Vorlesung „Objektorientierte Softwareentwicklung“

Sommersemester 2008

Teil 2. Objektorientierte Konzepte

Objekte und Klassen

universitätbonn

Objekte und Klassen

Gekapselte, dynamisch erzeugbare „Module“→ Objekte

Schablonen zur Objekterzeugung → Klassen

Klassen- versus Instanzmitglieder

Zustand und Verhalten

Von Modulen zu Objekten (1)

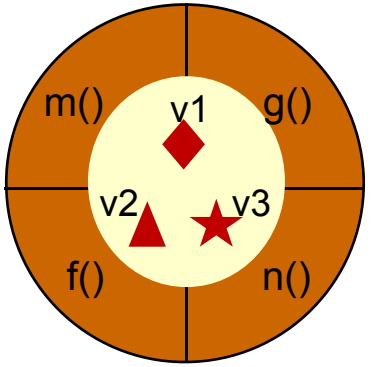
- Module sind statisch!
  - Man kann nur eine endliche Anzahl davon aufschreiben
- Die Welt ist dynamisch!
  - Man muss oft neue „Einheiten“ nach Bedarf erzeugen
  - Man weiß oft nicht in vorhinein wann und wie viele
  - Beispiel: Es gibt nicht nur ein Auto in der Welt!
- Idee: Alle Autos im gleichen Modul darstellen
  - Komplexität!? – Alle Exemplare, aller Varianten, aller Typen aller Hersteller in einem Modul?
  - Verständlichkeit und Wartbarkeit!? – Verquickung von verschiedenster Spezialfälle in einem Modul?
  - Schlechte Idee!!!

Von Modulen zu Objekten (2)

- Module sind statisch!
  - Man kann nur eine endliche Anzahl davon aufschreiben
- Die Welt ist dynamisch!
  - Man muss oft neue „Einheiten“ nach Bedarf erzeugen
  - Man weiß oft nicht in vorhinein wann und wie viele
  - Beispiel: Es gibt nicht nur ein Auto in der Welt!
- Idee: Modulbeschreibung und Speicherzuteilung trennen
  - Klasse bleibt ein statisches Modul
  - Sie beschreibt aber zusätzlich die gemeinsame Struktur einer beliebigen Menge „dynamisch erzeugbarer Module“ → Objekte
  - Daraus werden nach Bedarf „Module“ erzeugt die der Struktur entsprechen aber jeweils einen eigenen Speicherbereich haben

Objekte

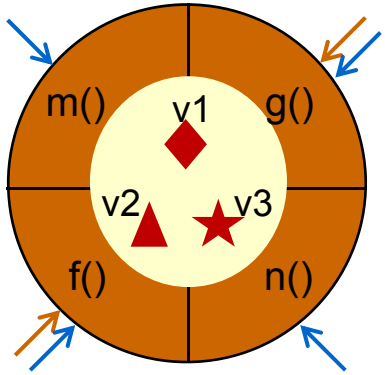
- Objekte
  - Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
    - Variablen → Zustand
    - Operationen → Verhalten



- Zustand eines Objektes
  - Werte der Variablen des Objektes zu einem gewissen Zeitpunkt → Zustand kann sich ändern
- Verhalten eines Objektes
  - Menge der Reaktionen auf Operationsaufrufe
  - Reaktionsmöglichkeiten: Eigene Zustandsübergänge und Aufruf von Operationen anderer Objekte

Objekte

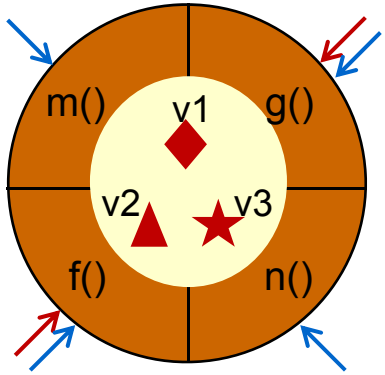
- Objekte
  - Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
    - Variablen → Zustand
    - Operationen → Verhalten



- Schnittstelle
  - Menge für einen bestimmten Benutzerkreis aufrufbaren Operationen
  - Verschiedene Arten von „Benutzern“ können evtl. unterschiedliche Schnittstellen angeboten bekommen (‚public‘, ‚package‘, …)

Objekte: Kapselung

- Objekte
  - Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
    - Variablen → Zustand
    - Operationen → Verhalten

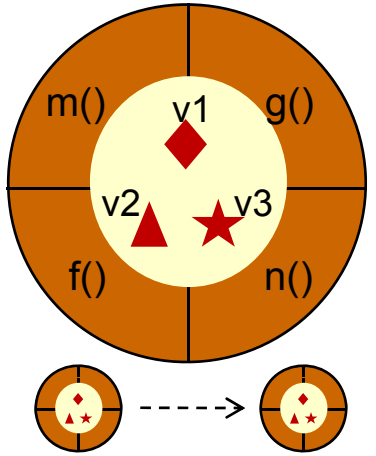


- Kapselung
  - Sprache stellt sicher, dass der Zustand eines Objektes nur über die in seiner Schnittstelle spezifizierten Operationen manipuliert wird
  - ➔ Bei gleichbleibendem Interface wirken sich Änderungen der lokalen Implementierung nicht auf andere Objekte aus

➔ Wartbarkeit und Zugriffs-Synchronisation!

Wie entstehend Objekte? - Spezifikation und Erzeugung

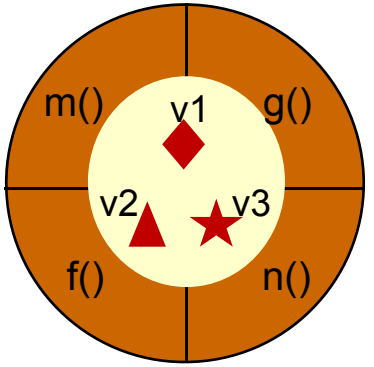
- Objekt-/Prototypbasierte Sprachen
  - Erzeugung durch „hinschreiben“:
    Die Variablen + Methoden können pro Objekt einzeln spezifiziert werden
  - Erzeugung durch kopieren („clonen“):
    Objekte werden durch Kopieren von bestehenden Objekten erzeugt und danach verändert



- Beispiele
  - Self – der Urvater aller prototypbasierten Sprachen
  - Newton-Script – Die Sprache des Urvaters aller PDAs
  - Java-Script – Die Sprache für dynamische Webseiten

Wie entstehend Objekte? - Spezifikation und Erzeugung

- Klassenbasierte Sprachen
  - Spezifikation durch „hinschreiben“:
    Die Variablen + Methoden können pro Objekt einzeln spezifiziert werden
    - Dadurch entsteht aber noch kein Objekt!



  - Erzeugung durch „Instantiierung“:
    Objekte werden durch Aufruf einer speziellen Operation aus der Spezifikation erzeugt
  - Klasse dient gleichzeitig als Modul mit Klassenvariablen und Methoden und als Schablone für Objekterzeugung durch „Instanziierung“
  - Objekte werden dementsprechend als „Instanzen“ bezeichnet

Beispiele klassenbasierter Sprachen

- Simula (1968)
  - Der Urvater aller objektorientierten Sprachen
- Smalltalk (1970)
  - Die erste „rein objektorientierte“ Sprache
    - „Rein“ heißt hier: „Alles ist ein Objekt!“ – Auch „elementare“ Datentypen!
- C++ (1979 / 1983)
  - Die erste effiziente Implementierung einer oo Sprache
- Eiffel (1986)
  - Die erste oo Sprache die für bestimmte Modellierungsprinzipien spezielle Sprachkonstrukte anbot

Klassen als „Objektschablonen“

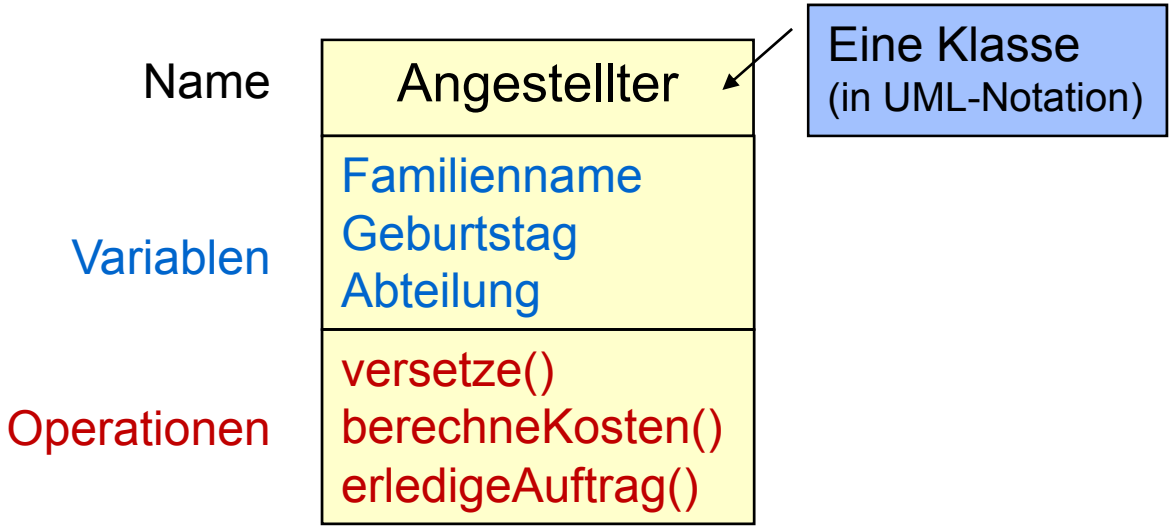
- Eine Klassendeklaration spezifiziert zusätzlich zu ihrem statischen Teil Instanz-Mitglieder:
  - Instanz-Felder: Variablen aus denen sich Objekte des Typs zusammensetzen
    - Auch „Attribute“, „Eigenschaften“, „Instanzvariablen“ (properties, instance variables, data members) genannt
  - Instanz-Methoden (instance methods) des Datentyps die auf den Objekten operieren
    - Auch Mitgliedsfunktionen (member functions) genannt
  - Die Konstruktoren (constructors), mit denen neue Objekte des Typs initialisiert werden

Instanz-Felder und Methoden

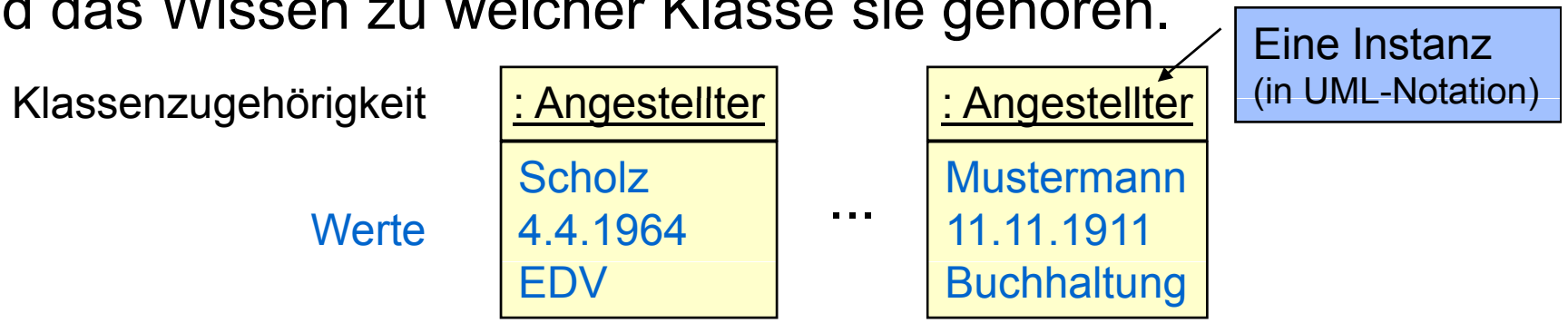
- Jede Instanz hat eigene Kopien der in ihrer Klasse spezifizierten Instanz-Variablen
  - Bei der Erzeugung eines Objekts (mit **new**) wird ein Speicherbereich reserviert, in dem neue Inkarnationen der Felder eingerichtet werden
- Jede Instanz hat in ihrer Klasse implementierten Instanz-Methoden
  - Diese werden jeweils beim Aufruf an ein Objekt „gebunden“ und operieren auf diesem Objekt
  - Wenn sie auf ein Feld einer Klasse zugreifen, dann ist für die Dauer des Aufrufs die Inkarnation des Feldes im gebundenen Objekt gemeint → siehe „this“

Klassifikation und Instantiierung: UML

Klassen definieren Schnittstelle, Operationen und Variablen ...

