Teil 2. Objektorientierte Konzepte

-- 9. Juni 2010 --

universitätbonn

Objekte und Klassen

Gekapselte, dynamisch erzeugbare „Module“→ Objekte

Schablonen zur Objekterzeugung → Klassen

Klassen- versus Instanzmitglieder

Zustand und Verhalten

Von Modulen zu Objekten (1)

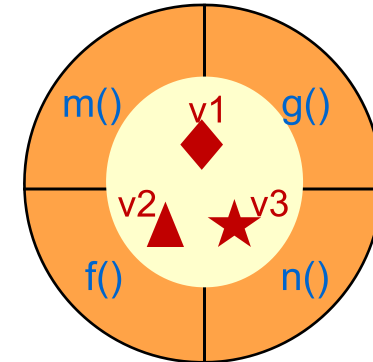
- Module sind statisch!
  - Man kann nur eine endliche Anzahl davon aufschreiben
- Die Welt ist dynamisch!
  - Man muss oft neue „Einheiten“ nach Bedarf erzeugen
  - Man weiß oft nicht in vorhinein wann und wie viele
  - Beispiel: Es gibt nicht nur ein Auto in der Welt!
- Idee: Alle Autos im gleichen 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

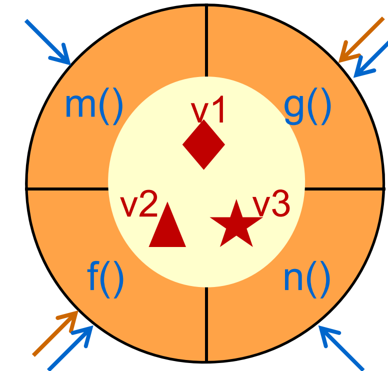
- Objekte
  - Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
    - Variablen → Zustand
    - Operationen → Verhalten



- 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

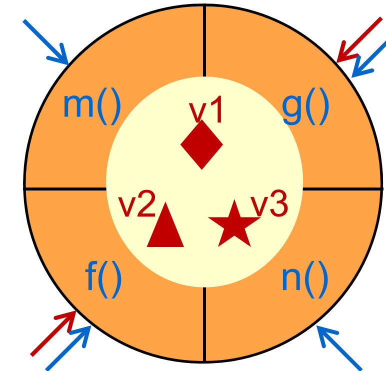
- Objekte
  - Gekapselte, dynamisch erzeugte Einheiten von Daten und Operationen
    - Variablen → Zustand
    - Operationen → Verhalten



- 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

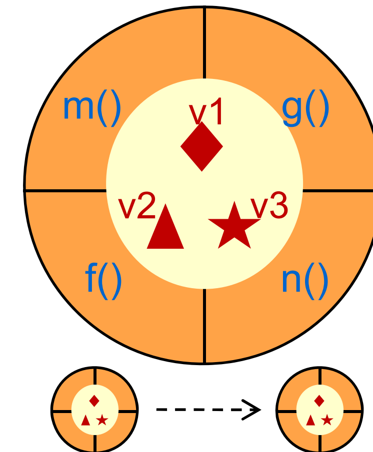


- Kapselung
  - Sprache stellt sicher, dass der Zustand eines Objektes nur über die in seiner Schnittstelle spezifizierten Operationen manipuliert wird
  - ➔ Bei gleichbleibendem Interface wirken sich Änderungen der lokalen Implementierung nicht auf andere Objekte aus

➔ Wartbarkeit und Zugriffs-Synchronisation!

Wie entstehend Objekte? - Spezifikation und Erzeugung

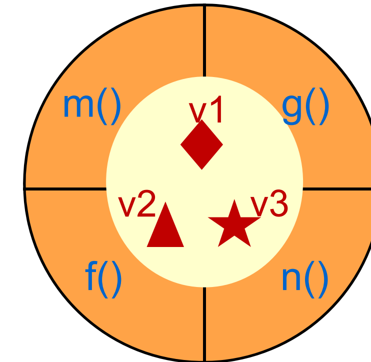
- Objekt-/Prototypbasierte Sprachen
  - Erzeugung durch „hinschreiben“:
    Die Variablen + Methoden können pro Objekt einzeln spezifiziert werden
  - Erzeugung durch kopieren („clonen“):
    Objekte werden durch Kopieren von bestehenden Objekten erzeugt und danach verändert



- 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"

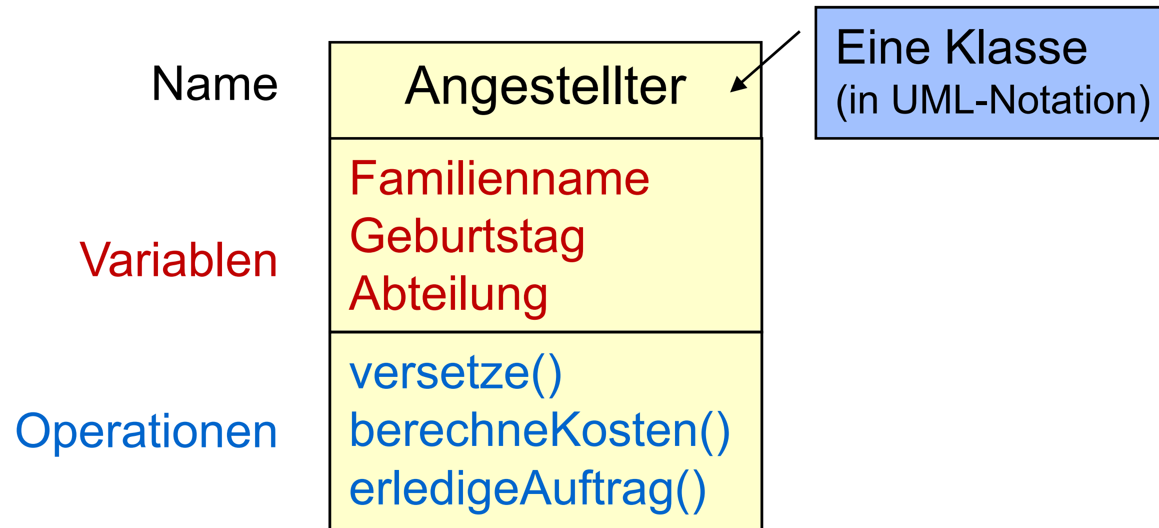
- Eine Klassendeklaration spezifiziert zusätzlich zu ihrem statischen Teil Instanz-Mitglieder:
  - Instanz-Felder: Variablen aus denen sich Objekte des Typs zusammensetzen
    - Auch „Attribute", „Eigenschaften", „Instanzvariablen" (properties, instance variables, data members) genannt
  - Instanz-Methoden (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

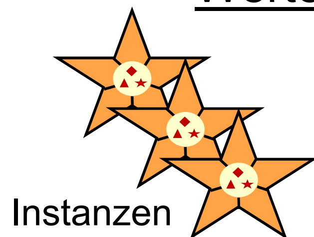
Klassen definieren Schnittstelle, Operationen und Variablen ...



... 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




```
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(...,...)
    }
}
```

Aufruf eines anderen Konstruktors der gleichen Klasse.

- 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;
 }
}
```

Objekte: Deklaration und Erzeugung

- Objektvariablen-Deklaration
  - 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`
  - Sein Ergebnis ist die Referenz auf dieses Objekt
  - Z.B. zeigt `v` nach Folgendem auf eine neue K-Instanz

```
K v;
v = new K();
```

- Objektinitialisierung
  - Als Nebeneffekt der Ausführung von `new` werden die Instanzvariablen des neuen Objektes initialisiert
    - Siehe Abschnitt „Initialisierung und Konstruktoren“

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++;
            }
        }
    }
}
```

Zugriff auf Instanz-Variablen und -Methoden

Sei `objektausdruck` ein Ausdruck der als Ergebnis eine Referenz auf ein Objekt liefert.

