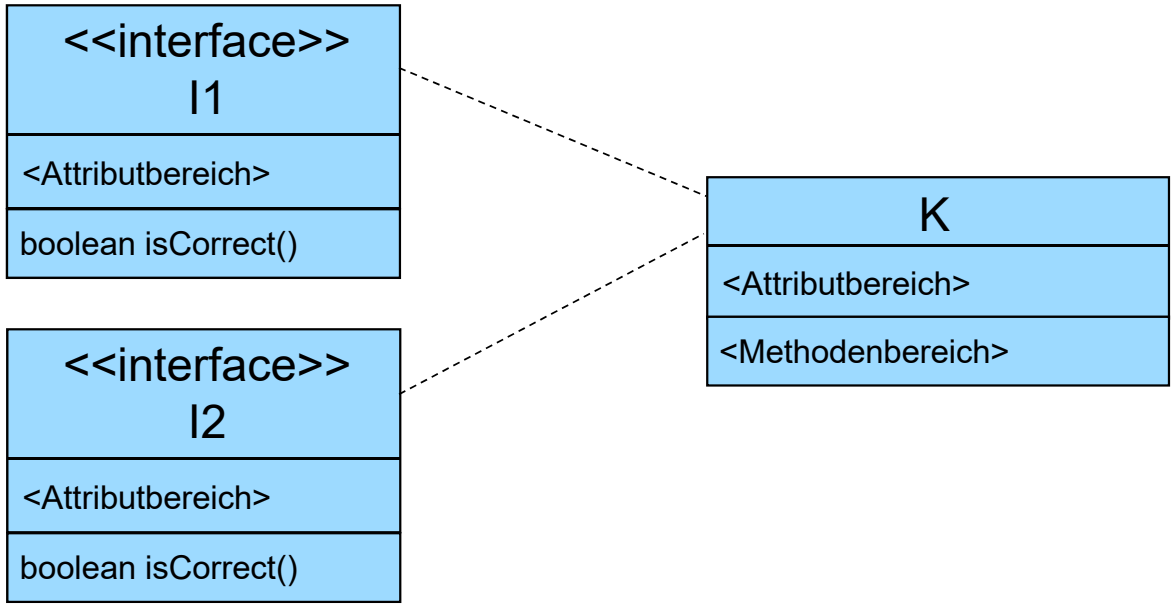
- Anmerkungen:
  - Bei Implementierung einer Schnittstelle müssen Methoden als `public` deklariert werden
  - Klasse implmentiert nicht alle Methoden einer Schnittstelle ⇒ Deklaration Klasse `abstract`
- Gründe für Einführung von Interfaces in Java:
  - Interfaces orthogonal zur Klassenhierarchie, d.h. jede Klasse kann sie implementieren, unabhängig von Position in Klassenhierarchie

Bsp. Klassenhierarchie mit Schnittstellen



Mehrfachimplementierung (I)

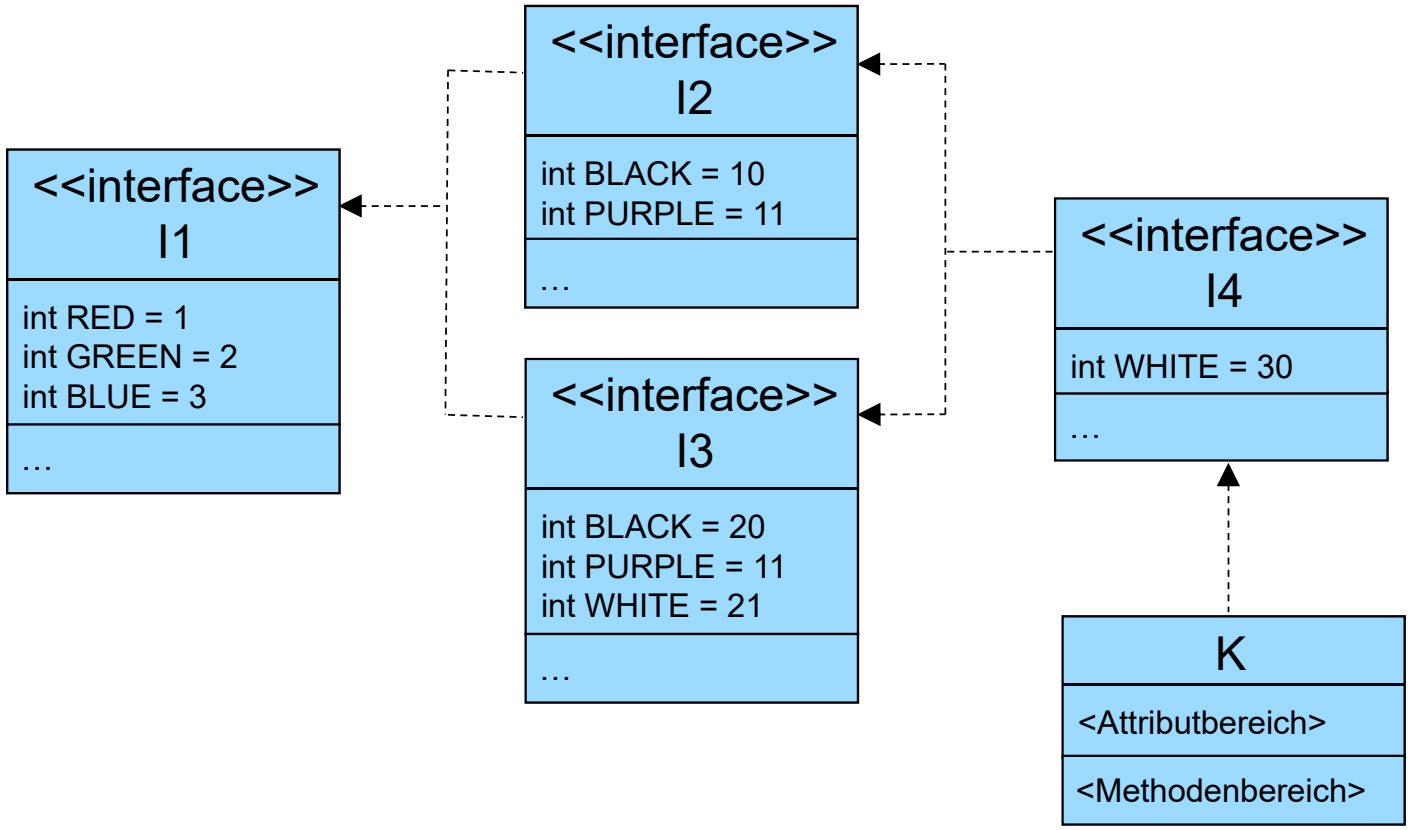
- Beispiel:



- Beide Interfaces schreiben Klasse K Implementierung der Methode `isCorrect` vor, was kein Problem darstellt
- Bei Mehrfachvererbung wäre dies anders: Superklassen könnten unterschiedliche Implementierung vorgeben

Mehrfachimplementierung (II)

- Beispiel:



- Wie kann Klasse K auf welche Konstanten zugreifen?

Mehrfachimplementierung (III): Regeln

- Schnittstellen vererben über mehrere Ebenen (z.B. RED wird von `I1` an `I2` an `I4` an K vererbt)
- Schnittstellen können von mehreren Schnittstellen gleiche Konstante erben (z.B. erbt `I4` die Konstante RED von `I2` und `I3`)
- Konstanten dürfen überschrieben werden (z.B. `I4` überschreibt den Wert von `WHITE`)
- Schnittstellen können von mehreren Schnittstellen Konstanten erben (z.B. erbt `I4` die Konstanten BLACK und PURPLE aus `I2` und `I3`):
  - Zugriff in diesem Fall über voll qualifizierten Namen: `I2.BLACK` bzw. `I3.BLACK`
  - Auch wenn beide Konstanten gleichen Wert haben (hier PURPLE) Zugriff nur über `I2.PURPLE` bzw. `I3.PURPLE` möglich (Grund: Sicherheit bzgl. späterer Änderungen)

Schnittstelle vs. abstrakte Klasse (I)

	Schnittstelle	Abstrakte Klasse
Idee (Gemeinsamkeit)	Unterklassen bzw. implementierenden Klassen Methoden vorschreiben, die diese implementieren müssen	
Idee (Unterschied)	Angabe ausschließlich von Konstanten und Methoden ohne jegliche Implementierung	Angabe von Attributen, Konstanten und Methoden; dabei kann ein gewisses Verhalten schon implementiert sein
Mehrfachvererbung/-implementierung	Möglich	Nicht möglich
Schlüsselwort `abstract`	Möglich, kann aber	Angabe notwendig
Deklaration von Attributen und Methoden	Nur als `public static final` gekennzeichnete Konstanten und als `public abstract` gekennzeichnete Methoden	Erlaubt, was in Klassen erlaubt ist

Schnittstelle vs. abstrakte Klasse (II)

	Schnittstelle	Abstrakte Klasse
Idee (Gemeinsamkeit)	Unterklassen bzw. implementierenden Klassen Methoden vorschreiben, die diese implementieren müssen	
Methodenrumpf	Muss leer sein	Kann Implementierung enthalten
Darstellung leerer Rumpf	Wird weggelassen	{} muss angegeben werden
Verwendung auch in Unterklasse	Ja	Ja
Organisation	In Paketen, beide haben Zugriffsmodifier	

Schnittstelle oder abstrakte Klasse (I)

- Grundsätzlich: Sind alle Methoden ohne Implementierung und sollen nur Konstanten weitergegeben werden, so ist Schnittstelle zu wählen, sonst abstrakte Klasse
- Entscheidung aber nicht immer einfach
- Bsp.: Timer

```
(1) abstract class Timer{
(2)   abstract long getTimeInMillis(){};
(3) }
```

```
(1) interface Timer{
(2)   long getTimeInMillis();
(3) }
```

- Beide Lösungen scheinbar gleichwertig
- Bei späterer Erweiterung um Methode `getTimeInSeconds()`: Erweiterung abstrakter Klasse einfacher

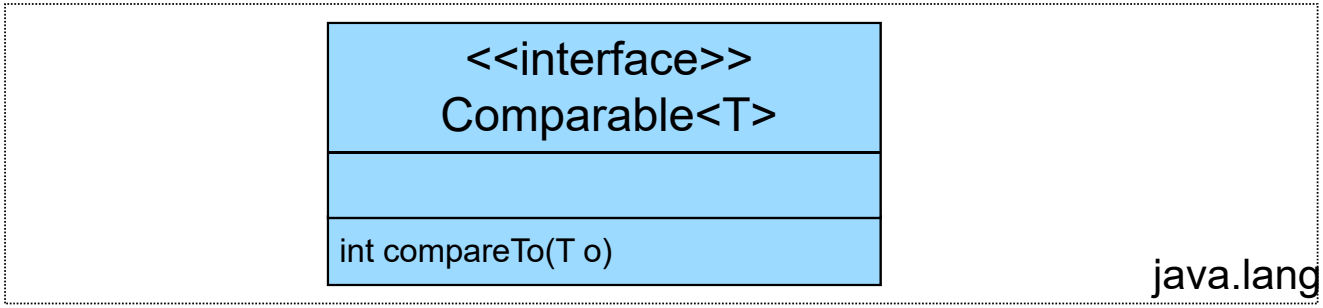
Schnittstelle oder abstrakte Klasse (II)

```
(1) public abstract class Timer{
(2)   abstract long getTimeInMillis(){};
(3)
(4)   long getTimeInSeconds(){
(5)     return getTimeInMillis()*1000;
(6) }
```

```
(1) public interface Timer{
(2)   long getTimeInMillis();
(3) }
```