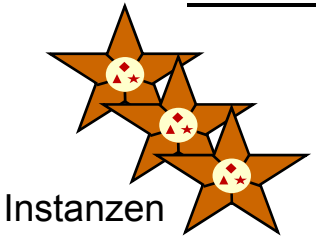


... für ihre Instanzen. Instanzen enthalten Variablen-Werte und das Wissen zu welcher Klasse sie gehören.



Klassifikation und Instantiierung: Java

- Klasse beschreibt Menge „gleichartiger“ Objekte
  - **gleiche** Schnittstelle
  - **gleiche** Implementierung
  - **gleiche** Variablen
  - **verschiedene** Variablen-Werte



Instanzen

Aufruf eines anderen Konstruktors der gleichen Klasse.

```
class Bike {
    // Instanz-Variablen:
    Bremse vorne, hinten;
    int gang = 18;

    // Instanz-Methoden:
    void schalten() {gang++;}

    // “Konstruktor”-Methoden:
    Bike(Bremse v, Bremse h) {
       vorne = v; hinten = h;
    }
    Bike() {
       this(...,...)
    }
}
```

- Klasse ist Schablone für Objekterzeugung

```
Bike meinRad = new Bike();
Bike deinRad = new Bike(v, h);
```

Klassen-Variablen und Klassen-Methoden

- Variablen und Methoden, die
  - nur ein mal pro Klasse existieren
  - für alle Instanzen zugreifbar sind
  - durch Nachrichten an die Klasse auch von außen zugreifbar sind:
    - Bike.verkaufe(10);
    - siehe auch Abschnitt über Sichtbarkeit
- Benutzung
  - gemeinsame Eigenschaften aller Instanzen
    - z.B. klassenspezifische Konstanten
  - Informationen über die Instanzen (Metainformationen)
    - z.B. Anzahl der Instanzen

```
class Bike {
 // Instanz-Variablen:
 ...; int gang = 18;

 // Klassen-Variable:
 static final int gänge = 21;

 // Instanz-Methode:
 void schalten() {
   if (gang < gänge) {gang++;}
 }

 // Klassen-Variable:
 static int nrOfBikes = 0;

 // Klassen-Methode:
 static verkaufe(int anz) {
   nrOfBikes = nrOfBikes - anz;
 }
}
```

Klassendeklaration: Beispiel

- Wir deklarieren eine Klasse **`Time`** für den Gebrauch in einer elektronischen Stoppuhr
- Die Instanzvariablen sec, min, hrs stellen die gemessene Zeit der *jeweiligen* Uhr dar
- Die Instanzmethode **`tick`** implementiert die Operation des Tickens des Sekundenzählers

```
class Time {
    byte sec=0; //seconds,0<=sec<60
    byte min=0; //minutes,0<=min<60
    int hrs=0;  //hours,  0<=hrs

    void tick() {
        sec++;
        if(sec >= 60) {
            sec -= 60;
            min++;
            if(min >= 60) {
                min -= 60;
                hrs++;
            }
        }
    }
}
```

Klassen als Typen

- Typdeklaration
  - Sei `K` der Name einer Klasse. Dann ist `K v;` die Deklaration einer Variablen `v` vom Typ `K`
- Objekterzeugung
  - Der Ausdruck `new K()` erzeugt ein neues Objekt vom Typ **K** und sein Wert ist die Referenz auf dieses Objekt
- Objektinitialisierung
  - Als Nebeneffekt der Ausführung von **new** wird das neue Objekt initialisiert
- Zugriff auf Instanz-Variablen und -Methoden
  - Man kann auf die Mitglieder des von **v** referenzierten Objekts mittels des Punkt-Operators `.` zugreifen

Zugriff auf Instanz-Variablen und -Methoden

- Eine Instanzvariable wird folgendermaßen angesprochen

```
objektausdruck.variablenname
```

```
Time t = new Time();
t.sec = 21; // Setze sec in t auf 21
t.min = 35; // Setze min in t auf 35
t.hrs = 2;  // Setze hrs in t auf 2
t.tick();   // Rufe tick() von Objekt t auf
```

- Eine Instanzmethode wird folgendermaßen aufgerufen

```
objektausdruck.methodenname(arg1, ..., argn)
```

universitätbonn

Initialisierung und Konstruktoren

Objekterzeugung und Initialisierung → new-Operator und Konstruktoren

Initialisierung von Klassen

Initialisierung und Konstruktoren

- Der new-Operator
  - Der Aufruf `new K()` reserviert den für eine neue Instanz der Klasse `K` den benötigten Speicherplatz
  - Er initialisiert die Instanzvariablen mit Varianten von Null, je nach dem Variablentyp
    - `0, 0d, 0f, \u0000, false` oder `null`
- Konstruktoren
  - Oftmals genügt die Initialisierung einer Variablen mit Null nicht
  - Zu diesem Zweck können in der Objekt-Klasse Konstruktoren (constructors) definiert werden

Initialisierung und Konstruktoren

- Konstruktoren haben Ähnlichkeit mit Klassen-Methoden, gelten aber nicht als Methoden
  - Alle Möglichkeiten der Zugriffskontrolle für Methoden gibt es auch für Konstruktoren
  - Aber sie haben keinen expliziten Ergebnistyp
    - Auch nicht den leeren Typ **`void`**!
    - Ihr impliziter Ergebnistyp ist die enthaltende Klasse
- Konstruktoren trägen den Namen ihrer Klasse
  - Verschiedene Konstruktoren einer Klasse unterscheiden sich lediglich in der Anzahl bzw. dem Typ ihrer Parameter
    - „overloading“ von Konstruktoren ist erlaubt

Initialisierung und Konstruktoren

- Ein Konstruktor kann einen anderen in derselben Klasse mit **`this(…)`** explizit aufrufen (explicit invocation)
  - Er kann einen Konstruktor der unmittelbaren Oberklasse mit **`super(…)`** aufrufen
- Ist in einer Klasse K kein Konstruktor definiert, so erzeugt der Übersetzer einen parameterlosen Standard-Konstruktor
  - `class K { K(){super();} ... }`
  - Dieser ruft den entsprechenden Konstruktor der Oberklasse auf.

Initialisierung und Konstruktoren: Beispiel A

- Die Klasse **`Date`** hat Konstruktoren der Stelligkeit 0 bis 3
- Der Nullstellige setzt das Datum auf den 1.1.0
- Die Konstruktoren der Stelligkeit n>0 rufen zunächst den (*n*-1)-stelligen Konstruktor explizit auf und überschreiben entweder
  - das Jahr,
  - Monat und Jahr,
  - oder Tag, Monat und Jahr

```
public class Date {
    private byte day, month;
    private short year;

    // Konstruktoren:
    public Date() {
        day=1; month=1;
    }
    public Date(short y) {
          this(); year=y;
    }
    public Date(byte m, short y) {
        this(y); month = m;
    }
    public Date(byte d, byte m, short y){
        this(m,y); day=d;
    }
    ...
}
```

Initialisierung und Konstruktoren: Beispiel A

- Damit sind folgende Initialisierungen von Variablen vom Typ Date möglich:

```
Date aday =
   new Date();
   // Der 1.1.0

Date bday =
   new Date((short) 1970);
   // Der 1.1.1970

Date cday =
   new Date((byte) 8, (short) 1975);
   // Der 1.8.1975

Date dday =
   new Date((byte)23, (byte)8, (short)2002);
   // Der 23.8.2002
```

```
public class Date {
    private byte day, month;
    private short year;

    // Konstruktoren:
    public Date() {
        day=1; month=1;
    }
    public Date(short y) {
          this(); year=y;
    }
    public Date(byte m, short y) {
        this(y); month = m;
    }
    public Date(byte d, byte m, short y){
        this(m,y); day=d;
    }
    ...
}
```

Initialisierung von Klassenvariablen

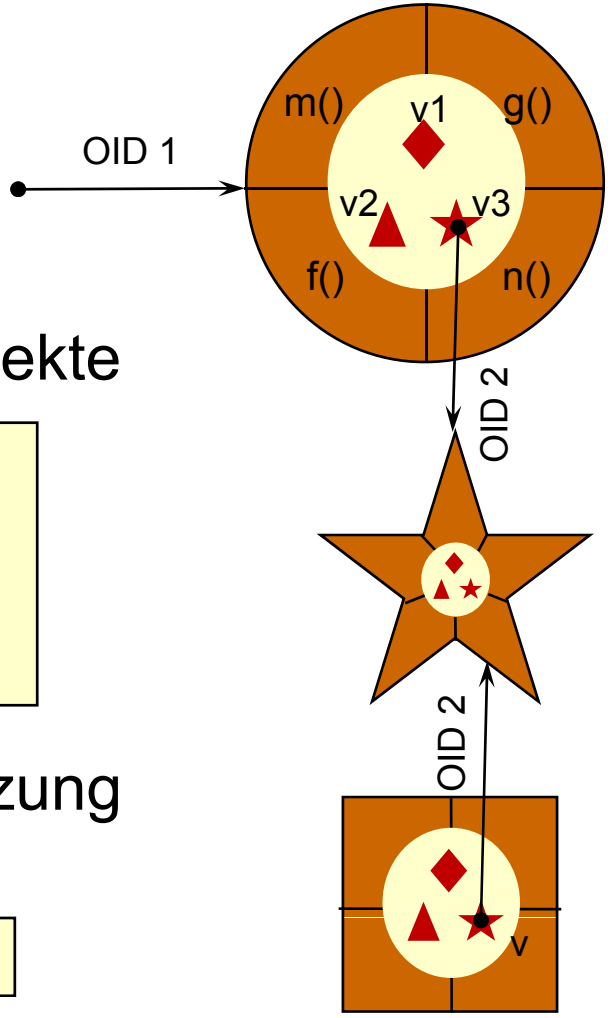
- Klassen werden beim ersten Gebrauch angelegt
  - incl. der Initialisierung ihrer Klassenvariablen
  - noch bevor Instanzen davon erzeugt werden.
- Initialisierung von Klassenvariablen
  - Vorinitialisierung mit (Varianten von) Null
  - Initialisierungsanweisungen von Klassenvariablen aussführen
    - Beispiel: `static int i=1;`
  - Statische Initialisierungsblöcke (static initializers) ausführen
    - Erlauben mehr Freiheit in der Initialisierung, inklusive des Aufrufs von Klassenmethoden. Beispiel: `static { i=f()+2; }`

Objektidentität

Objektidentität
Sharing
Aliasing

OOP: Objekt-Identität (OID)

- OID = Objekt-Referenz, die unabhängig ist von
  - Zustand des Objekts
  - Ort der Speicherung
  - ...