- Der Zugriff auf eine Variable des referenzierten Objektes erfolgt durch:

```
objektausdruck.variablenname
```

- Der Aufruf einer Methode des referenzierten Objektes erfolgt durch :

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

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

universitätbonn

Initialisierung und Konstruktoren

Objekterzeugung und Initialisierung von Instanzvariablen → new-Operator und Konstruktoren

Initialisierung von Klassen und Klassenvariablen

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 anschließend
  - 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;
    }
    ...
}
```

- Frage: Sind die expliziten Typkonversionen erforderlich und wenn ja warum?
  - Tip: Siehe Kapitel 1, Abschnitt „Ausdrücke: Typkonversionen“ (S. 1-64 bis 1-75)

Initialisierung von Klassenvariablen

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

Objektidentität

Objektidentität

Sharing

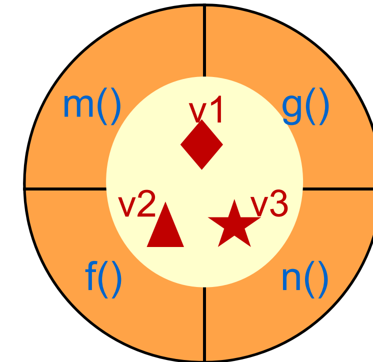
Aliasing

OOP: Objekte

Objekt

Dynamisch erzeugte, gekapselte Einheit von Daten und Code

- Variablen → Zustand
- Operationen → Verhalten



- Dynamisch erzeugt
  - `new` Operator

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

```
public class Kreis {
  public Raute v1;
  public Dreieck v2;
  Stern v3;
  public void g() {}
  public void f() {}
  void m() {}
  void n() {}
}
```

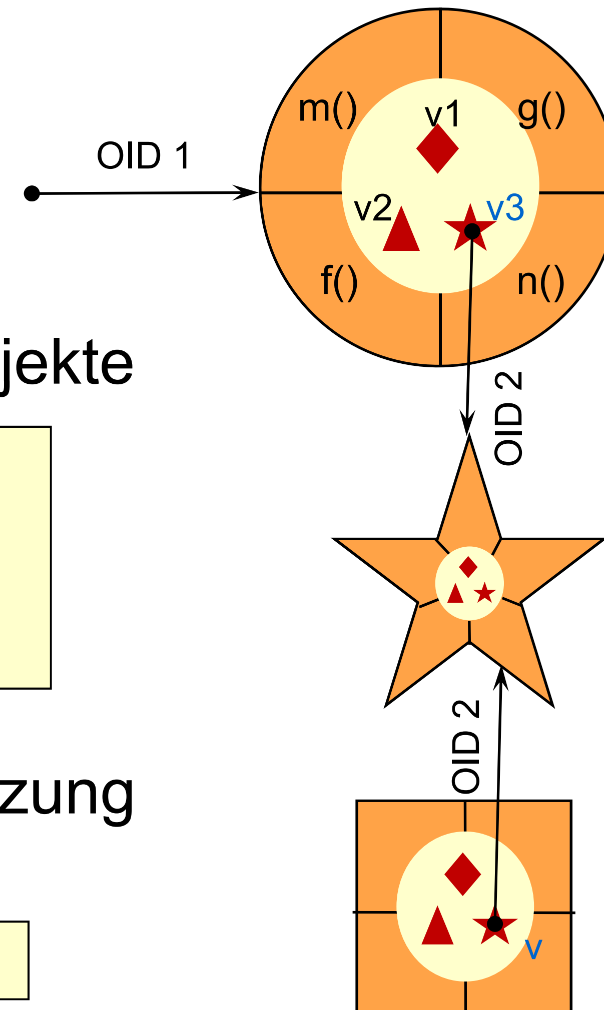
OOP: Objekt-Identität (OID)

- 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;
```

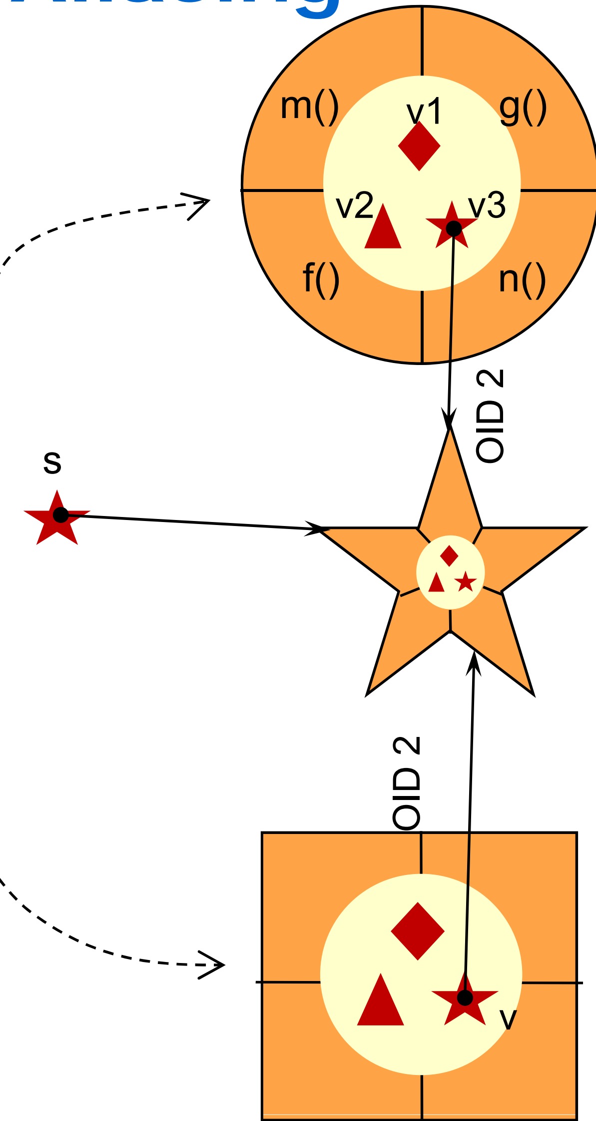


Entstehung von Sharing / Aliasing

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

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

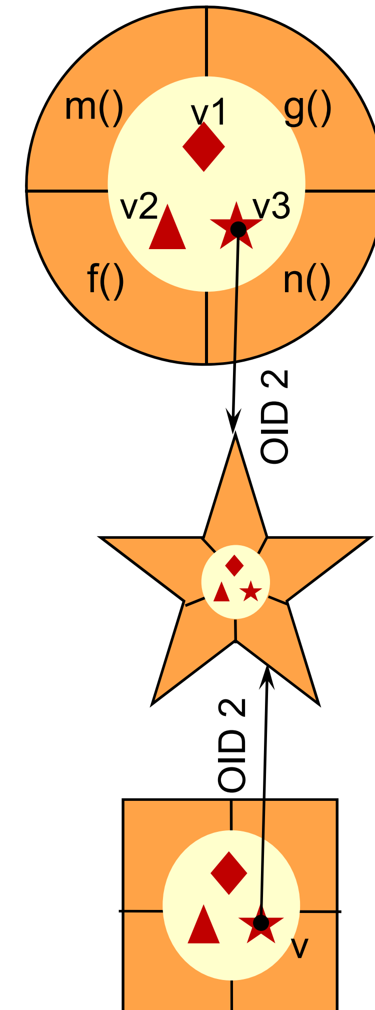
```
public class Quadrat {
   ...
   Form v;
   public  Quadrat(Form f) {v = f;}
   ...
}
```



Objekt-Identität → Sharing / Aliasing

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

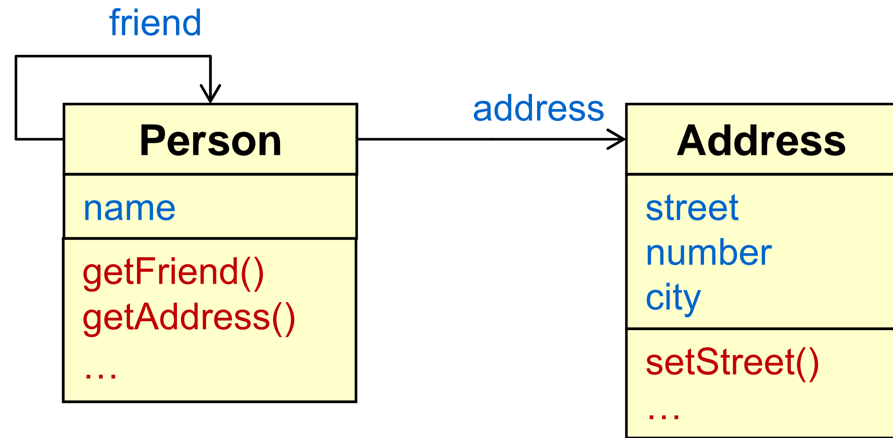
- Nutzen: „Sharing“
  - Möglichkeit, explizit zu machen, dass zwei unterschiedliche Variablen das selbe Objekt referenzieren.
- Gefahr: „Aliasing“
  - Risiko, dass unterschiedliche Variablen das selbe Objekt referenzieren, ...
  - ... ohne dass man sich dessen bewusst ist.
  - Versehentliche Änderungen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