- In Interface wäre dies nicht möglich
- Anm.: In Java-Klassenbibliothek gibt es abstrakte Klassen und dazu korrespondierende Interfaces

Interface `Comparable<T>` (I)



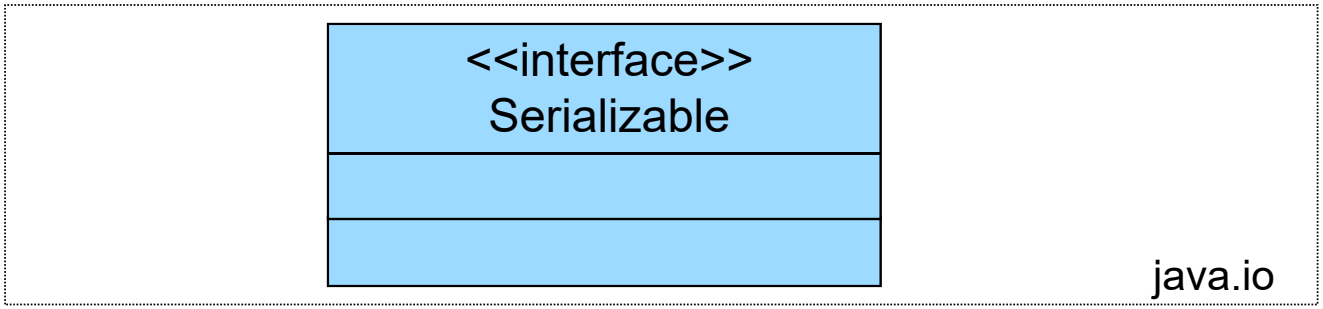
- Definiert totale Ordnung auf Objekten implementierender Klasse
- Wird als natürliche Ordnung der Klasse bezeichnet, `compareTo` als natürliche Vergleichsmethode
- Methode `compareTo` liefert negativen Wert, 0 bzw. positiven Wert, wenn Objekt kleiner, gleich oder größer dem Vergleichsobjekt ist
- Es gilt: Sei sgn die mathematische Signum(Vorzeichen)-Funktion, d.h.

$$\text{sgn}(expr) = \begin{cases} -1, \textit{falls } expr < 0 \\ 0, \textit{falls } expr = 0 \\ +1, \textit{falls } expr > 0 \end{cases}$$

Interface `Comparable<T>` (II)

- Entwickler*in muss sicherstellen:
  - Symmetrie: Für alle x, y muss sgn(x.compareTo(y)) == sgn(y.compareTo(x)) gelten
  - Transitivität: Für alle x, y, z muss gelten: $((x.compareTo(y)) > 0 \;\&\&\; (y.compareTo(z)) > 0) \Rightarrow x.compareTo(z) > 0$
  - Gleichheit: $x.compareTo(y) == 0 \Rightarrow \text{sgn}(x.compareTo(z)) == \text{sgn}(y.compareTo(z))$ für alle z
- Beziehung zur Methode `equals`:
  - Def.: Konsistenz mit Methode `equals` g.d.w. für alle `e1` und `e2` gilt `(e1.compareTo((Object) e2) == 0)` hat Wert wie `e1.equals((Object) e2)`
  - Bei Nicht-Erfüllung Kommantar hinzuzufügen: „This class has a natural ordering that is inconsistent with equals.“

Interface `Serializable`



- Ermöglicht Schreiben in Ausgabestrom bzw. Lesen aus Eingabestrom von Objekten (Siehe Abschnitt „Ein-/Ausgabe“)
- Interface ohne Konstanten und Methoden
- Bezeichnung: Markierungsschnittstelle (Marker interface)
- In implementierender Klasse:
  - Konstante `serialVersionUID` vom Typ `long` definieren
  - Dient Überprüfung beim Ein-/Auslesen