- Variablen speichern OIDs, keine Objekte

```
public class Kreis {
  ...
 Stern v3 = new Stern(..);
  ...
}
```

- OIDs ermöglichen gemeinsame Nutzung von Objekten („Sharing“)

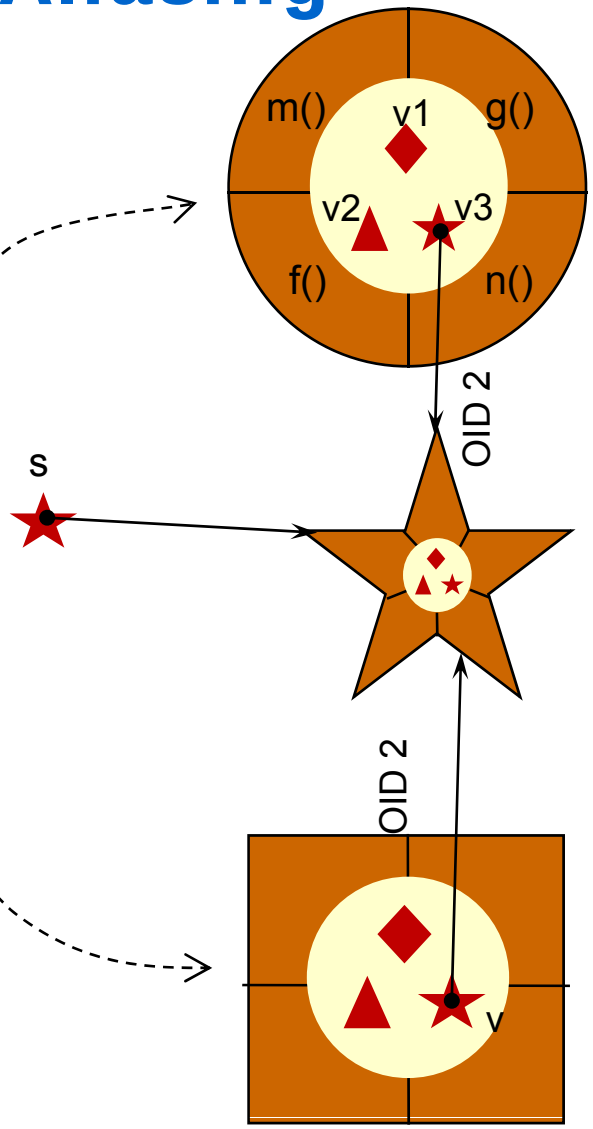
```
Stern v = v3;
```

Entstehung von Sharing / Aliasing

```
public class Kreis {
  ...
  Form v3;
  public Kreis(Form f) {v3 = f;}
  ...
}
```

```
  ...
  Stern s = new Stern();
  new Kreis(s);
  ...
  new Quadrat(s);
```

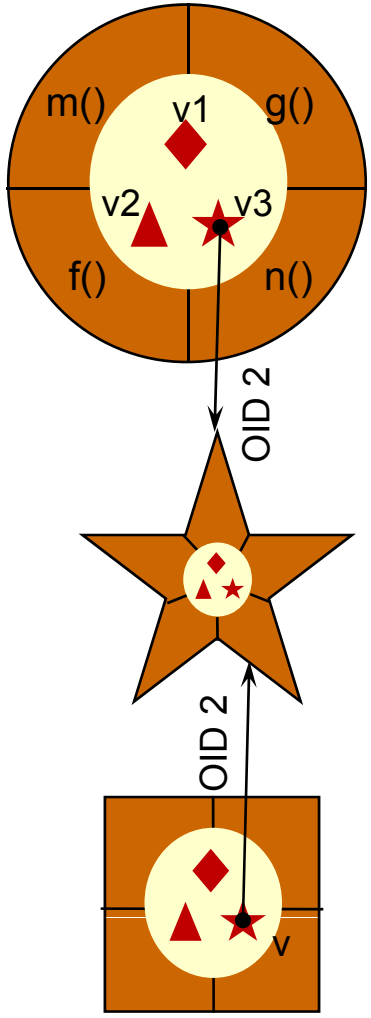
```
public class Quadrat {
  ...
  Form v;
  public  Quadrat(Form f) {v = f;}
  ...
}
```



Objekt-Identität → Sharing / Aliasing

Peter Grogono (Concordia University):
„Copying, Sharing and Aliasing“

- Nutzen: „Sharing“
  - Möglichkeit, explizit zu machen, dass zwei unterschiedliche Variablen das selbe Objekt referenzieren.
- Gefahr: „Aliasing“
  - Risiko, dass unterschiedliche Variablen das selbe Objekt referenzieren, ...
  - ... ohne dass man sich dessen bewusst ist.
  - Versehentliche Änderungen
- Sharing = Aliasing
  - unterschiedliche Seiten der gleichen Medaille.



Selbstbezug: this

Bezug auf das „aktuelle“ Objekt → this / self / current

Umsetzung als impliziter Parameter

Typisierung von this

Selbstbezug: this

- Problem
  - Die selbe Instanz-Methode kann von jeder Instanz einer Klasse ausgeführt werden!
  - Wie nimmt man in der Methode auf das Objekt Bezug, für das die Methode zur Laufzeit ausgeführt werden wird?
- Lösung
  - Variable, deren Wert nur vom Compiler / Laufzeitsystem gesetzt wird
  - Java, C++: „this“
  - Smalltalk: „self“
  - Eiffel: „current“

```
void registrierenBei(Amt a)  {
   a.fahrradRegistrieren(this)
}

void schalten(int gang) {
  if (gang>0 && gang<gänge) {
     this.gang = gang;
  }
}
```

- Konvention
  - this ist implizites Zielobjekt eines Zugriffs, wenn kein explizites Ziel angegeben ist
    - "schalten(3)" ist gleichbedeutend mit "this.schalten(3)"

Umsetzung von „this“: Impliziter Parameter einer jeden Instanz-Methode

- Ursprünglicher Java-Code

```
void registrierenBei(Amt a)  { a.fahrradRegistrieren(this) }
```

- Prinzip
  - „This“ wird vom Compiler jeder Instanz-Methode als erster Parameter hinzugefügt.
  - Jeder „this“-Zugriff ist ein Zugriff auf diesen Param.
  - Bei jeder Nachricht, wird das Zielobjekt als Wert von „this“ in der aufgerufenen Methode übergeben.
- Explizit gemachter „this“ Parameter

```
void registrierenBei(... this, Amt a)  { a.fahrradRegistrieren(a, this) }
```

In diesem Aufruf von fahrradRegistrieren() wird das Amt a der Wert des „this“-Parameters sein.

Der Typ von this

- Der statische Typ von this ist jeweils die enthaltende Klasse
  - In Klasse **`K`** hat **`this`** den statischen Typ **`K`**
- Der dynamische Typ von this ist jeweils die Klasse des Objektes für das eine Methode gerade ausgeführt wird
  - Durch „Vererbung“ kann es sein dass diese beiden Klassen unterschiedlich sind → Später

Interfaces, Subtypen und Nachrichten

Abstrakte Datentypen → Implementierung der von Methoden abstrahieren

Sehr abstrakte Datentypen → Interfaces → auch Datenstrukturen abstrahieren

Nutzen → Verschiedene Implementierungen des gleichen Datentyps → implements-Beziehung

Subtypen "revisited": Objektorientierte Subtypen

Unterschied von statischem und dynamischem Typ

Bisher: Konkrete Datentypen

- Klassennamen können in Typdeklaration verwendet werden
- Problem: Das ist zu starr
  - Klasse legt komplette Definition von Zustand und Verhalten fest (Datenstrukturen und Implementierung
  - Andere Implementierungen des gleichen Konzeptes werden somit ausgeschlossen
- Idee: Abstrakte Datentypen
  - Ein abstrakter Datentyp legt nur eine Datenstruktur („Trägermenge“) und die Signaturen der darauf arbeitenden Operationen fest
  - Verschiedene konkrete Datentypen können einen abstrakten Typ unterschiedlich implementieren

Schnittstellen: Abstrakter als abstrakt!

- Kritik an abstrakten Datentypen
  - Sie legen eine Datenstruktur („Trägermenge“) fest!
  - Somit besteht nicht mehr die Freiheit, die gleiche Funktionalität auf einer ganz anderen Datenstruktur zu implementieren
  - Beispiel: Suchen auf Listen, Bäumen, …
- Idee: Auch von Datenstruktur abstrahieren!
- Schnittstellen (Interfaces)
  - Benannte Menge von Operationsspezifikationen
  - Jeweils Name, Parametertypen, Ergebnistyp

Interface (Schnittstelle)

- Enthält Typen und Namen von Operationen aber keine Implementierung!
- Dient der Angabe der Operationen die ein Objekt mindestens bieten muss, um in einem bestimmten Kontext benutzbar zu sein
- ... unabhängig von seiner konkreten Implementierung

```
// Druckbare Objekte

public interface Printable {
  void print();
}
```