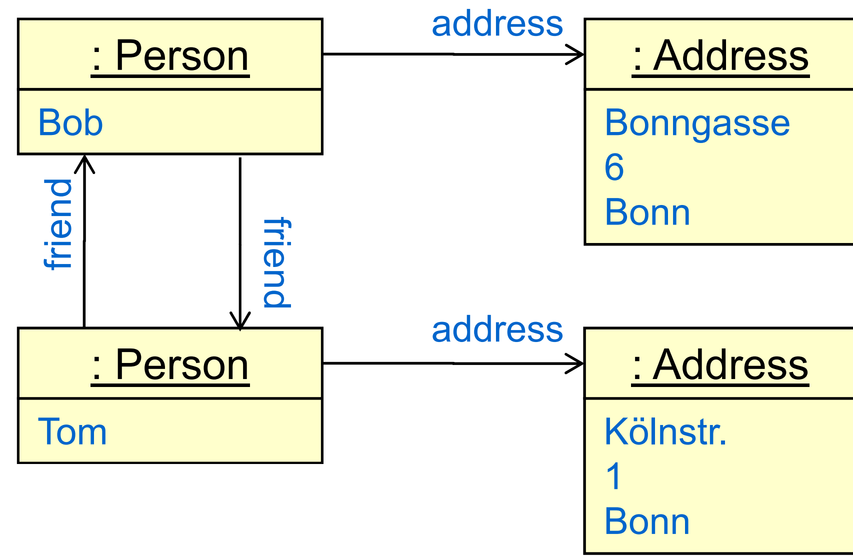


Beispiel: Person / Addresse

```
public class Person {
    private String name;
    private Address address;
    private Person friend;

    public Address getAddress() {
    return address;
    }
    public void sendLetterToFriend(Letter l) {
        Address fa = friend.getAddress() ;
        l.sendTo(fa);
    }
    public void sillyOrMalign() {
        Address fa = friend.getAddress() ;
        fa.setStreet("Ätsch!");
    }
}
```

```
public class Address {
    private String street;
    …
    public void setStreet(String str){
    street = str;
    …
}
```





Beispiel READONLY: Person / Addresse

```
public class Person {
    private Address address;                          // My own address is writeable to me
    private readonly Person friend;                   // I cannot modify my friend but I can
                                                      // make another Person my friend
    public readonly Person getFriend() readonly{      // I can't return more rights than I have
        return friend;
    }
    public void setFriend( readonly Person p) {       // I will never modify my friend
        friend = p;
    }
    public Address getAddress() readonly {            // My address has the same protection as I
        return address;
     }
    public void setAddress(Address a) {
    address = a;
    }
    public mutable Person dClone() readonly {
        Person child = new Person();
        child.setFriend(this);
        child.setAddress(this.address.dClone());
        return child;
     }
}
```

Beispiel READONLY: Person / Addresse

```
public class Address {
    private String street;
    private int number;
    private String city;

    public String getStreet() readonly {            // The street has the same protection as I
        return street;
    }
    public void setStreet(String str) {             // Those who may modify me may set
        street = str;                               // another street
    }
    public mutable Address dClone() readonly {      // dClone does not modify me but returns
         ...                                        // a modifiable copy of myself
    }
}
```

Selbstbezug: this

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

Umsetzung als impliziter Parameter

Typisierung von this

Selbstbezug: „this“

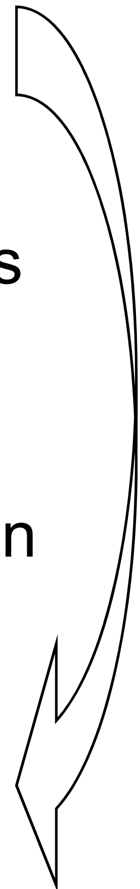
- Problem
  - Die selbe Instanz-Methode kann von jeder Instanz einer Klasse ausgeführt werden!
  - Wie bezieht man sich in der Methode auf das Objekt für das die Methode zur Laufzeit gerade ausgeführt wird?
- Lösung
  - Variable, deren Wert nur vom Compiler / Laufzeitsystem gesetzt wird
  - Java, C++: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

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

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

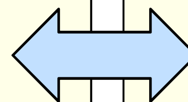
```
void registrierenBei(... this, Amt a) { a.fahrradRegistrieren(a, this) }
```

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

Implizite Ergänzung von „this“

- this ist implizit das Zielobjekt eines Feldzugriffs / Methodenaufrufs wenn kein explizites Ziel angegeben ist

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



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

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

Implizite Ergänzung von „this“ (2)

- this ist implizit das Zielobjekt eines Feldzugriffs / Methodenaufrufs wenn kein explizites Ziel angegeben ist

  … und es keine lokale Variable / keinen Parameter mit gleichem Namen gibt

- Prinzip: Namen in „inneren“ Blöcken verdecken Namen in „äußeren“ Blöcken
  - Hier bezieht sich „gang“ auf den Parameter, this.gang auf das Feld:

```
class Fahrrad {
    static int gänge = 21;
    int gang = 18;

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

Interfaces, Subtypen und Nachrichten

Abstrakte Datentypen → Implementierung 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

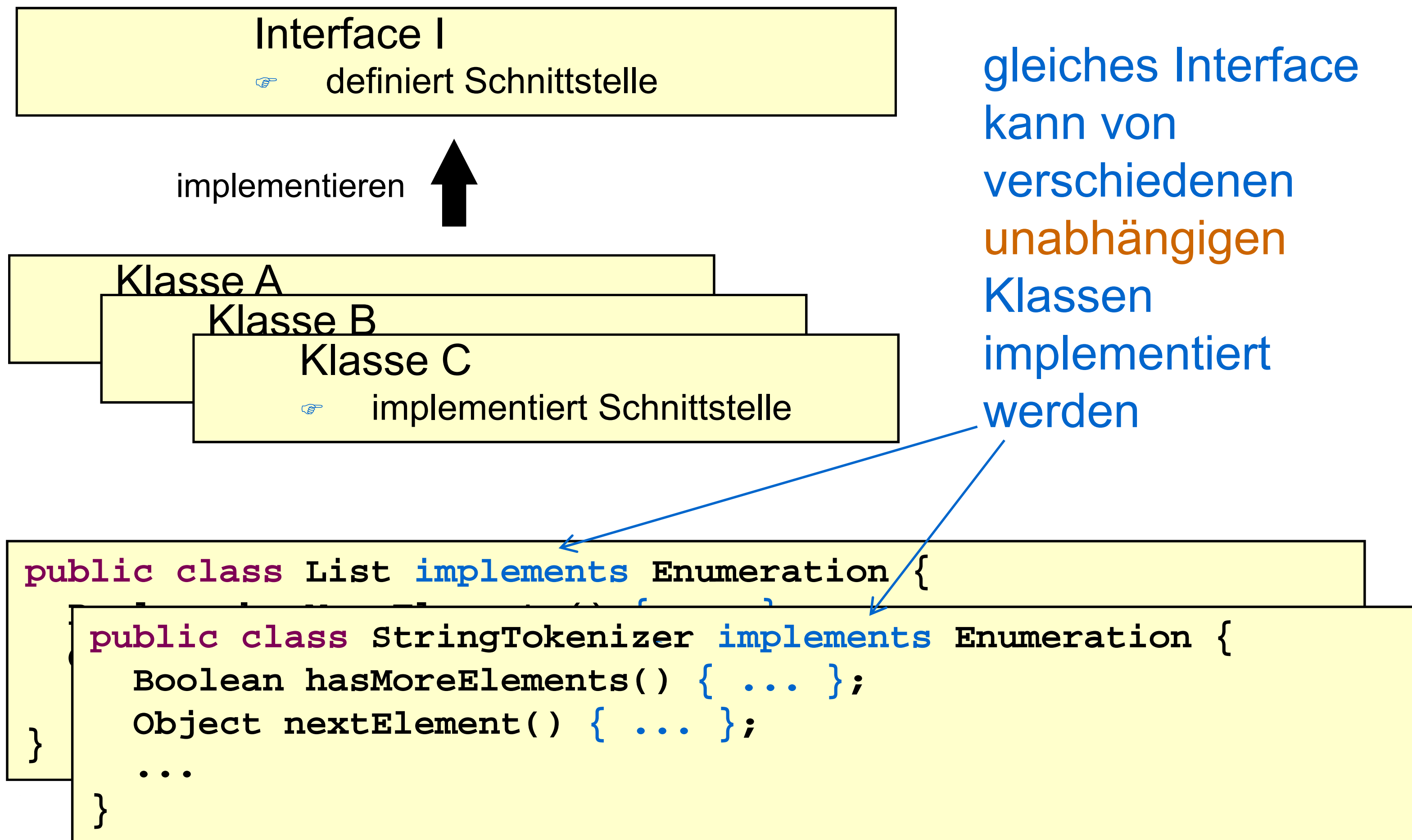
```
// Druckbare Objekte