Übersicht

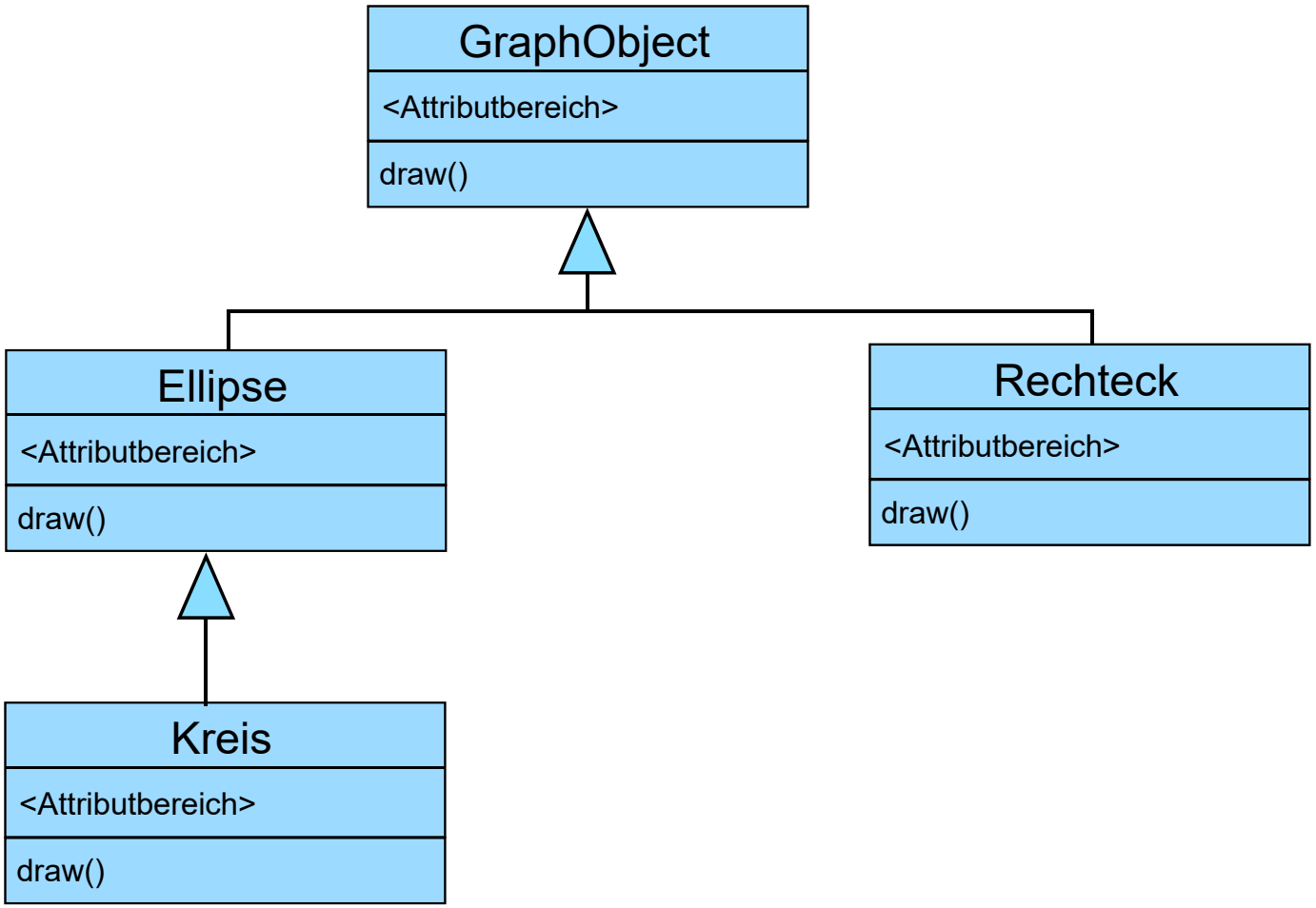
- Operatoren `this/super/instanceof`
- Parameterübergabe
- Beziehungen zwischen Klassen
- Schnittstellen
- Polymorphie
- Kopieren von Objekten
- Klassenbibliotheken

Idee

- Zwei Formen der Polymorphie:
  - Statische Polymorphie
  - Dynamische Polymorphie
- Statische Polymorphie: Überladen (siehe Abschnitt Objektorientierung (I))
- Dynamische Polymorphie:
  - Idee: Alle Objekte einer Superklasse (z.B. Mitarbeiter) in gleicher Art und Weise behandeln
  - Umsetzung:
    - Objektvariable einer Superklasse kann auf Objekte der Subklassen verweisen
    - Methodenaufruf: Verwendung der Methode der Subklasse bzw. die spezifischte Implementierung (überschriebene Methode)
    - Ermittlung entsprechender Methode erst zur Laufzeit (Dynamisches Binden)

Beispiel (I)

- Gegeben:



Beispiel (II)

- Realisierung `draw()`-Methode durch Ausgabe Objekttyp, z.B.

```
(1) public void draw(){
(2)   System.out.println("Ellipse");
(3) }
```

- „Ganz normaler“ Aufruf `draw()`-Methoden im Hauptprogramm, z.B.

```
( 1) // Anlegen der Objekte
( 2) GraphObject meinGraphObject = new GraphObject();
( 3) Ellipse meineEllipse = new Ellipse();
( 4) Rechteck meinRechteck = new Rechteck();
( 5) Kreis meinKreis = new Kreis();
( 6) // Ausgeben der Objekte
( 7) meinGraphObject.draw();
( 8) meineEllipse.draw();
( 9) meinRechteck.draw();
(10) meinKreis.draw();
```

Beispiel (III)

- Führt zur Ausgabe

```
GraphObject
Ellipse
Rechteck
Kreis
```

Beispiel (IV)

- Verwendung von Polymorphie: Anlegen Objekte, Speicherung in Feld vom Typ `GraphicObjects`, einheitlicher Methodenaufruf :

```
( 1) // Anlegen der Objekte
( 2) GraphObject meinGraphObject = new GraphObject();
( 3) Ellipse meineEllipse = new Ellipse();
( 4) Rechteck meinRechteck = new Rechteck();
( 5) Kreis meinKreis = new Kreis();
( 6)
( 7) // Initialisieren eines Feldes von Grafik-Objekten
( 8) GraphObject[] meinGraphObjectFeld;
( 9) meinGraphObjectFeld = new GraphObject[4];
(10) ...
```

Beispiel (V)

```
...
(11) // Zuweisen der Objekte
(12) meinGraphObjectFeld[0] = meinGraphObject;
(13) meinGraphObjectFeld[1] = meineEllipse;
(14) meinGraphObjectFeld[2] = meinRechteck;
(15) meinGraphObjectFeld[3] = meinKreis;
(16) // Aufruf der draw-Methode
(17) for (int i=0; i<=3; i++){
(18)   meinGraphObjectFeld[i].draw();
(19) }
```

führt ebenfalls zur Ausgabe

```
GraphObject
Ellipse
Rechteck
Kreis
```

Beispiel (VI)