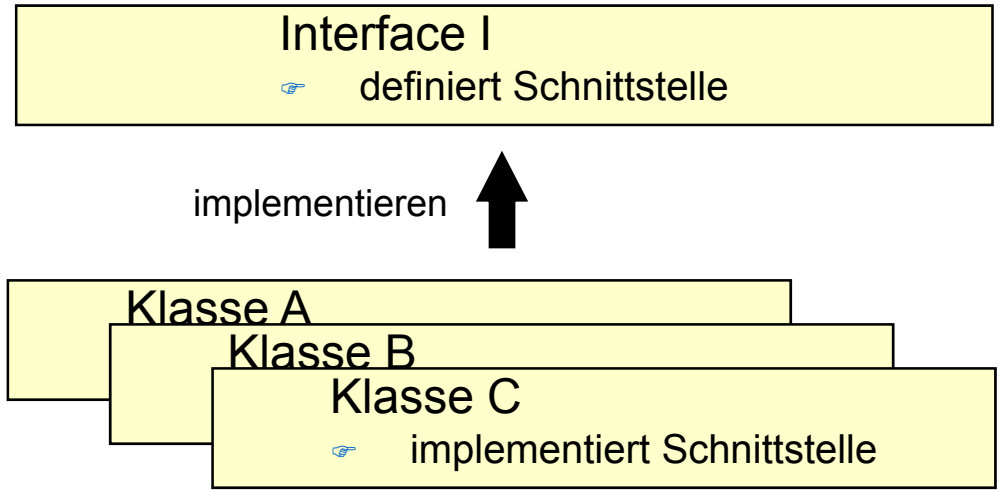
Alle Operationen im Interface sind implizit „public“

```
// x referenziert druckbare Objekte

Printable x;
...
x.print();
```

„Implementierungs“-Beziehung

- Klassen implementieren Interfaces



gleiches Interface kann von verschiedenen unabhängigen Klassen implementiert werden

```
public class List implements Enumeration {
}
```

```
public class StringTokenizer implements Enumeration {
  Boolean hasMoreElements() { ... };
  Object nextElement() { ... };
  ...
}
```

Die Implementierungs-Beziehung und die Subtyp-Beziehung

- Ersetzbarkeitsprinzip: **B** ist ein Subtyp von **A** ⇔ Instanzen von **B** sind immer für Instanzen von **A** einsetzbar
- Teilmengen-Kriterium: Die Menge der Instanzen von **B** ist eine Teilmenge der Instanzen von **A**

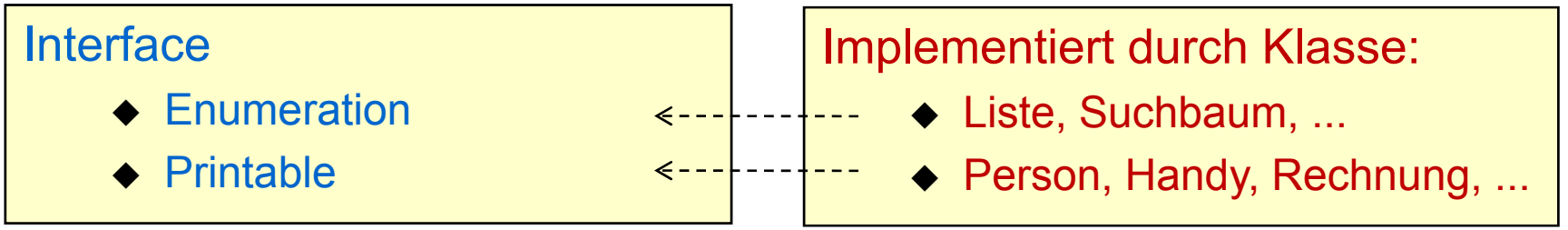
Konsequenz

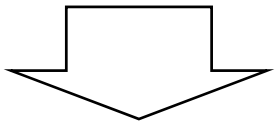
- Implementierende Klassen sind Subtypen der implementierten Interfaces
  - Ersetzbarkeit: In jeden Kontext einsetzbar wo das Interface erwartet wird
  - Teilmenge: Die Instanzen des Interfaces sind die Vereinigung der Instanzen aller implementierenden Klassen.

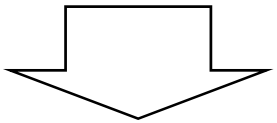
Interfaces versus Klassen

- Empfehlung
  - Zuerst möglichst eine Interface-Hierarchie entwerfen
  - Interface-Namen als Typdeklarationen verwenden
  - Klassen (die die Interfaces implementieren) nur zur Objekt-Erzeugung benutzen (siehe Beispiel auf der nächsten Seite)
- Warnung
  - Wenn ein Klassenname C als Typdeklaration verwendet wird (C var;), wird die Menge der einsetzbaren Untertypen auf die Unterklassen von C eingeschränkt
  - Instanzen von Klassen, die die gleiche Schnittstelle wie C implementieren, aber nicht von C erben, könnten nicht an var zugewiesen werden

Interfaces versus Klassen: Beispiel

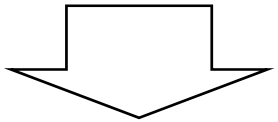






Design-Prinzip

Typdeklaration ➔ Interface Objekterzeugung ➔ Klasse



```
Enumeration e = new List(...); // Aufzählung druckbarer Objekte
Printable elem;

while (e.hasMoreElements()) {
    elem = (Printable) e.nextElement();
    elem.print();
}
```

Dynamisches Binden (Dynamic Binding)

Subtyping mit statischem Binden wäre sinnlos.

Nachrichten und Dynamisches Binden

Nachrichtenauflösung → Bestimmung der Signatur + dynamisches Binden

Implementierung von dynamischem Binden

Unterschied static versus nicht static für methoden

Nutzen der erweiterten Subtypbeziehung?

Typ- und Untertyp-Deklarationen

```
interface                Point { int getX(); int getY() }

class MyPoint implements Point {
    private int x; public int getX(){…}
    private int y; public int getY(){…}

}
```

Benutzung

```
Point p;                 // Variablendeklaration
MyPoint mp;              // Variablendeklaration
...
p = mp;                  // ok: Point ist Obertyp von MyPoint
...
p.getX();                // ok: getX() ist in Point enthalten
```

Was soll hier passieren? Welche Methode soll aufgerufen werden?

Hier wird die Ersetzbarkeit ausgenutzt

Dilemma: Bisher nur „statisches Binden“

Typ- und Untertyp-Deklarationen

```
interface                Point { int getX(); int getY() }

class MyPoint implements Point {
    private int x; public int getX(){…}
    private int y; public int getY(){…}

}
```

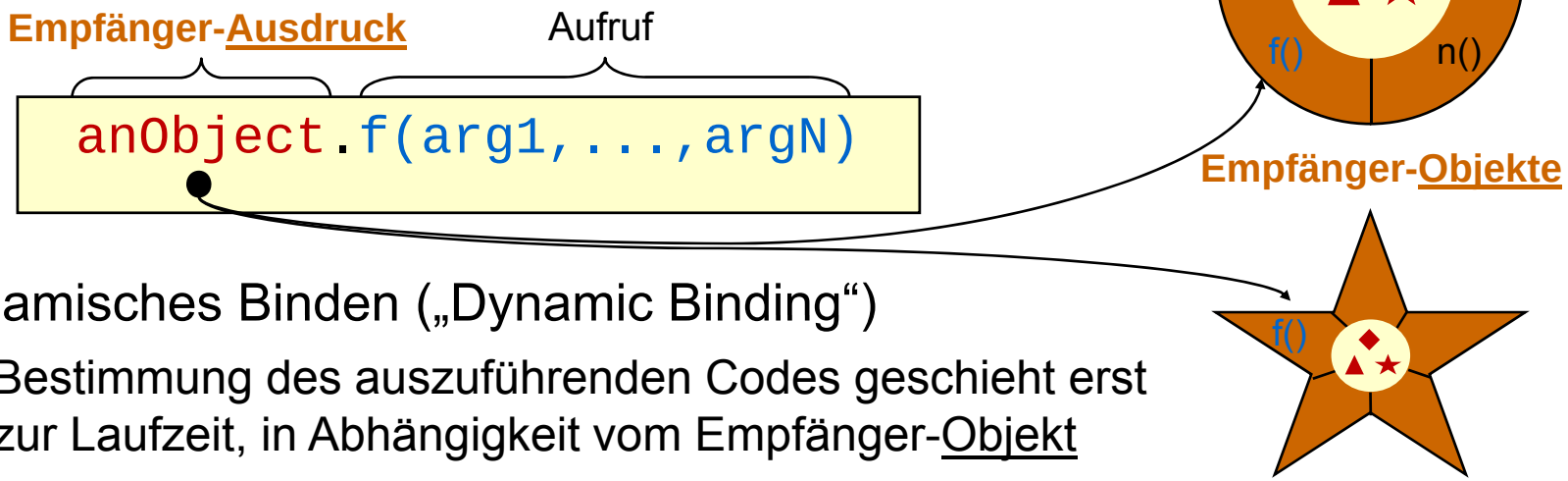
Benutzung

```
Point p;                 // Variablendeklaration
MyPoint mp;              // Variablendeklaration
...
p = mp;                  // ok: Point ist Obertyp von MyPoint
...
p.getX();                // ok: getX() ist in Point enthalten
```

- Statisches Binden: Führt die Methode aus dem statischen Typ des Empfänger-Ausdrucks aus
  - Leider hat **`Point`**, der statische Typ von **`p`**, keine Implementierung von `getX()` die man ausführen könnte ☹

Nachrichten und „Dynamisches Binden“

- Nachricht
  - Aufforderung an ein Objekt, eine seiner Methoden auszuführen
  - Java-Syntax: <empfängerausdruck>.<aufruf>



- Dynamisches Binden („Dynamic Binding“)
  - Bestimmung des auszuführenden Codes geschieht erst zur Laufzeit, in Abhängigkeit vom Empfänger-Objekt
  - Die Methode des Empfänger-Objekts wird ausgeführt!

- Nachricht ≠ Prozeduraufruf
  - Eine Nachricht → Verschiedene Effekte, je nach Empfängerobjekt!

Obiges Beispiel und „Dynamic Binding“

Typ- und Untertyp-Deklarationen

```
interface                Point { int getX(); int getY() }

class MyPoint implements Point {
    private int x; public int getX(){…}
    private int y; public int getY(){…}

}
```