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

```
// x referenziert druckbare Objekte

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

Alle Operationen im Interface sind implizit „public"

„Implementierungs"-Beziehung

- Klassen implementieren Interfaces



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**

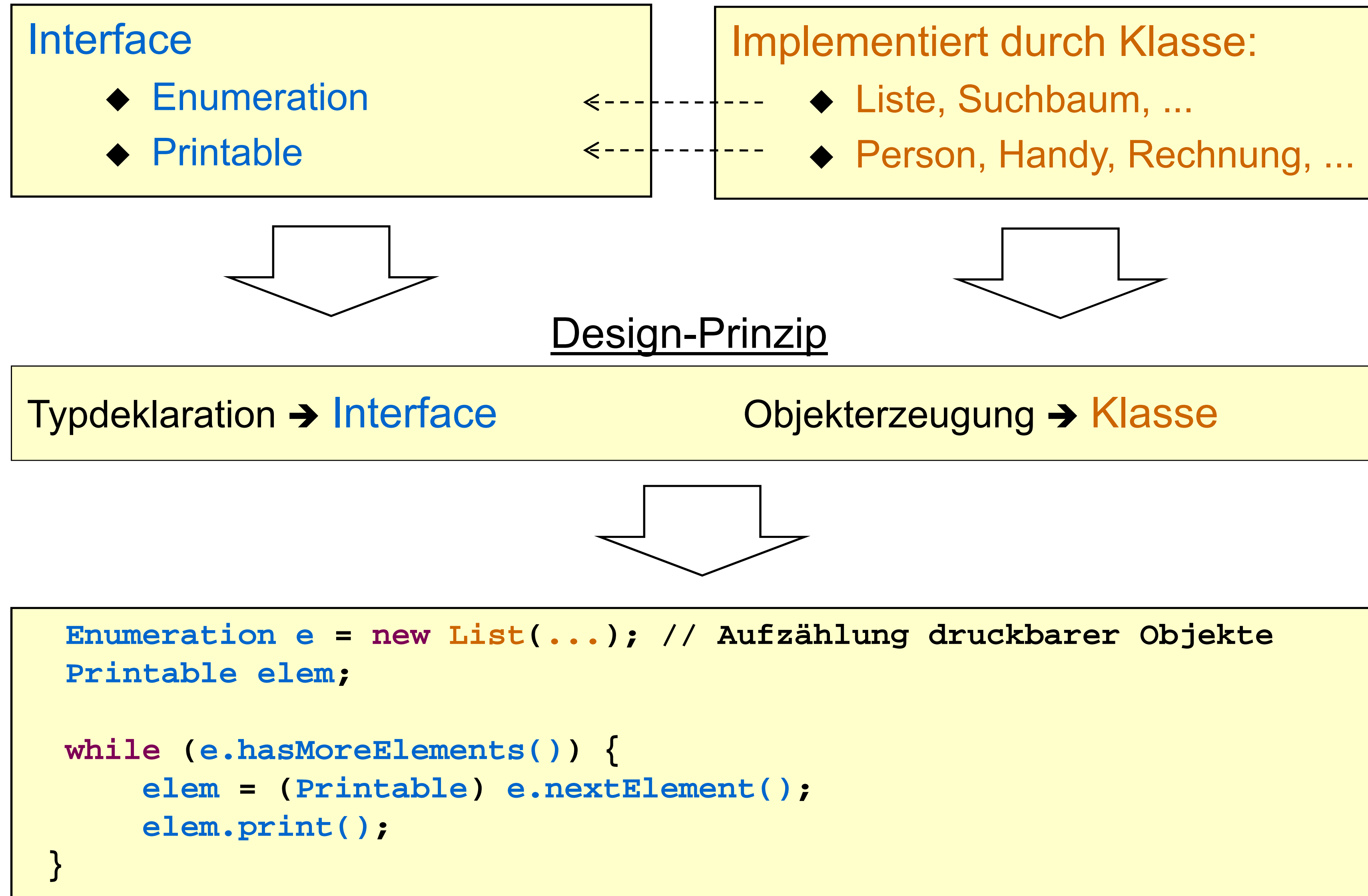
Konsequenz

- 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




```
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
```

Welche Methode soll hier aufgerufen werden? (→ `p.getX();`)

Hier wird die Ersetzbarkeit ausgenutzt (→ `p = mp;`)

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(){…}

}
```

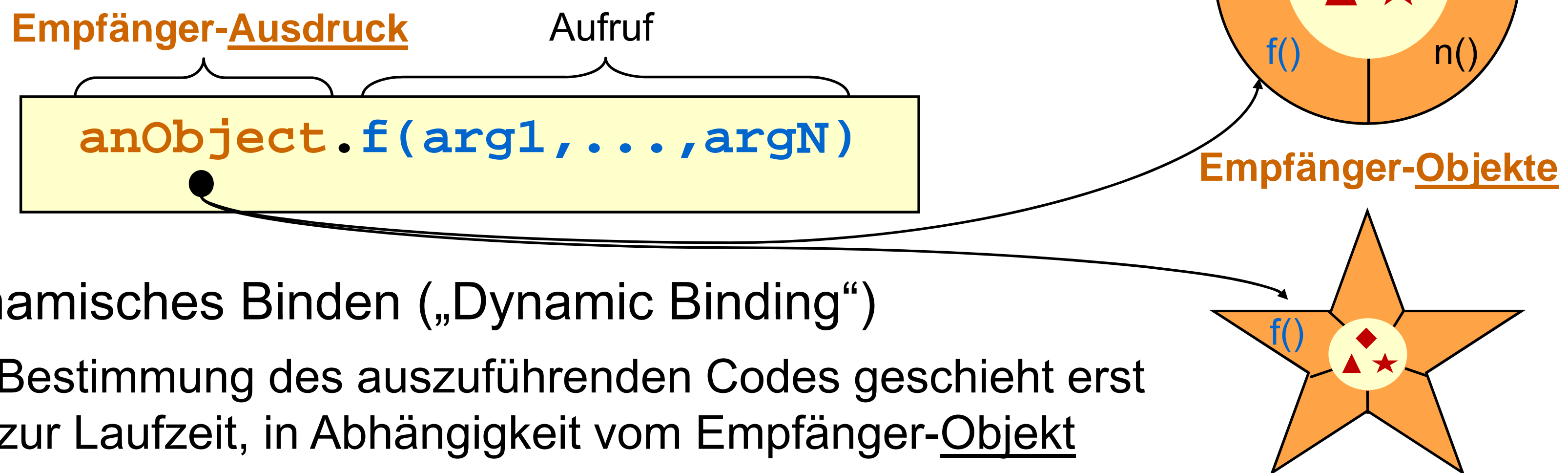
Benutzung

```
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

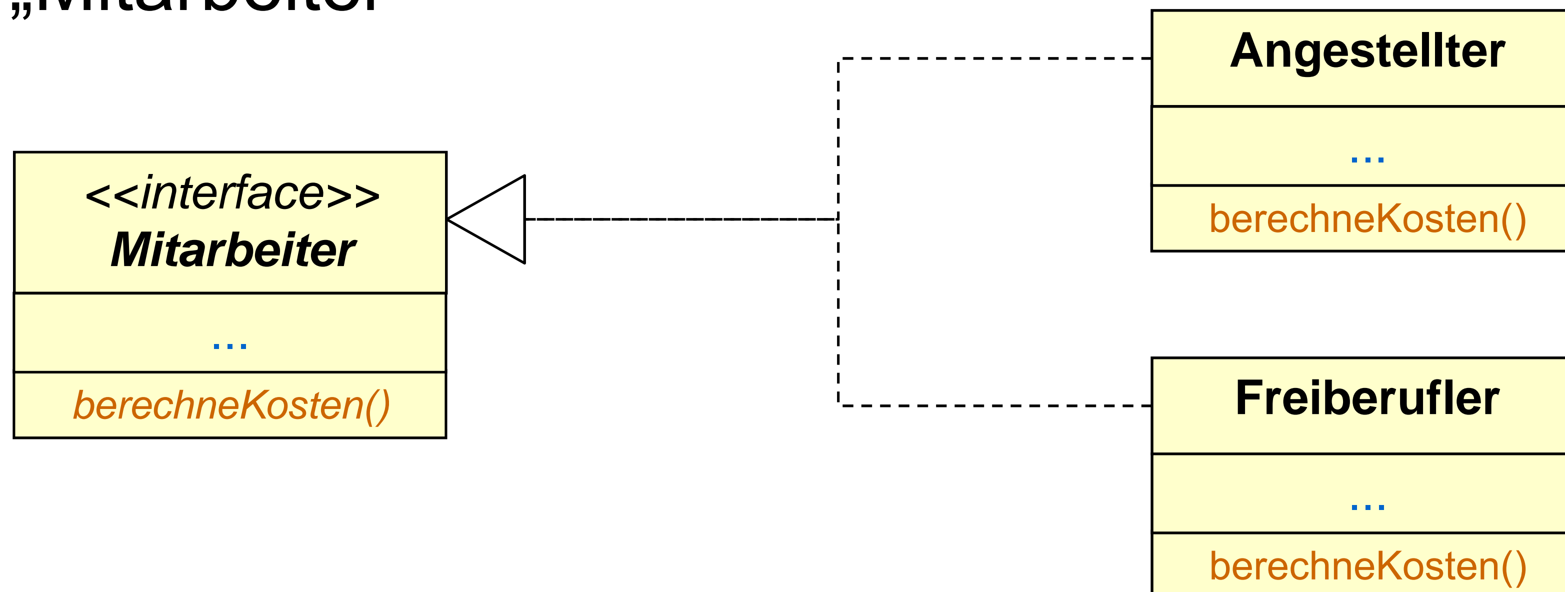
```
Point p;                 // Variablendeklaration
MyPoint mp;              // Variablendeklaration
...
p = mp;                  // ok: Point ist Obertyp von MyPoint
...
p.getX();                // ok: getX() ist in Point enthalten
```

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

Hier wird die Ersetzbarkeit ausgenutzt

Nachrichten und dynamisches Binden: Beispiel

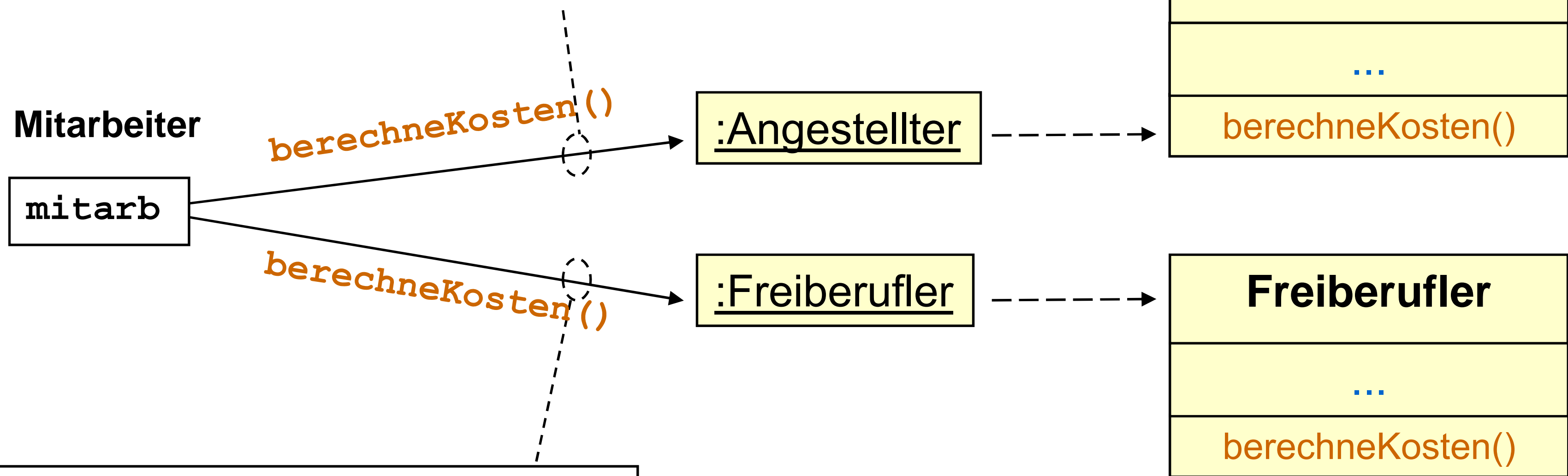
- Angenommen die Klassen „Angestellter“ und „Freiberufler“ implementieren das Interface „Mitarbeiter“



Nachrichten und dynamisches Binden: Beispiel (Fortsetzung)