- Anwendung Polymorphie auf Konstruktoren möglich, z.B.:

```
( 1) // Feld von Grafikobjekten
( 2) GraphObject[] meinGraphObjectFeld;
( 3) meinGraphObjectFeld = new GraphObject[4];
( 4) // Polymorphe Kontruktorenaufrufe
( 5) meinGraphObjectFeld[0] = new GraphObject();
( 6) meinGraphObjectFeld[1] = new Ellipse();
( 7) meinGraphObjectFeld[2] = new Rechteck();
( 8) meinGraphObjectFeld[3] = new Kreis();
( 9) // Ausgabe
(10) for (int i=0; i<=3; i++){
(11)   meinGraphObjectFeld[i].draw();
(12) }
```

führt zur bekannten Ausgabe

Übersicht

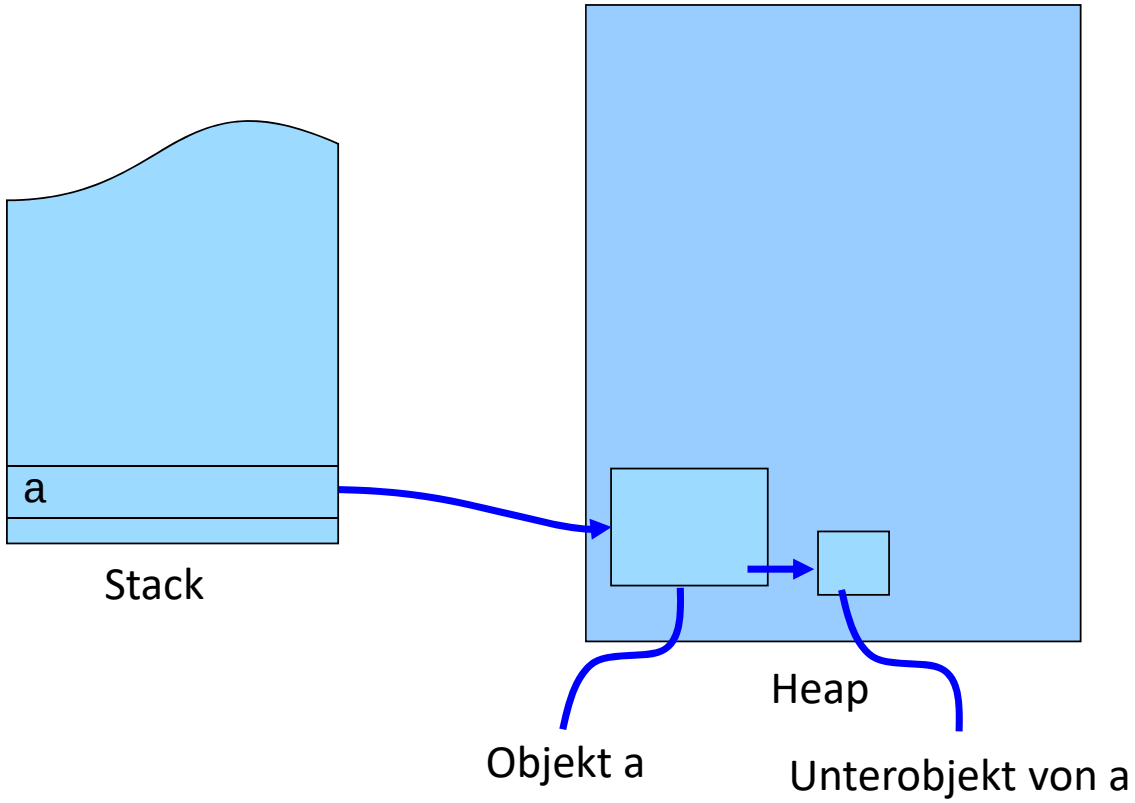
- Operatoren `this/super/instanceof`
- Parameterübergabe
- Beziehungen zwischen Klassen
- Schnittstellen
- Polymorphie
- Kopieren von Objekten
- Klassenbibliotheken

Referenz und Kopie

- W.D.H.:
  - Anwendung Zuweisungsoperator auf Objekte: Setzen Referenz, keine Kopie
- Anforderung: Erstellung „richtiger“ Kopie
- Zwei Möglichkeiten:
  - Eigene Methode schreiben, von Hand neues Objekt anlegen, alle Attribute kopieren und Referenz auf dieses Objekt zurückgeben
  - Aus Klasse `Object` geerbte Methode `Object clone()` benutzen
- Beim Kopieren in objektorientierten Sprachen unterscheidet man:
  - Flache Kopie: Unterobjekte werden nicht mitkopiert
  - Tiefe Kopie: Unterobjekte werden mitkopiert
- Anm.: Methode `clone()` erzeugt eine flache Kopie

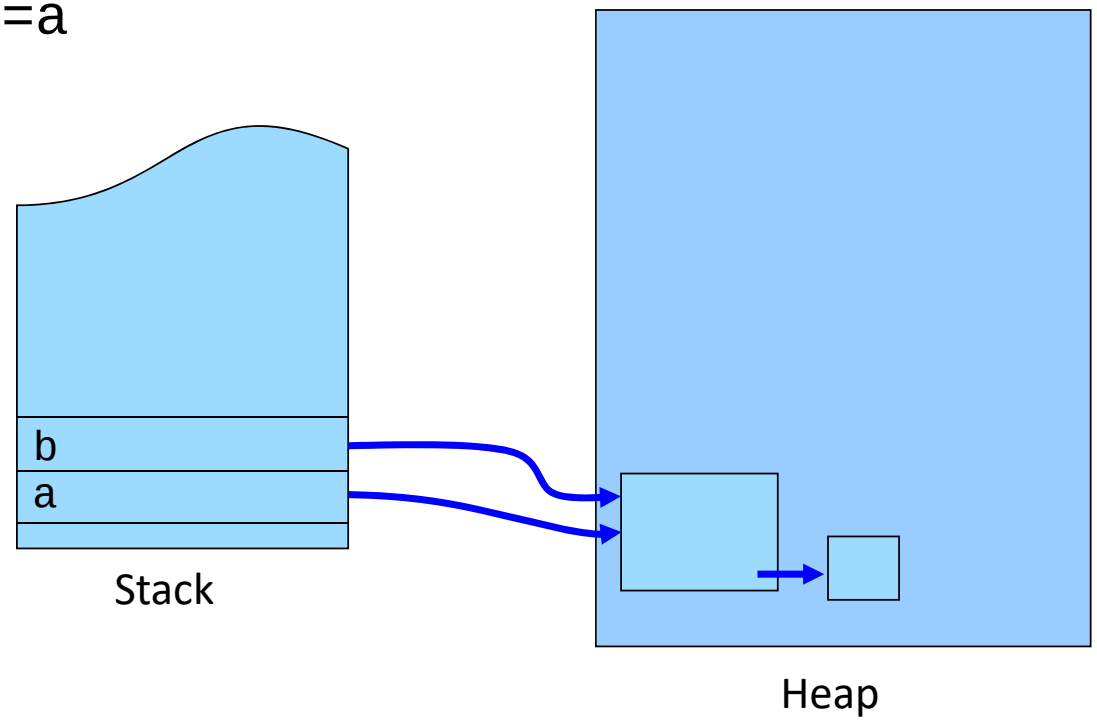
Flache und tiefe Kopie (I)