Benutzung

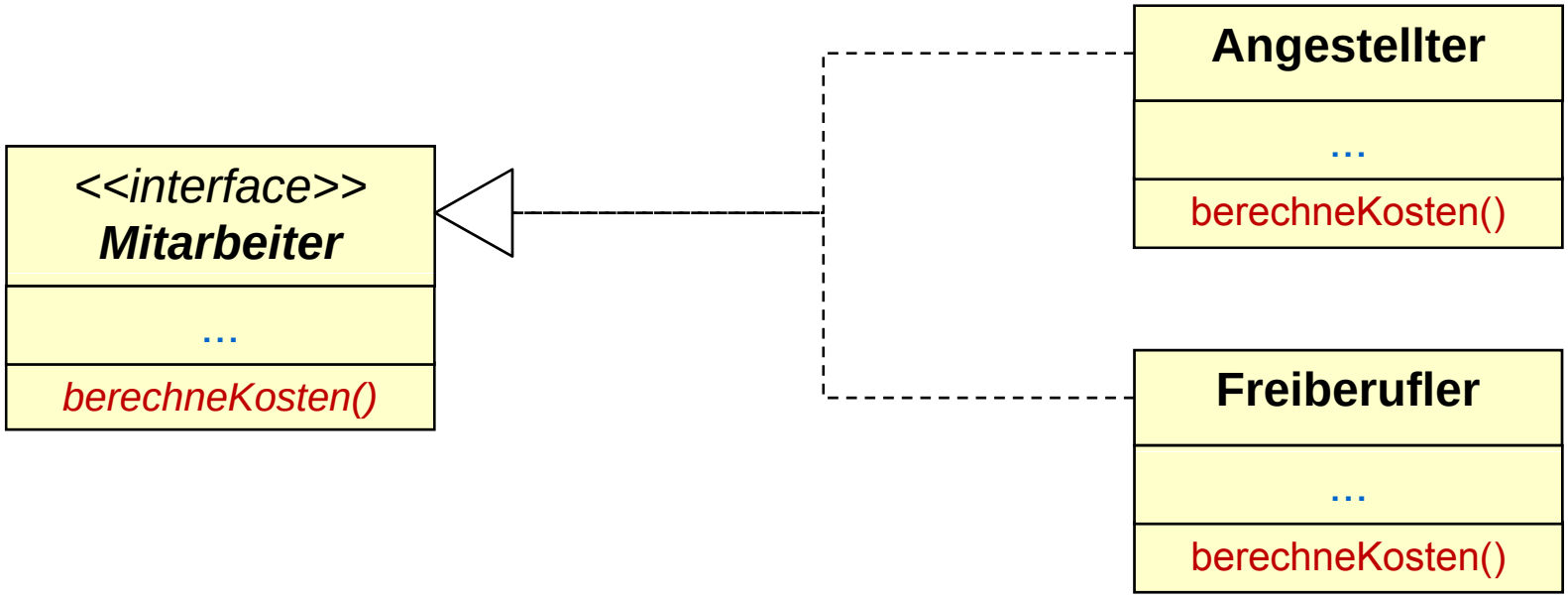
```
Point p;                 // Variablendeklaration
MyPoint mp;              // Variablendeklaration
...
p = mp;                  // ok: Point ist Obertyp von MyPoint
...
p.getX();                // ok: getX() ist in Point enthalten
```

Hier wird die Methode aus dem von p aktuell referenzierten MyPoint-Objekt aufgerufen!

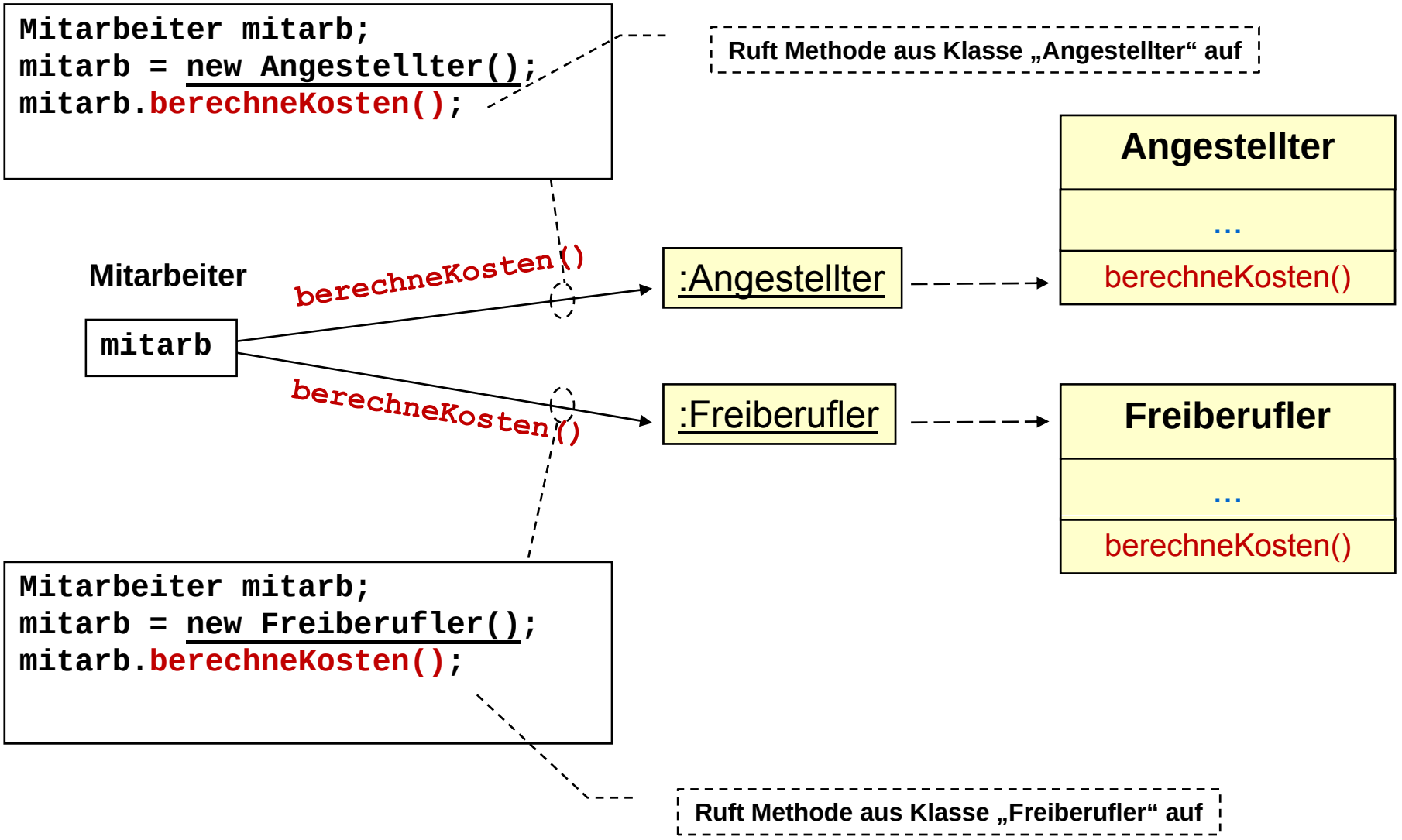
Hier wird die Ersetzbarkeit ausgenutzt

Nachrichten und dynamisches Binden: Beispiel

- Angenommen die Klassen „Angestellter“ und „Freiberufler“ implementieren das Interface „Mitarbeiter“



Nachrichten und dynamisches Binden: Beispiel (Fortsetzung)



Nutzung: Prozeduraufrufe versus Nachrichten

- Frage: Was kostet ein Angestellter?
  - ... für alle möglichen Arten von Angestellten!

Prozeduraufrufe (Pascal, C & Co)

```
Mitarbeiter *ma;
float kosten = 0;
switch (ma->anstellungsart) {
 case 1:        // Freiberufler
       kosten = kostenFreib(ma);
 case 2:        // Angestellter
       kosten = kostenAngeste(ma);
 case 3:        // Sonstiger
       kosten = kostenXXX(ma);
}
```

FEHLER
falls "anstellungsart" falschen Wert hat!

Nachrichten (Java & Co)

```
Mitarbeiter ma;
float kosten = 0;



kosten = ma.kosten();
```

Nutzen: Prozeduraufrufe versus Nachrichten

Prozedureaufrufe

```
switch type(anObject) {
 case Kreis: kreis_f(anObject,...);
 case Stern: stern_f(anObject,...);
}
```

- ☹ unterschiedlicher Aufruf und explizite Fallunterscheidung für jede Art von Empfänger
- ☹ Hinzufügen einer weiteren Fallunterscheidung für jede neue Empfängerart erforderlich
- ☹ ... an jeder Aufrufstelle im Programm!!!
- ☹ Fehleranfälligkeit

Nachricht

```
anObject.f(...)
```

- ☺ Dynamisches Binden
- ☺ gleiche Nachricht für alle möglichen Empfänger
- ☺ keine Änderung erforderlich um weitere Empfängerarten zu behandeln

➔ Vermeidung von Typ-Fehlern

➔ Wiederverwendbarkeit

➔ Erweiterbarkeit

Auflösen von Nachrichten

```
anObject.f(arg1, ..., argN)
```

- Zwei Schritte
  - Bestimmung der aufgerufenen Operation
  - Bestimmung der aufgerufenen Implementierung
- Bestimmung der aufgerufenen Operation
  - Bestimmung der Aufrufsignatur
  - Bestimmung der Kandidatensignaturen
  - Bestimmung der spezifischsten Kandidatensignatur
  - Alles wie in Folie 1-109 bis 1-118 beschrieben
    - Einzige Änderung: Die Bestimmung der Kandidatensignaturen betrachtet auch Signaturen aus den Oberklassen
- Bestimmung der aufgerufenen Implementierung
  - Dynamisches Binden

Ersetzbarkeit

B ist ein Subtyp von A

:⇔

Instanzen von **B** sind immer für Instanzen von **A** einsetzbar

⇔

Instanzen von **B** fordern höchstens und bieten mindestens
das gleiche wie Instanzen von **A**

Allgemienes Prinzip

⇒

Instanzen von B haben alle Methoden von A
mit genau gleichem Namen und gleichen Parameter-Typen,
sowie einem evtl. spezifischerem Ergebnis-Typ

ab Java 5

```
class MyPoint implements Point {
  private int x;
  public int getX(){…}
  private int y;
  public int getY(){…}
  private int z;
  public int getY(){…}
}
```

```
interface Point {
   int getX();
   int getY();
   void move(int x, int y);
}
```

Typanpassungen in Objekttyphierarchien

- Beispiel: Sei TNode ein Subtyp von Link der zusätzlich eine setData()-Methode enthält

```
{
    Link l;
    ...
    l = new TNode();          // OK: TNode is a Link.
    ...
    l.setData();              // ERROR: no setData() method in a
Link
    ((TNode) l).setData();    // OK: l refers to a TNode and
                              //     setData exists in TNode
}
```

„Type cast“ Operator

Wichtig: nur der deklarierte Typ wird geändert, nicht das Objekt!
(Da in Java die deklarierten Variablen stets Referenzen sind, kann dies in Java auch gut auseinander gehalten werden.)