```
Mitarbeiter mitarb;
mitarb = new Angestellter();
mitarb.berechneKosten();
```

Ruft Methode aus Klasse „Angestellter“ auf




```
Mitarbeiter mitarb;
mitarb = new Freiberufler();
mitarb.berechneKosten();
```

Ruft Methode aus Klasse „Freiberufler“ auf

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 Aufrufsignatur
  - Bestimmung der Kandidatensignaturen im statischen Typ des Empfängers
  - Bestimmung der spezifischsten Kandidatensignatur

  Alles wie in Folie 1-109 bis 1-118 beschrieben
- Bestimmung der aufgerufenen Implementierung
  - Dynamisches Binden

Typen und Untertypen

Typen

- Funktion
  - Typen beschränken die zulässigen Werte von Ausdrücken
- Sinn statische Typdeklaration
  - Dokumentation: Mitarbeiter mitarb;
  - Korrektheit: mitarb.wiehern(); // ☹
  - Effizienz: statische Indexbestimmung
- Problem statischer Typsysteme: Starrheit
  - Nur Werte die genau dem deklarierten Typ entsprechen sind zulässig
- Idee: Ersetzbarkeit = auch Werte zulassen, die
  - nicht genau dem deklarierten Typ entsprechen, aber
  - in jedem Kontext eingesetzt werden können, wo der deklarierte Typ erwartet wird

Ersetzbarkeit

B ist ein Untertyp von A

$:\Leftrightarrow$

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

$\Leftrightarrow$

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

Allgemeines Prinzip

$\Rightarrow$

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

Automatisch überprüfbares Kriterium

Eine Methode fordert korrekte Eingaben, d.h. korrekte Parameterwerte

Eine Methode bietet Ergebnisse eines bestimmten Typs

Ein Typ bietet eine Menge von Methoden mit einer bestimmten Signatur

Ersetzbarkeit

SHOW

B ist ein Subtyp von A

$:\Leftrightarrow$

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

$\Leftrightarrow$

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

Allgemeines Prinzip

$\Rightarrow$

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

Automatisch überprüfbares Kriterium

Eine Methode fordert korrekte Eingaben, d.h. korrekte Parameterwerte

Eine Methode bietet Ergebnisse eines bestimmten Typs

Ein Typ bietet eine Menge von Methoden mit einer bestimmten Signatur

Ersetzbarkeit

Instanzen von des Untertyps haben
(mindestens) alle Methoden des Obertyps mit
genau gleichem Namen und Parameter-Typen,
sowie einem evtl. spezifischerem Ergebnis-Typ

```
class TwoDPoint implements Point {
  private int x, y;
  public int getX(){ return x; }
  public int getY(){ return y; }


  public TwoDPoint move(Point p) {
     x += p.getX();
     y += p.getY();



     return this;
  }
}
```

```
interface Point {

   public int getX();
   public int getY();

   public Point move(Point p);
}
```

Ersetzbarkeit

SHOW

Instanzen von des Untertyps haben (mindestens) alle Methoden des Obertyps mit genau gleichem Namen und Parameter-Typen, sowie einem evtl. spezifischerem Ergebnis-Typ

```
class TwoDPoint implements Point {
  private int x, y;
  public int getX(){ return x; }
  public int getY(){ return y; }


  public TwoDPoint move(Point p) {
     x += p.getX();
     y += p.getY();



     return this;
  }
}
```

```
class ThreeDPoint implements Point {
  private int x, y, z;
  public int getX(){ return x; }
  public int getY(){ return y; }
  public int getZ(){ return z; }

  public ThreeDPoint move(Point p) {
     x += p.getX();
     y += p.getY();
     if (p instanceof ThreeDPoint) {
        z += ((ThreeDPoint) p).getZ();
     }
     return this;
  }
}
```

```
     }
}
```

Ersetzbarkeit: Nutzen

1. Die Definition von f() – blauer Hintergrund – ist anwendbar auf Arrays die Instanzen jeglicher Untertypen von Point enthalten
2. Die Erzeugung von Objekten konkreter Untertypen – hellbrauner Hintergrund –kann losgelöst von der Verwendung dieser Objekte – blauer Hintergrund – stattfinden.
   - Beide müssen lediglich die Definition von Point kennen.
3. Lediglich der objekterzeugende Teil (braun) muss konkrete Untertypen von Point kennen
4. Der blaue Teil muss keinen der Untertypen von Point kennen

Fazit

- 1 → Wiederverwendung
- 2, 3, 4 → Wartbarkeit