- Beispiel: Objekt a (mit Unterobjekt) vorhanden



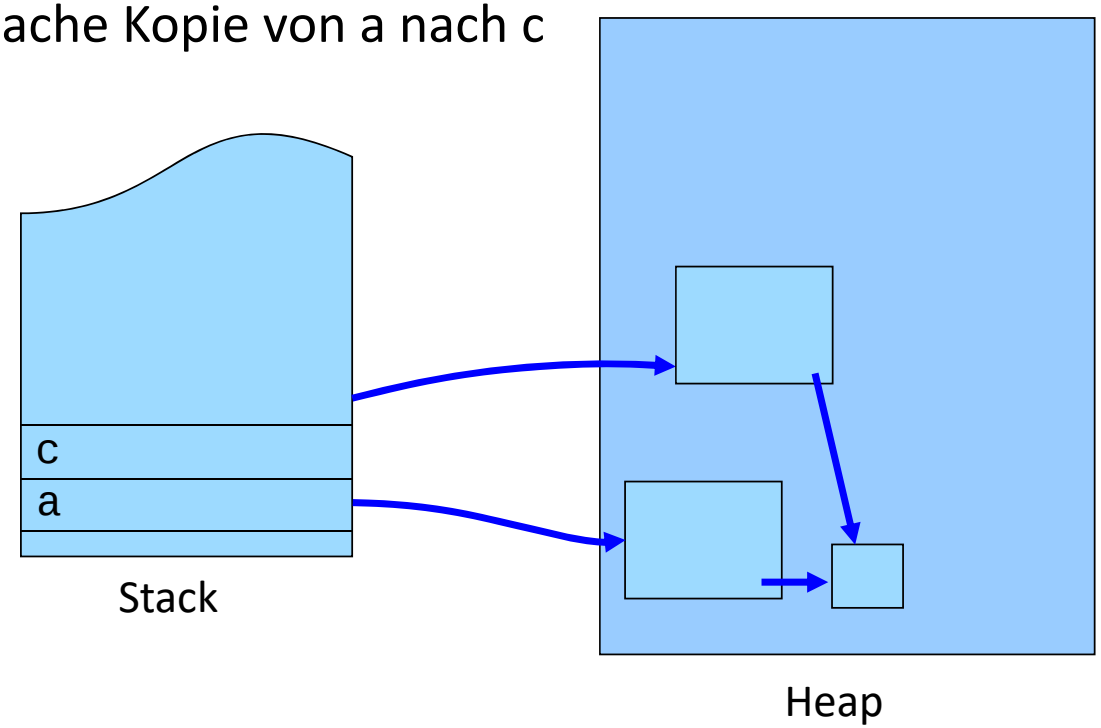
Flache und tiefe Kopie (II)

- Beispiel: Objekt a (mit Unterobjekt) vorhanden
- Ausführen b=a



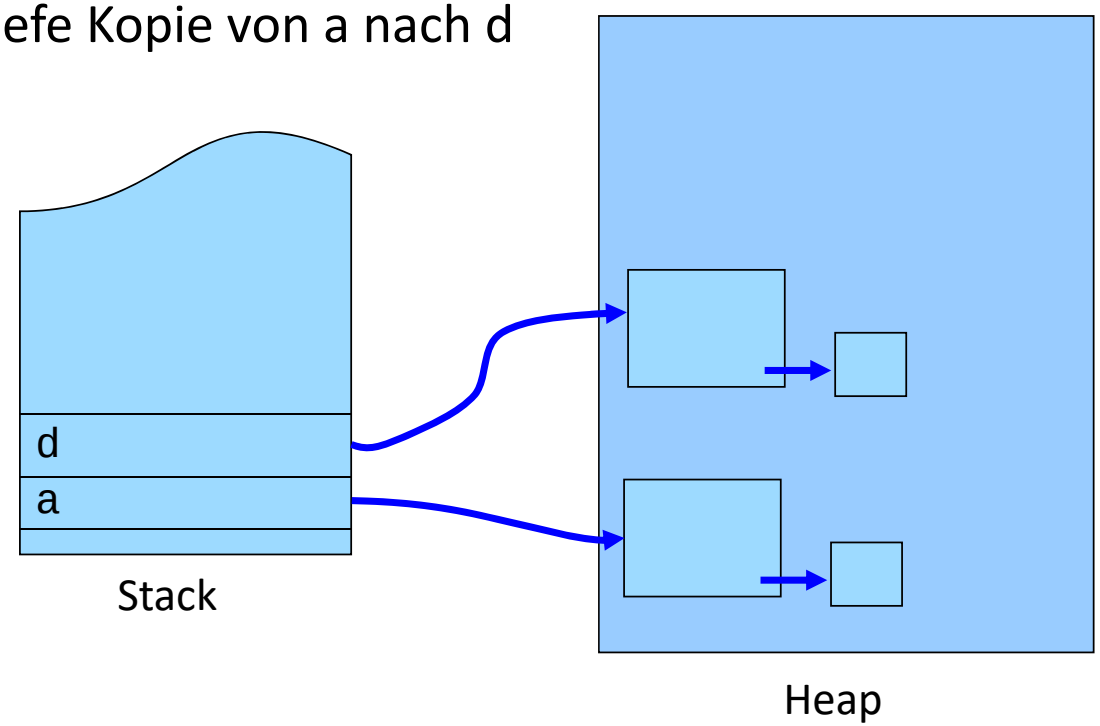
Flache und tiefe Kopie (III)