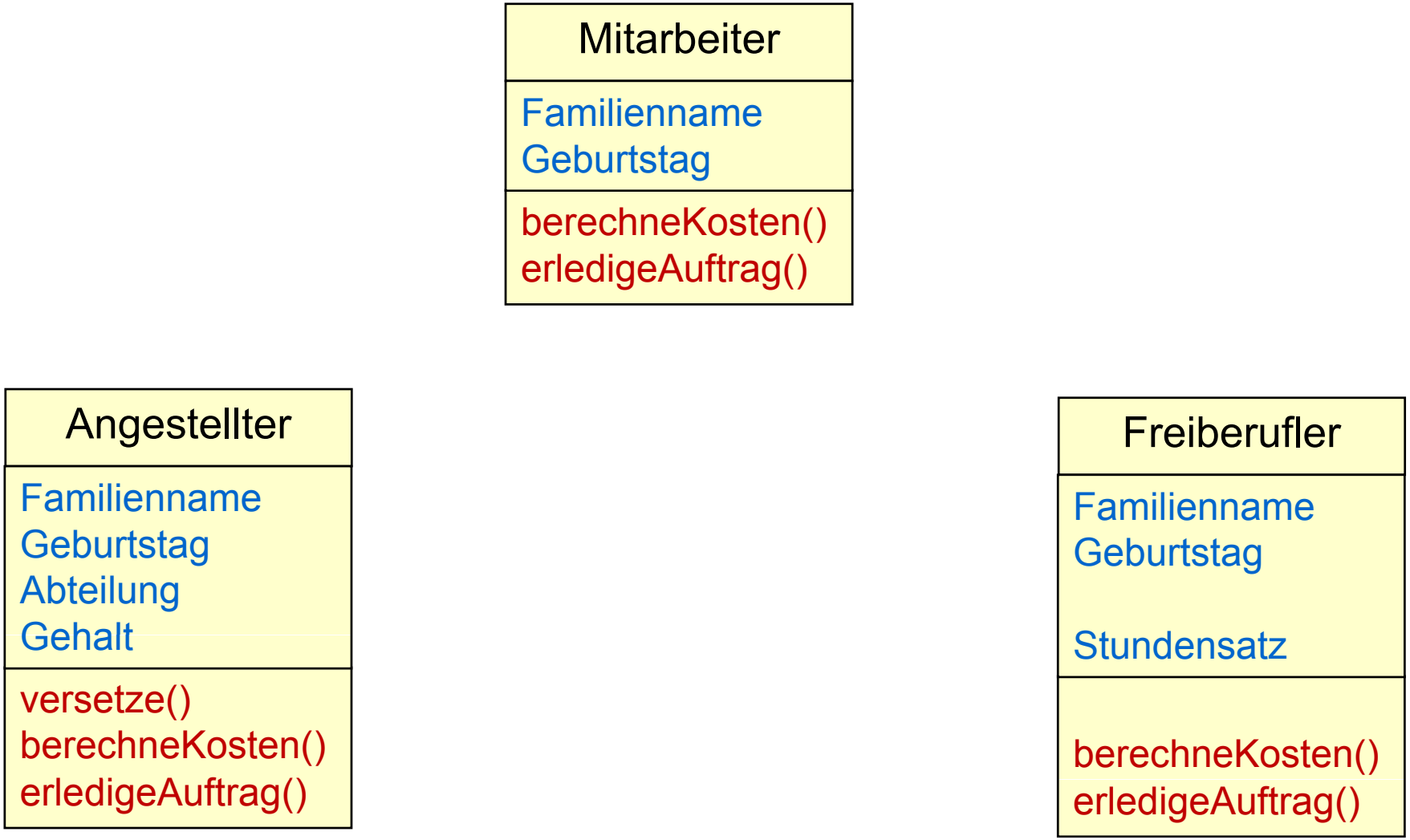
universitätbonn

Vererbung

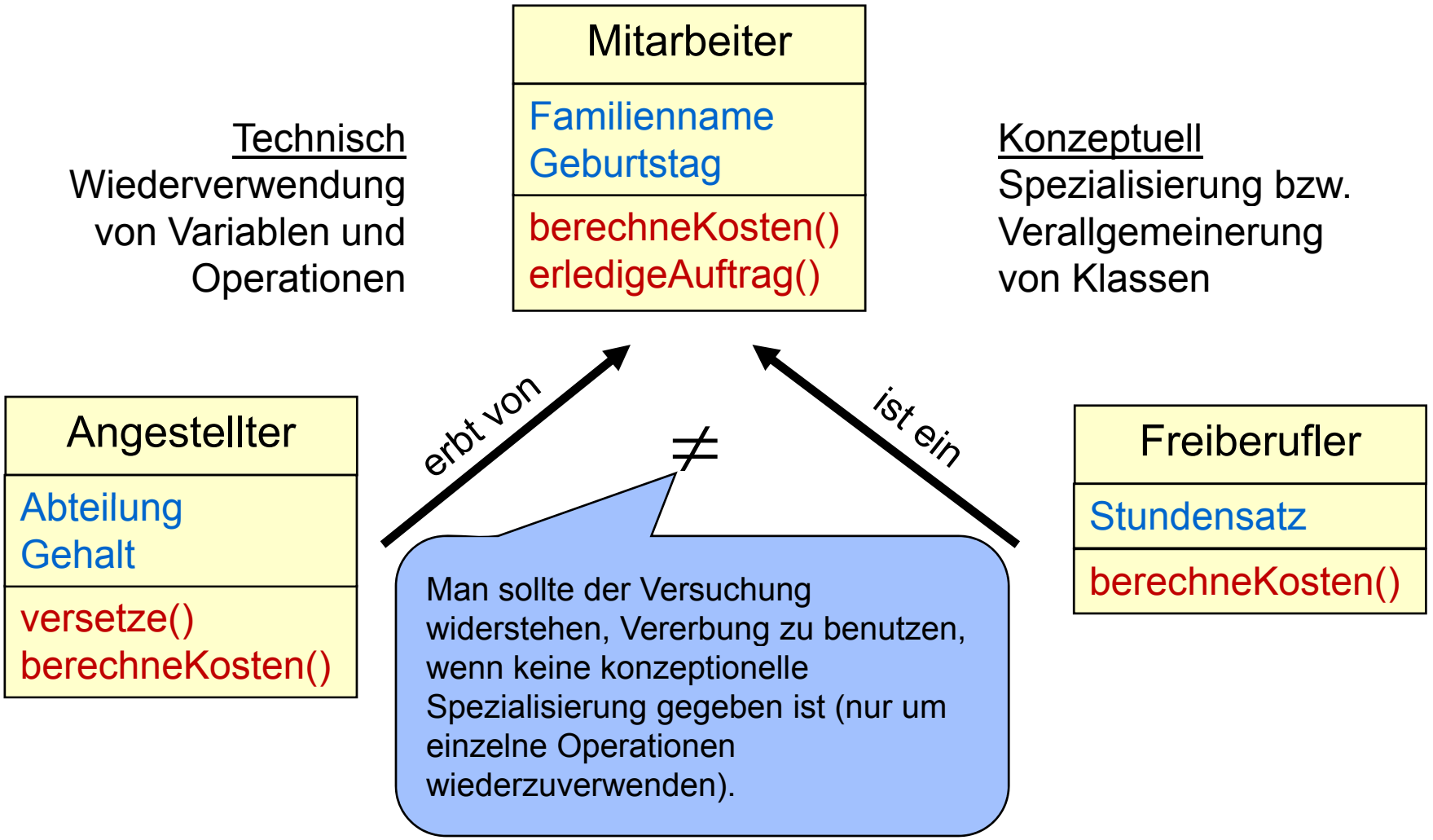
Vererbung, Verdecken und Überschreiben

Abstrakte Basisklassen

Vererbung


Mitarbeiter
Familienname
Geburtstag
berechneKosten()
erledigeAuftrag()
Angestellter
Familienname
Geburtstag
Abteilung
Gehalt
versetze()
berechneKosten()
erledigeAuftrag()
Freiberufler
Familienname
Geburtstag
Stundensatz
berechneKosten()
erledigeAuftrag()

Vererbung


Technisch
Wiederverwendung von Variablen und Operationen
Mitarbeiter
Familienname
Geburtstag
berechneKosten()
erledigeAuftrag()
Konzeptuell
Spezialisierung bzw. Verallgemeinerung von Klassen
Angestellter
Abteilung
Gehalt
versetze()
berechneKosten()
erbt von
≠
ist ein
Freiberufler
Stundensatz
berechneKosten()
Man sollte der Versuchung widerstehen, Vererbung zu benutzen, wenn keine konzeptionelle Spezialisierung gegeben ist (nur um einzelne Operationen wiederzuverwenden).

Vererbung in Java

- Unter-Klassen

```
class SubC extends SuperC { ... }
```

Java erlaubt nur eine Ober-Klasse ("Einfachvererbung")

  - **SubC** enthält implizit alle Methoden + Variablen aus **SuperC**, ...
  - ... die nicht lokal redefiniert (reimplementiert) wurden
    - ➔ Overriding

- Unter-Interfaces

```
interface SubI extends I1 ... In { ... }
```

Java erlaubt beliebig viele Ober-Interfaces ("Mehrfachvererbung")

  - `SubI` enthält implizit alle Methodensignaturen aus `I1 ... In`
  - verschiedene Signaturen für gleichen Namen möglich
    - ➔ kein Konflikt sondern verschiedene Methoden
    - ➔ Overloading

Vererbung und abgeleitete Klassen

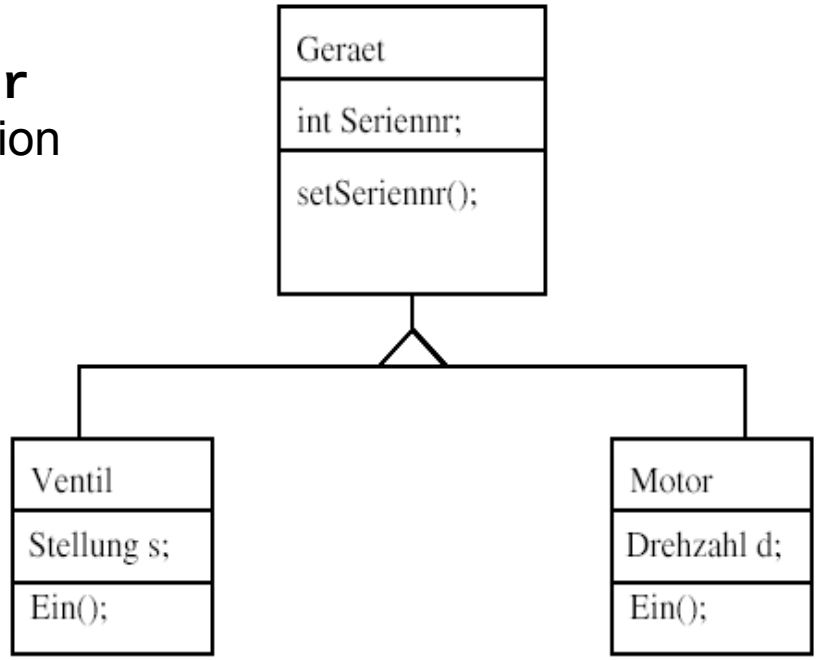
- Beispiel A:

Die Klasse **`Ventil`** und die Klasse **`Motor`** erben das Feld **`seriennr`** und die Funktion **`setSeriennr()`** von der Basisklasse **`Geraet`** **`Ventil`** und **`Motor`** sind abgeleitete Klassen

```
Ventil v = new Ventil();

// ein Ventil ist ein
Gerät
v.setSeriennr(1508);

// ein Ventil ist ein
Gerät Geraet g = new
Ventil();
```

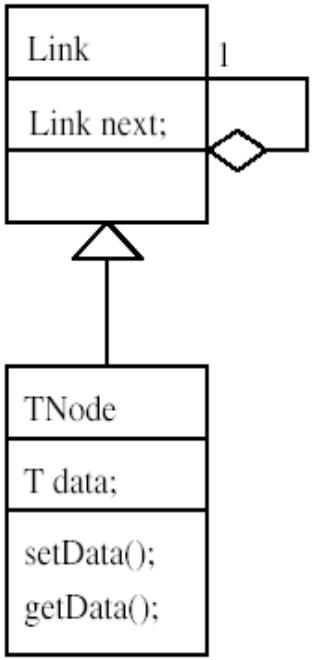


Klassendiagramm

Vererbung und abgeleitete Klassen

- Beispiel B:

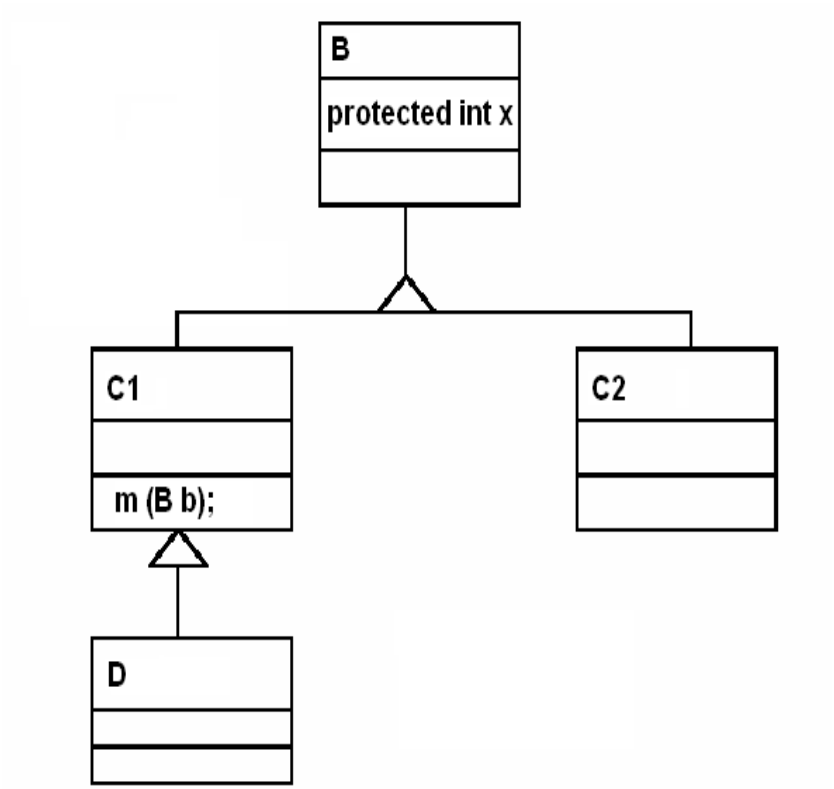
```
class Link {
    Link next;
}
class TNode extends Link {
    T data;
    public void setData(T
d){
        data = d;
    }
    public T getData() {
        return data;
    }
}
```



Klassendiagramm

Zugriff „protected“ in Klassenhierarchien

- Zugriffsschutz **`protected`** macht Mitglieder für Unterklassen sichtbar
- Auch für Unterklassen außerhalb des eigenen Pakets!



Konstruktoren in Klassenhierarchien

- In einer abgeleiteten Klasse **`C`** wird der no-arg Konstruktor der unmittelbaren Oberklasse **`B`** mit **`super()`** bezeichnet
  - Ein Konstruktor mit einem Argument **`arg`** als **`super(arg)`**, und so weiter
- Eine Methode der übergeordneten Klasse kann von der abgeleiteten Klasse aus mithilfe des Schlüsselworts **`super`** aufgerufen werden
  - wie z. B. in **`super.m()`**

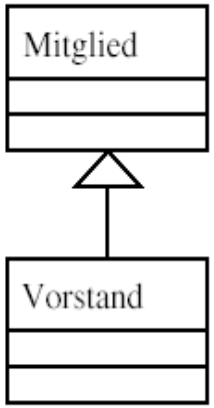
Beispiel: Hierarchie in einem Schachclub

- Basisklasse **`Mitglied`**

```
public class Mitglied {
    protected String name;  // Mitgliedsname
    protected int nummer;   // Mitgliedsnummer
    // Konstruktoren
    public Mitglied(String s, int n) {
        name = s;
        nummer = n;
    }
    // Selektoren:
    public int getNumber()
    { return nummer; }
    public String toString() {
        return "Name: " + name + ", Nummer: " +
nummer;
    }
}
```

Vererbung und abgeleitete Klassen

- Abgeleitete Klasse **`Vorstand`**




```
public class Vorstand extends Mitglied {
    protected String amt;  // Präsident,
Kassenwart, ...
    //Konstruktoren
    public Vorstand (String n, int m, String a)
    { super(n,m); amt=a; }
    public String toString() {
        return("Vorstandsmitglied: " +
super.toString()
                + ", Amt: " + amt);
    }
}
```

Konstruktor-Aufruf der Oberklasse

Aufruf einer Methode der Oberklasse

universitätbonn

Verdecken und Überschreiben

Verdecken von Feldern

- Die Deklaration eines Feldes in einer Klasse verdeckt oder verbirgt (hides) jedes Feld gleichen Namens in Oberklassen
  - Auch dann, wenn die Variablen verschiedenen Typ haben!
- Sowohl Klassenvariablen als auch Instanzvariablen können beim Zugriff auf ein Objekt verdeckt werden
- Über Obejktvariablen vom entsprechenden statischen Typ kann man auf verdeckte Felder zugreifen

Verdecken: Motivation

- Verschiedene Felder gleichen Namens sind (deshalb) zugelassen, damit die Basisklasse weiterentwickelt werden kann unabhängig davon, in welchen abgeleiteten Klassen sie schon verwendet wurde
  - Das Hinzufügen weiterer Felder und Methoden zu **B** verletzt den Kontrakt eines Obertyps ja nicht und muss deshalb erlaubt sein

Verdecken: Motivierendes Beispiel

- Programmierer, die mit der in einer Bibliothek implementierten einfach verketteten Liste arbeiten, möchten auch auf das letzte Element zugreifen
- Darum fügen sie einer abgeleiteten Klasse MyTList ein Feld last hinzu.

```
class TList {
    protected TNode head;
}
```

```
class MyTList extends TList
{
    protected TNode last;
}
```

Verdecken: Motivierendes Beispiel

- Gleichzeitig fügen auch die Entwickler der Bibliothek ein gleichnamiges Feld zu TList hinzu
  - … weil so oft nach diesem Feature gefragt wurde.

```
class TList {
    protected TNode head;
    protected TNode last;
}
```

```
class MyTList extends TList
{
    protected TNode last;
}
```

Verdecken: Motivierendes Beispiel

- Durch die Existenz zweier gleichnamiger Felder in einer Klassenhierarchie soll diese nicht unbrauchbar werden.
- Daher sind gleichnamige Felder auf verschiedenen Vererbungsstufen in Java erlaubt.

```
class TList {
    protected TNode head;
    protected TNode last;
}
```

```
class MyTList extends TList
{
    protected TNode last;
}
```

Verdecken von Klassenmethoden

- Eine Deklaration einer Klassenmethode verdeckt oder verbirgt (hides) alle Methoden gleicher Signatur in Oberklassen
  - Dies gilt nur für Methoden, die in der Unterklasse sichtbar sind → Private Klassenmethoden der Oberklasse werden nicht verdeckt!
- Es ist aber nicht erlaubt, dass eine Klassenmethode eine Instanzmethode verbirgt
  - … wohingegen eine Klassenvariable eine Instanzvariable verdecken darf!

Überschreiben von Methoden

- Die Deklaration einer Instanzmethode überschreibt (overrides) alle Instanzmethoden mit gleicher Signatur in Oberklassen
  - Dies gilt nur für Methoden, die in der Unterklasse sichtbar sind → Private Methoden der Oberklasse werden nicht überschrieben!
- Es ist in Java nicht erlaubt, Klassenmethoden oder als final deklarierte Instanzmethoden zu überschreiben
  - Dies führt zu einem Fehler bei der Übersetzung

Überschreiben und Verdecken : Beispiel

- Diese Beispiel verwenden wir im Folgenden mehrmals

```
class T { }

class Link {
    Link next;
    Link prev;
}

class TNode extends Link {
    T data;
    void setData (T d) {
        data=d;
    }
}

class TDNode extends TNode {
    Link prev;
    static int counter = 0;
    void setData(T d) {
        counter++;
        data = d;
    }
}
```

Super-Zugriff

- Das Schlüsselwort **`super`** kann als Empfänger-ausdruck benutzt werden, um auf Felder / Methoden in der unmittelbaren Oberklasse zuzugreifen
- Prinzip
  - Zugriff **`super.name`** in einer Methode von Klasse **`C`**, die **`B`** als unmittelbare Oberklasse hat
  - Dies ist einfach eine Abkürzung für **`((B) this).name`**!
- Verwendbarkeit von super
  - Überall, wo die Verwendung von **`this`** erlaubt ist

Super-Zugriff: Beispiel

- Im obigen Beispiel hätten wir **`setData`** in **`TDNode`** auch wie folgt implementieren können:

```
class TDNode extends TNode {
    static int counter = 0;
    Link prev;
    void setData(T d) {
        super.setData(d);
        counter++;
    }
}
```

Überschreiben und Verdecken: Zugriffsregeln

Bei einem Zugriff auf eine Variable oder einem Methodenaufruf sind immer zwei Fragen zu lösen

- Ist der Zugriff für den Empfängerausdruck gültig?
  - Kriterium: statischer Empfängertyp
- Welches von gegebenenfalls mehreren Mitgliedern gleichen Namens in der Klassenhierarchie wird ausgewählt?
  - Verdecken → statischer Empfängertyp genügt
  - Überschreiben → statischer Empfängertyp bestimmt die Signatur
  - Dynamisches Binden → dynamischer Empfängertyp bestimmt die Implementierung

Statische Typisierung

- Ist ein Variablenzugriff oder Methodenaufruf gültig?
  - Der statische Typ eines Ausdrucks bestimmt, welche Namen zulässig sind!
  - Der statische Typ ist der deklarierte Typ oder der vom Compiler aus Deklarationen ohne Datenflussanalyse herleitbare Typ
- Beispiel
  - Sei das Feld data und die methode getData in TNode aber nicht in Link definiert und öffentlich (public)

```
Link l = new TNode(); // OK: a TNode is a Link
l.data;               // ERROR: no data field in a Link!
l.getData();          // ERROR: no getData method in a Link!

TNode n = (TNode) l;  // OK: l is instance of a TNode.
n.data;               // OK: TNode has a data field.
n.getData();          // OK: TNode has a getData method
n.next                // OK: a TNode is a Link with next
```

Zugriffsregeln: Konsequenzen

- Beim Zugriff auf ein Feld ist der statische Typ des Empfängerausdrucks entscheidend
  - Entsprechende Deklaration oder type cast
- Der Zugriff auf verborgene Klassenvariablen und Klassenmethoden ist möglich indem man als Empfänger
  - … den qualifizierten Namen der Klasse verwendet
  - … eine Objektvariable verwendet, die den gewünschten statischen Typ hat