```
interface Point {

   public int getX();
...
}
```

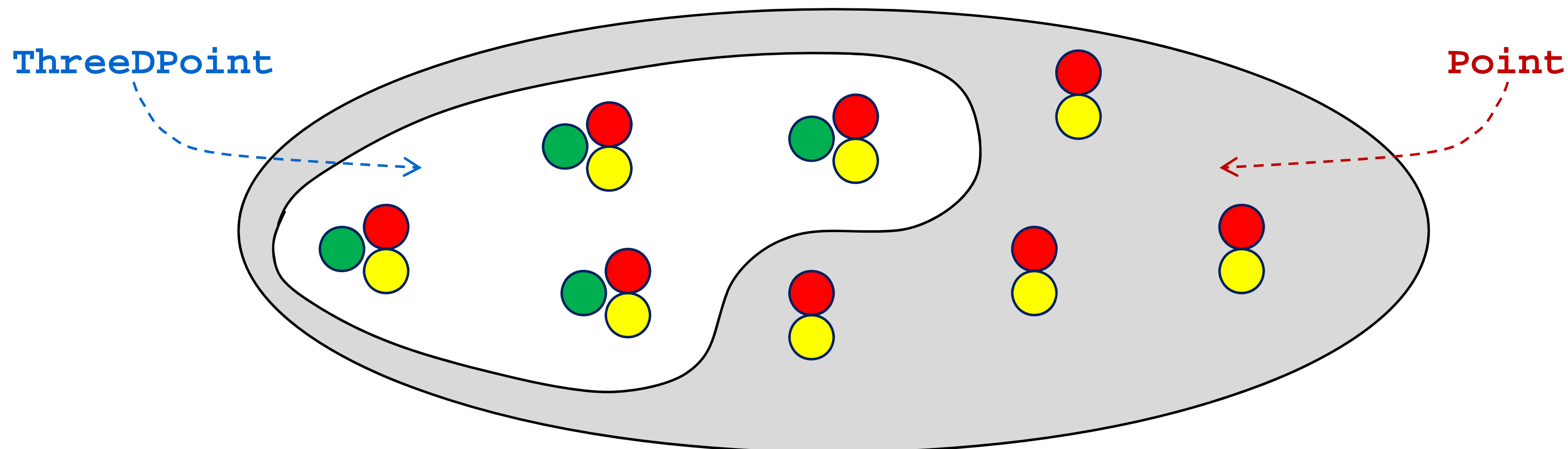
```
Point[] pa = new Array[10];
pa[0] = new TwoDPoint();
pa[1] = new ThreeDPoint();
…
f(pa);
```

```
void f( Point[] pa) {
   Point p;
   int max = pa.length-1;
   for (i=0; i<max; i++) {
      p = pa[i].move(pa[i+1]);
      p.getX();
   }
}
```

Ersetzbarkeit: Mengensicht

SHOW

Teilmengenbeziehung: Instanzen von des Untertyps sind immer auch Instanzen des Obertyps

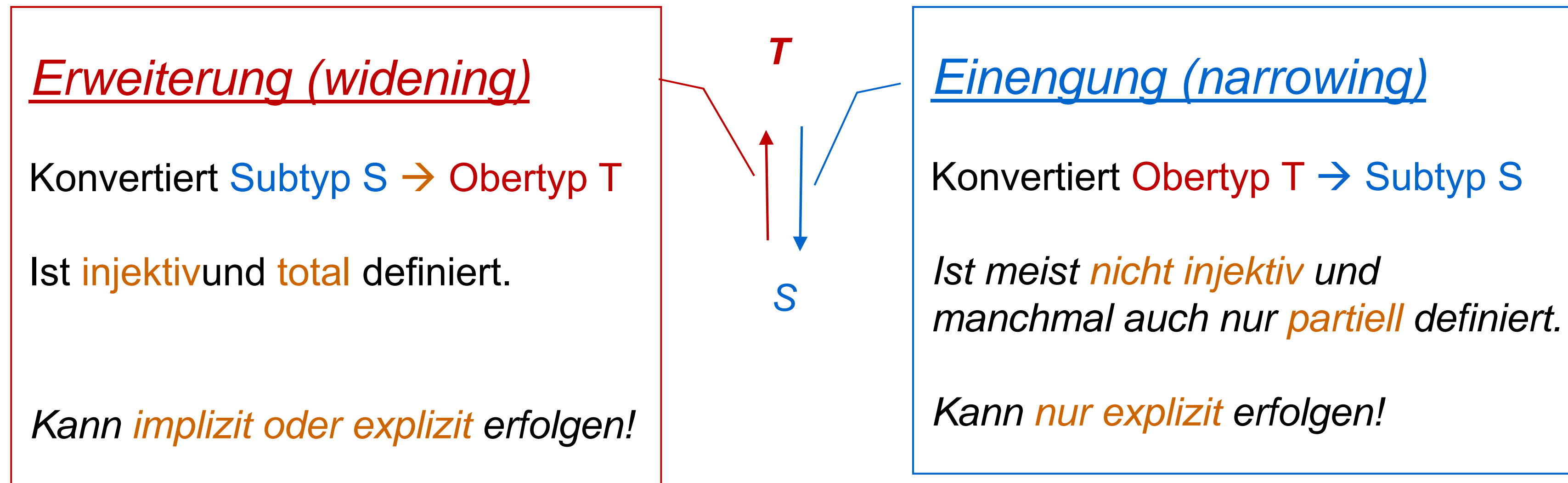


→ Je mehr Eigenschaften gefordert werden um so weniger Instanzen gibt es die all die Eigenschaften haben!

Ersetzbarkeit bei Reihungen (Arrays)

- Ein Reihungstyp ist dann ein Obertyp eines anderen Reihungstyps, wenn dies für die Typen der Komponenten gilt
- Beispiel
  - `B[]` ist Obertyp von `C[]` wenn `B` Obertyp von `C` ist
  - `B[] xb = new C[10];` ist dann eine gültige Zuweisung

Das Ersetzbarkeitsprinzip bestimmt die möglichen Typkonversionen



Implizite Typkonversion

- Implizit durchgeführt wenn ein S als ein T benutzt wird
- Führt immer nur eine Erweiterung durch

Explizite Typkonversion

- Explizite Anweisung `(T1) T2`
- Cast `(S)T` ist der einzige Weg um eine Einengung zu erzwingen

Beispiel: Explizite Typanpassung

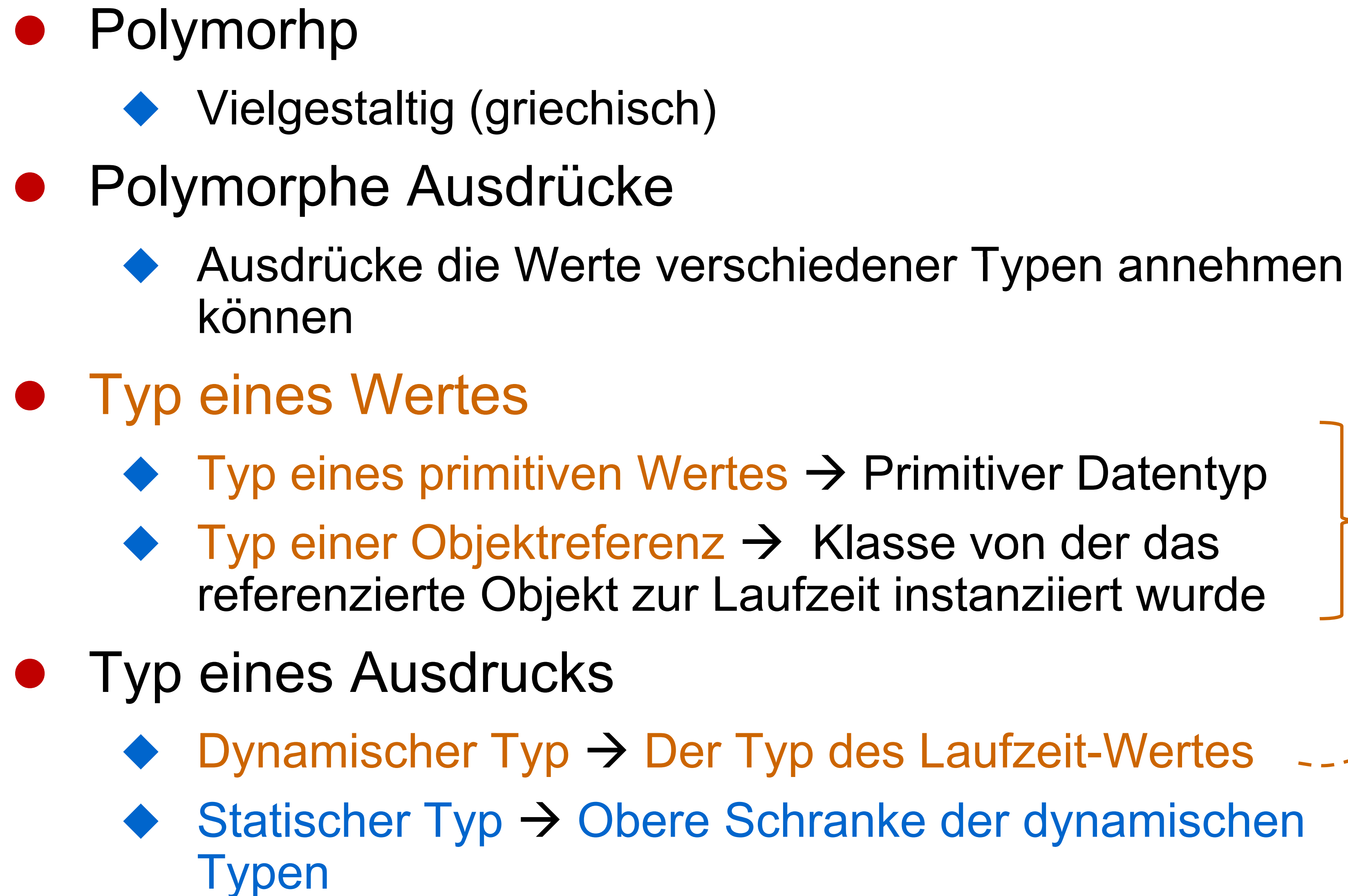
- Beispiel: Sei TNode ein Subtyp von Link der zusätzlich eine setData()-Methode enthält