- Beispiel: Objekt a (mit Unterobjekt) vorhanden
- Ausführen flache Kopie von a nach c



Flache und tiefe Kopie (IV)

- Beispiel: Objekt a (mit Unterobjekt) vorhanden
- Ausführen tiefe Kopie von a nach d



Methode clone (I)

- Beispiel: Manager mit Objekttyp Fahrzeug

```
( 1) public class Manager{
( 2)   ...
( 3)   private String name;
( 4)   private int persNr;
( 5)   private String wohnort;
( 6)   private Fahrzeug fahrzeug;
( 7)   ...
( 8)   public setKennzeichen(String s){
( 9)     this.fahrzeug.setKennzeichen(s);
(10)   }
(11) }
```

```
( 1) public class Fahrzeug{
( 2)   ...
( 3)   String fahrzeug;
( 4)   ...
( 5) }
```

Methode clone (II)

- Vorgehen:
  - Methode `clone()` überschreiben
  - Methode der Superklasse muss aufgerufen werden
  - Schnittstelle `Cloneable` muss implementiert werden
- Erweiterung Klasse `Manager`:

```
( 1) public class Manager implements Cloneable{
( 2)   ...
( 3)   public Object clone(){
( 4)     try{
( 5)       return super.clone();
( 6)     }
( 7)     catch (CloneNotSupportedException e){
( 8)       „Ausnahmebehandlung“
( 9)     }
(10)   }
(11) }
```

Methode clone (III)

```
( 1) public class Anwendung{
( 2)   ...
( 3)   public static void main(String[] args){
( 4)     // Anlegen eines Managers
( 5)     Manager a =
( 6)           new Manager("Meier",4711,"Stuttgart",12);
( 7)     a.setKennzeichen("S-ZZ 99");
( 8)     // Kopieren mittels der Methode clone
( 9)     Manager  b = new Manager();
(10)     b = (Manager) a.clone();
(11)     // Ausgeben der beiden Objekte
(12)     a.drucken();
(13)     b.drucken();
(14)     // Ändern des Names und Kennzeichens bei Manager a
(15)     a.setName("Müller");
(16)     a.setKennzeichen("S-XX 11");
(17)     // Erneute Ausgabe der beiden Objekte
(18)     a.drucken();
(19)     b.drucken();
(20)   }
(21) }
```

Methode clone (IV)

- Ausgabe:

```
Personalnummer: 4711
Name:           Meier
Ort:            Stuttgart
Kennzeichen :   S-ZZ 99

Personalnummer: 4711
Name:           Meier
Ort:            Stuttgart
Kennzeichen :   S-ZZ 99

Personalnummer: 4711
Name:           Müller
Ort:            Stuttgart
Kennzeichen :   S-XX 11

Personalnummer: 4711
Name:           Meier
Ort:            Stuttgart
Kennzeichen :   S-XX 11
```

Übersicht

- Operatoren `this/super/instanceof`
- Parameterübergabe
- Beziehungen zwischen Klassen
- Schnittstellen
- Polymorphie
- Kopieren von Objekten
- Klassenbibliotheken

Klassenbibliotheken (I)

- Kern jeder Programmiersprache: elementaren Anweisungen (z.B. Kontrollstrukturen, Möglichkeit der Klassenbildung in objektorientierten Sprachen usw.)
- Ergänzung um Klassenbibliotheken: Sammlungen von häufig benötigten Funktionalitäten (z.B. mathematische Funktionen, Dateibehandlung, Grafik, Gestaltung GUI, etc.)
- Spezifikation durch API (Application Programming Interface):
  - Beschreibung aller öffentlichen Klassen, Methoden und Schnittstellen der Bibliothek
  - Private Elemente werden hier nicht dokumentiert

Klassenbibliotheken (II)

- Verschiedene Ansätze:
  - Funktionen integraler Bestandteil der Sprache
  - Sprachkern mit elementaren Funktionen, Ergänzung um Funktionen von (verschiedenen) Herstellern/Entwicklern
  - Java: Mischung, d.h. Sprachkern mit elementaren Funktionen, zusätzliche Funktionen genormt (Java-Klassenbibliothek)
- Einbindung:
  - Statisch: Bibliotheken sind Bestandteil des Programms
  - Dynamisch: Separates Speichern Bibliotheken, Nutzung von mehreren Programmen

Java-Klassenbibliothek

- Sehr umfangreich, einige Tausend Klassen und Schnittstellen
- Zusammengefasst in Paketen
- Oberhalb von Paketen: Module als eine weitere Orgaeinheit
- Bibilithek kann nicht „auf Vorrat“ gelernt werden
- Dokumentation: Overview (Java SE 21 & JDK 21) (oracle.com)

Zusammenfassung (I)

- Operatoren `this/super/instanceof`:
  - `this`: Zeigt auf aktuelles Objekt
  - `super`: Zeigt auf Superklasse
  - `instanceof`: Liefert Klassenzugehörigkeit

- Parameterübergabe:
  - Call by Value
  - Call by Reference

- Beziehungen zwischen Klassen:
  - Assoziationen
  - Aggregationen
  - Kompositionen

Zusammenfassung (II)

- Schnittstellen:
  - Vollständig abstrakte Klassen
  - Orthogonalität zur Klassenhierarchie
  - Erben und Implementieren
  - Mehrfachimplementierung
  - Interface vs. abstrakte Klasse
  - Interface `Comparable<T>`
  - Interface `Serializable`