Überschreiben und Verdecken : Beispiel

```
class T { }

class Link {
    Link next;
    Link prev;
}

class TNode extends Link {
    T data;
    void setData (T d) {
        data=d;
    }
}

class TDNode extends TNode
    Link prev;
    static int counter = 0
    void setData(T d) {
        counter++;
        data = d;
    }
}
```

Folgende Programmzeilen greifen auf die im jeweils zugehörigen Kommentar genannten Felder zu

```
Link l = new TDNode();
l.prev = new Link();          // prev in Link
((TNode)  l).prev = null;     // prev in Link
(inherited)
((TDNode) l).prev = null;     // prev in TDNode

TNode n = (TNode) l;
n.setData(new T());           // setData() exists in
TNode
                              // setData() from TDNode
is
                              // executed on object l
                              // (counter is
incremented)

TDNode.counter = 0;           // qualified access to
counter
((TDNode) n).counter = 0;     // access to counter via
object
                              // reference of type
TDNode
```

Überschreiben und Verdecken

- Als wichtigste Regeln können wir uns merken:
  1. Es kann **nur** auf solche Attribute und Methoden zugegriffen werden, deren Namen aufgrund der Typdeklaration **gültig** sind
  2. Instanzmethoden überschreiben, Klassenmethoden verbergen
  3. Es werden immer die dem tatsächlichen Objekt zugehörigen Varianten von Instanzmethoden auf diesem ausgeführt
  4. Statische und finale Methoden **dürfen nicht** überschrieben werden
  5. Auf verborgene Felder und (Klassen-)Methoden kann man über qualifizierte Namen oder über Referenzvariablen vom passenden Typ zugreifen

universitätbonn

“Abstrakte” und “finale” Methoden

Abstrakte Methoden

Finale Methoden

Abstrakte Methoden

- Methoden haben üblicherweise eine Standardimplementierung in der Basisklasse, die bei Bedarf in der abgeleiteten Klasse abgeändert wird
  - **Fehlt** die Standardimplementierung, so spricht man von einer **abstrakten Methode** (abstract method) und von einer **abstrakten Basisklasse** (abstract base class)
  - Eine abstrakte Klasse **kann nur** als Oberklasse und als statischer Objekttyp dienen aber **nicht** zur Erzeugung von Objekten, da sie nicht voll implementiert ist
- Eine abstrakte Methode wird in Java durch das Schlüsselwort **`abstract`** gekennzeichnet

- In **C++** spricht man statt von einer abstrakten Funktion von einer **rein virtuellen Funktion** (pure virtual function)

Überschreibbare und Finale Methoden

- In Java ist jede Instanzmethode, die nicht weiter gekennzeichnet ist
  - dynamisch gebunden
  - potentiell überschreibbar
- Es gibt aber auch Instanzmethoden, die **nicht** überschreibbar sein sollen
  - Weil man sich aus Sicherheitsgründen darauf verlassen muss, dass an der Implementierung nichts geändert wurde
  - Zugriff kann auch minimal effizienter erfolgen

Überschreibbare und Finale Methoden

- Nicht überschreibbare Instanzmethoden heißen **endgültig** oder **final** (final)
  - Sie werden mittels des Schlüsselworts `final` gekennzeichnet
  - Finale Methoden können von abgeleiteten Klassen nur noch real geerbt aber **nicht** mehr überschrieben werden
  - Die mit `final` gekennzeichnete Implementierung ist von da abwärts die letzte Implementierung der Methode in der Klassenhierarchie
    - Im Gegensatz zu `static`-Methoden, die ja auch nicht überschrieben werden können, sind jedoch als final deklarierte Methoden nach wie vor Instanzmethoden, werden **beim Aufruf** an ein Objekt gebunden und können daher auch auf die Felder dieses Objekts zugreifen

Überschreibbare und Finale Methoden

- **Bemerkung zu C++**
  - In C++ hingegen sind alle Instanzmethoden implizit final
  - Virtuelle Funktionen müssen in C++ durch das Schlüsselwort **`virtual`** gekennzeichnet werden
  - Instanzmethoden in C++, die nicht als virtuelle Funktionen gekennzeichnet sind, können aber **verdeckt** werden, während dies bei Methoden in Java, die als final gekennzeichnet sind, nicht möglich ist
    - Siehe später
  - In C++ ist also zwischen dem **Überladen**, dem **Überschreiben** und dem **Verdecken** von Methoden zu unterscheiden

Finale Klassen

- Eine finale Klasse (final class) ist eine von der nicht mehr abgeleitet werden soll
  - Dies wird durch Angabe des Schlüsselworts `final` deklariert (`final class C … {…}`)
- Die vorliegende Implementierung von `C` ist damit endgültig
  - Alle Methoden einer finalen Klasse sind implizit final
- Eine abstrakte Klasse **darf nicht** final sein
  - Da ihre Implementierung sonst nie ergänzt werden könnte!

universitätbonn

Abstrakte Klassen

Abstrakte Klassen

Abstrakte Klassen und Interfaces

Abstrakte Klassen

- Definition und Syntax
  - Eine Klasse ist abstrakt (abstract class) wenn sie welche mindestens eine abstrakte Methode besitzt
  - Unterklassen, die nicht alle abstrakten Methoden implementieren, sind selbst abstrakt
  - In Java werden abstrakte Klassen durch das Schlüsselwort **`abstract`** gekennzeichnet
- Konsequenzen
  - Abstrakte Klassen dürfen nicht nicht instanziiert werden!
  - Abstrakte Klassen dürfen nicht final sein!
- Sinn
  - Abstrakten Klassen dienen der Vereinbarung von gemeinsamen Daten, Methoden und Schnittstellen von Unterklassen

Abstrakte Klassen: Beispiel

Beispiel 8.4.1. Jedes Gerät besitzt eine Seriennummer und eine Methode `ein()` zum Einschalten. Diese Methode ist in jedem konkreten Gerät vorhanden, aber immer verschieden implementiert. Eine allgemeine Implementierung in der Basisklasse `Geraet` ist nicht möglich. Diese Methode muß also in der Basisklasse abstrakt sein und daher auch die Basisklasse `Geraet`. Diese Klasse können wir auch nicht als Schnittstelle definieren, da wir das Attribut `seriennummer` in unserer Basisklasse definieren wollen (vgl. Abschnitt 8.4.2).

```
abstract class Geraet {
    int seriennummer;
    abstract void
ein();
}
```

Interfaces als sehr abstrakte Klassen

- Eine Schnittstelle (interface) kann als eine spezielle abstrakte Klasse aufgefasst werden,
  - bei der alle Methoden abstrakt sind
  - und die keine Attribute besitzt
    - außer solchen, die als public, static und final deklariert sind
- Entsprechungen
  - konkreter Datentyp → Klasse
  - abstrakter Datentyp → Abstrakte Klasse, die nur Felder und abstrakte Methoden definiert
  - noch abstrakterer Datentyp → Interface

Interfaces: Besonderheiten

- **Beispiel:** Die Deklaration des im Paket **`java.util`** enthaltenen Interface **`Enumeration`** ist von folgender Form:

```
package java.util;
public interface Enumeration {
    public abstract boolean hasMoreElements();
    public abstract Object nextElement() …;
}
```

`public abstract` **wird in Interface-Deklarationen nie explizit geschrieben, da implizit in der Definition des Interface-Konzepts enthalten!**

universitätbonn

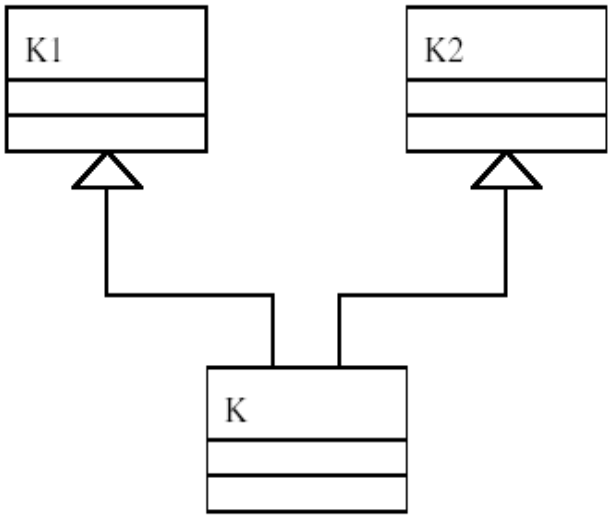
Multiple Vererbung

Definition

Probleme

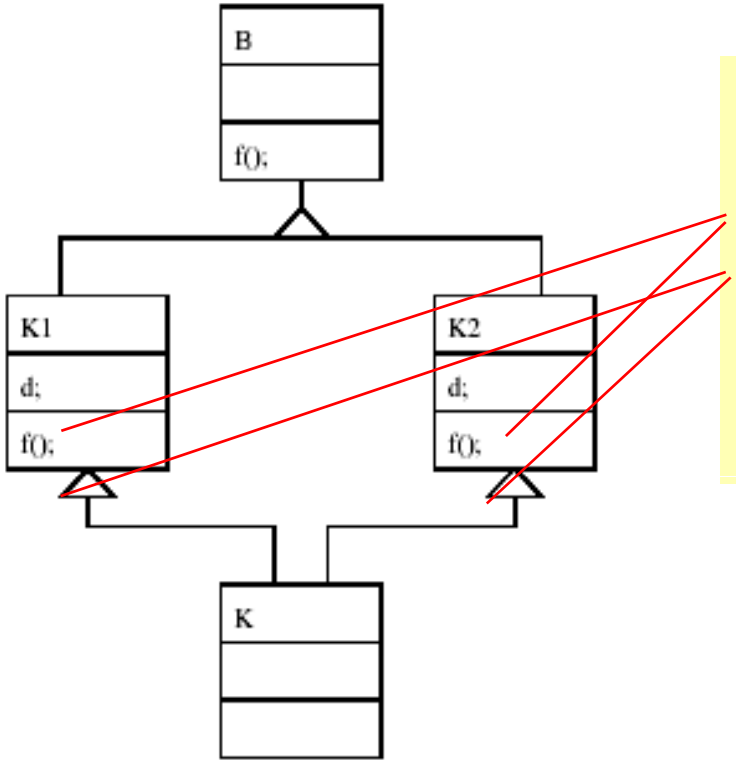
Mehrfachvererbung

- Ein Beispieldiagramm zu Mehrfachvererbung



Mehrfachvererbung

- Eines der Probleme bei der Mehrfachvererbung:




```
K v = new K();
v.d;   // Welches d ist
gemeint?
v.f(); // Welches f ist
gemeint?
```

Mehrfachvererbung

- Java erlaubt Mehrfachvererbung **nur** zwischen Interfaces
- Zwischen Klasse ist nur Einfachvererbung erlaubt
  - Die Einschränkungen in Java verhindern die problematischen Fälle, dass nicht-konstante Attribute von mehreren Klassen geerbt werden, oder virtuelle Funktionen wieder verwendet werden können, die nicht neu implementiert (überschrieben) werden müssen