```
{
    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 statisch bekannteTyp von `l` wird geändert, nicht das Objekt auf das `l` verweist!

Polymorphismus



- Polymorhp
  - Vielgestaltig (griechisch)
- Polymorphe Ausdrücke
  - Ausdrücke die Werte verschiedener Typen annehmen können
- Typ eines Wertes
  - Typ eines primitiven Wertes → Primitiver Datentyp
  - Typ einer Objektreferenz → Klasse von der das referenzierte Objekt zur Laufzeit instanziiert wurde
- Typ eines Ausdrucks
  - Dynamischer Typ → Der Typ des Laufzeit-Wertes
  - Statischer Typ → Obere Schranke der dynamischen Typen

Statischer versus dynamischer Typ

- Dynamischer Typ eines Ausdrucks
  - Der Typ des Wertes eines Ausdrucks zur Laufzeit
  - Notation: ⌊e⌋ bezeichnet den dynamischen Typ des Ausdrucks e
- Statischer Typ eines Ausdrucks
  - Der deklarierte bzw. der aus Deklarationen (ohne Datenflussanalyse) herleitbare Typ
  - Notation: ⌈e⌉ bezeichnet den statischen Typ des Ausdrucks e
- Beispiel

```
Link l = new TNode();      // ⌈new TNode()⌉ == Tnode ==  ⌊new TNode()⌋
                           // ⌈l⌉ == Link
                           // ⌊l⌋ == ⌊new TNode()⌋ == TNode

TNode n = (TNode) l;       // ⌈(TNode) l⌉ == TNode
                           // ⌊(TNode) l⌋ == ⌊l⌋ == TNode
                           // ⌈n⌉ == TNode
                           // ⌊n⌋ == ⌊(TNode) l⌋ == Tnode
```

Der Typ von `this`

- Der statische Typ von **`this`** ist jeweils die Klasse die die Methode enthält
  - In allen Methoden der 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 dies eine „Unterklasse“ der Klasse ist, die die Methode enthält
  - Vererbung & Unterklassen → nächster Abschnitt

Zusammenfassung: Polymorphismus & dynamisches Binden

- Polymorphismus
  - Ermöglicht, dass der gleiche Code Objekte verschiedener Typen bearbeitet
- Dynamisches Binden
  - Ermöglicht, dass der gleiche Code trotzdem verschiedene Effekte hat, je nach bearbeitetem Objekt
- Gemeinsamer Effekt
  - Wiederverwendung des gleichen Codes für eine unbegrenzte Anzahl von Variationen
  - Wartbarkeit!!!

Vererbung

Vererbung, Verdecken und Überschreiben

Abstrakte Basisklassen

universitätbonn

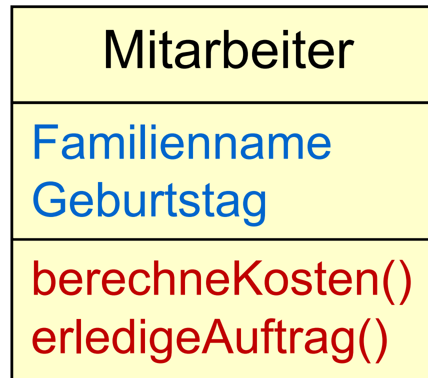
Vererbung

Mitarbeiter
Familienname Geburtstag
berechneKosten() erledigeAuftrag()

Angestellter
Familienname Geburtstag Abteilung Gehalt
versetze() berechneKosten() erledigeAuftrag()

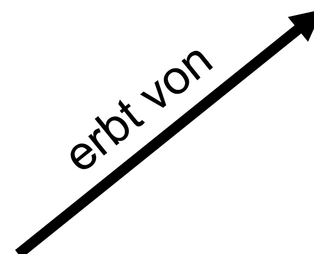
Freiberufler
Familienname Geburtstag Stundensatz
berechneKosten() erledigeAuftrag()

Vererbung

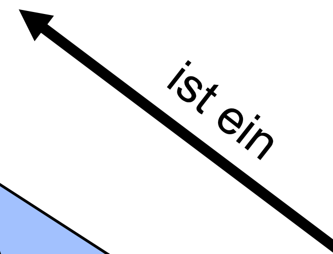


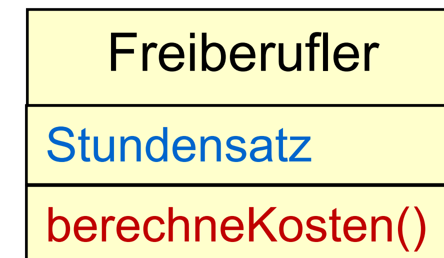
<u>Technisch</u>
Wiederverwendung von Variablen und Operationen

<u>Konzeptuell</u>
Spezialisierung bzw. Verallgemeinerung von Klassen



≠





Vererbung nur dann benutzen, wenn auch eine konzeptionelle Spezialisierung gegeben ist (nicht nur um einzelne Operationen wiederzuverwenden).

Vererbung in Java

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

- Unter-Klassen `class SubC extends SuperC { ... }`
  - `SubC` enthält implizit alle Methoden + Variablen aus `SuperC`, ...
    - ➔ Vererbung
  - ... die nicht lokal redefiniert (reimplementiert) ...
    - ➔ Overriding

> Java erlaubt beliebig viele Ober-Interfaces ("Mehrfache Subtypbeziehung")

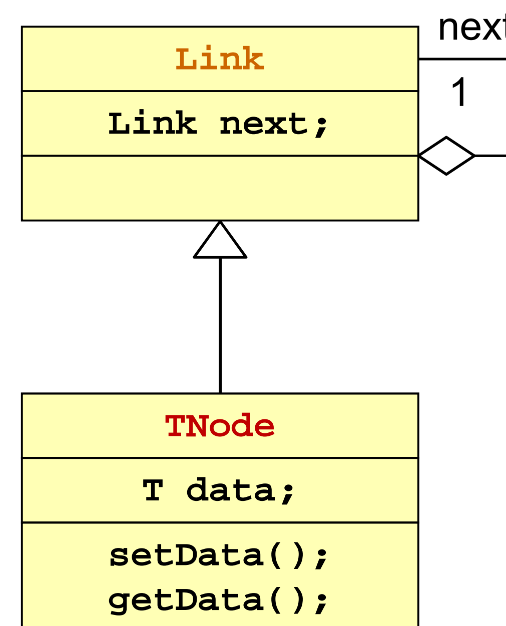
- Unter-Interfaces `interface SubI extends I1 ... In { ... }`
  - `SubI` enthält implizit alle Methodensignaturen aus `SubI, I1 ... In`
  - Mehrfaches Auftreten der gleichen Signatur in `SubI, I1 ... In` möglich
    - ➔ kein Konflikt sondern eine einzige Methodensignatur
    - ➔ Overloading bei verschiedenen Signaturen für gleichen Namen