- Polymorphie:
  - Statische Polymorphie
  - Dynamische Polymorphie

Zusammenfassung (III)

- Kopieren von Objekten:
  - Kopie und Referenz
  - Tiefe und flache Kopie
  - Methode `clone()`

- Klassenbibliotheken:
  - Einleitung
  - Java-Klassenbibliothek

Aufgaben (I)

- **Aufgabe 1**

  Kopieren Sie Ihre Lösung aus Aufgabe 1 von Übungsblatt 6 (Paket personal) und führen Sie die folgenden Modifikationen durch:

  a) Vereinfachen Sie bei allen Unterklassen die `print()`-Methode, indem Sie die gleichnamige Methode aus der jeweiligen Oberklasse aufrufen!

  b) Modifizieren Sie die Konstruktoren mit den Attributen als Parameter dahingehend, dass die Attributbezeichnungen in den Parameterlisten den gleichen Namen tragen wie die Attribute der Objekte!

Aufgaben (II)

- **Aufgabe 2**

  In einer Klassenhierarchie „Fahrzeuge“ soll folgendes dargestellt werden: Es gibt eine abstrakte Superklasse Fahrzeug, die an Land-, Wasser- und Luftfahrzeuge vererbt. Außerdem gibt es Amphibienfahrzeuge, die die Eigenschaften von Land- und Wasserfahrzeugen vereinen.

  a) Erstellen Sie unter Nutzung des Interface-Konzepts ein UML-Klassendiagramm, in dem nur von Interfaces mehrfach geerbt werden darf.
  b) Realisieren Sie diese Schnittstellen und Klassen in Java!

  Anmerkung: Wichtig ist bei dieser Aufgabe das Prinzip des Vererbens von Klassen und Schnittstellen, die Anzahl der Attribute und Methoden können Sie möglichst gering halten!

Aufgaben (III)

- **Aufgabe 3**

  Erweitern Sie Ihre Klasse Matrix von Übungsblatt 5 um eine Methode zur Matrixmultiplikation, in der zwei Matrizen multipliziert werden, wenn sie kompatible Formate besitzen.

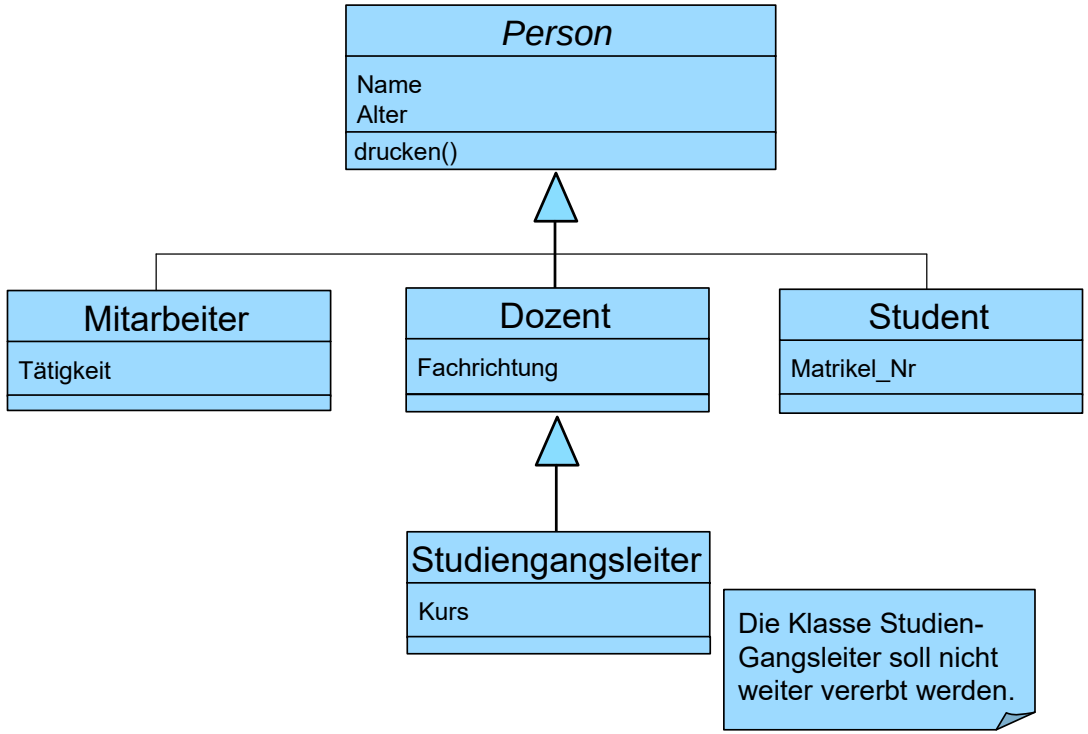
  Rückgabetyp der Methode soll `boolean` sein, wobei `false` zurückgegeben wird, wenn die Matrizen für die Multiplikation inkompatible Formate haben.

  Hinweis: Sollte die Matrixmultiplikation aus der Schule nicht bekannt sein, so schauen Sie bitte z.B. unter https://www.mathebibel.de/matrizenmultiplikation oder https://www.youtube.com/watch?v=W6sTa0dYrK4 oder https://www.youtube.com/watch?v=uIykRXQhQtE&t=12s nach.

Aufgaben (IV)

- **Aufgabe 4**

  Gegeben sei das folgende Klassendiagramm:



Aufgaben (V)

Setzen Sie dieses in Java um, indem Sie ein Paket `personenDH` erstellen. Beachten Sie dabei das Prinzip der Kapselung. Realisieren Sie eine Klasse Anwendung, die außerhalb des Paketes definiert ist und Objekte der verschiedenen Klassen anlegt und ausgibt. Nutzen Sie dabei das Prinzip der Polymorphie.