Vererbung: Beispiel A

Java-Code

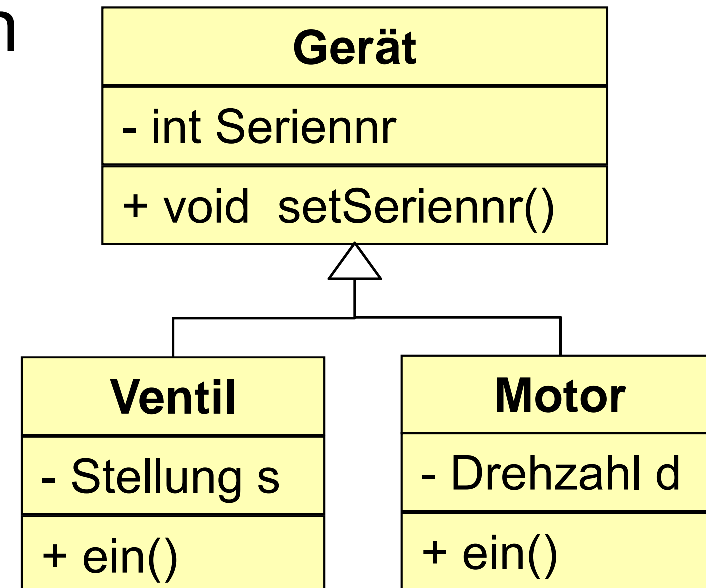
```
class Link {
    Link next;
}


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

Klassendiagramm



Vererbung und Typsystem

- Unterklassen sind Untertypen
  - Entweder sie erben
    - unveränderte Schnittstelle
  - … oder sie überschreiben
    - unverändert bis auf Ergebnis-Typ
  - … oder sie verdecken
    - unveränderte Schnittstelle
  - … oder sie fügen Neues hinzu
    - erweiterte Schnittstelle




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

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

Auflösen von Nachrichten (mit Vererbung)

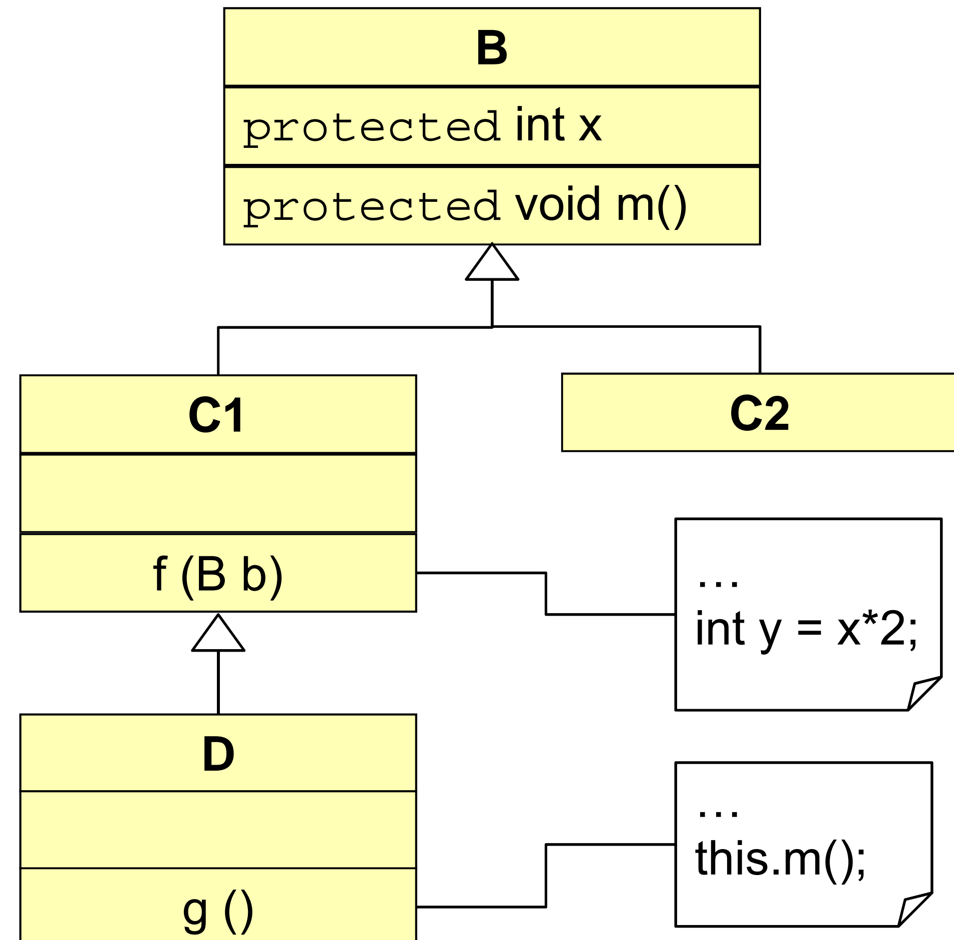
Einzige Änderung gegenüber Folie 2-61:
- Die Bestimmung der Kandidatensignaturen betrachtet auch Signaturen aus den Oberklassen
- Das dynamische Binden betrachtet auch Methoden aus den Oberklassen

Zwei Schritte

- Bestimmung der aufgerufenen Operation
  - Bestimmung der Aufrufsignatur
  - Bestimmung der Kandidatensignaturen
    im statischen Typ des Empfängers
    unter Einbeziehung der Signaturen geerbter Methoden
  - Bestimmung der spezifischsten Kandidatensignatur
- Bestimmung der aufgerufenen Implementierung
  - Dynamisches Binden
    unter Einbeziehung der geerbten Methoden

Vererbung und Zugriffkontrolle: „protected“

- Zugriffsschutz `protected` macht Mitglieder nur für Unterklassen sichtbar
- … auch für Unterklassen außerhalb des eigenen Pakets!



„Oberkonstruktoren“ und „Obermethoden“

Methoden einer Klasse können Konstruktoren und Methoden ihrer unmittelbaren Oberklasse aufrufen und auf Oberklassenfelder zugreifen

- Super-Konstruktoraufrufe
  - super() ruft den parameterlose Konstruktor der Oberklasse auf
  - super(arg) ruft einen Konstruktor mit einem Argument aus der Oberklasse auf
    - Welcher genau hängt vom statischen Typ von arg ab
- Super-Methodenaufrufe
  - super.m() ruft die Methode m() der Oberklasse auf

Super-Zugriff

- Das Schlüsselwort `super` kann als Empfängerausdruck benutzt werden, um auf Felder und 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

Beispiel: Schachclub

- Klasse Mitglied

Für Unterklassen sichtbar

```
public class Mitglied {
    protected String name;   // Mitgliedsname
    protected int nummer;    // Mitgliedsnummer

    // Konstruktor
    public Mitglied(String s, int n) {
        name = s;
        nummer = n;
    }

    // Selektoren
    public int getNumber(){
        return nummer;
    }
    public String toString() {
        return "Name: " + name + ", Nummer: " + nummer;
    }
}
```

Beispiel: Schachclub

- Unterklasse 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

Super-Zugriff: Beispiel

- Folgende 2 Beispiele tun das Gleiche:

```
class TNode {
      public T data;
      public void setData (T d) {
          data=d;
      }
}

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

```
class TNode {
      public T data;
      public void setData (T d) {
          data=d;
      }
}

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

- Welches ist wohl vorzuziehen?

Vererbung in Java: Besonderheiten

- Jedes Interface und jede Klasse erbt (implizit) von der Klasse `Object`
  - `String toString()`
  - `Object clone(Object)`
  - `boolean equals(Object)`
  - `int hashCode()`
  - `...`

- http://java.sun.com/javase/6/docs/api/java/lang/Object.html

Abstrakte Klassen

Abstrakte Klassen

Abstrakte Klassen und Interfaces

universitätbonn

Abstrakte Methoden und Klassen

In **C++** spricht man von rein virtuellen Funktionen (pure virtual functions)

- Abstrakte Methode (abstract method)
  - Hat keine Implementierung!
  - Ihre Deklaration enthält nur Modifier, Ergebnistyp, Name und Parameter → aber keinen Block!
- Abstrakte Klasse (abstract class)
  - Hat mindestens eine abstrakte Methode
  - Darf nicht nicht instanziiert werden!
    - Da sie unvollständig ist
  - Sie dient nur als statischer Objekttyp und als Oberklasse
- Java-Syntax
  - Abstrakte Methoden und Klassen müssen durch das Schlüsselwort `abstract` gekennzeichnet werden!

Abstrakte Klassen: Beispiel

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

- 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 muss auch die Basisklasse Geraet abstrakt sein.
- Geraet können wir nicht als Schnittstelle definieren, da wir das Attribut seriennummer in unserer Basisklasse definieren wollen.
  - In Java-Interfaces kann man nur Methoden und Konstanten definieren, keine Variablen.

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 final deklariert sind (= Konstanten)
- Entsprechungen
  - konkreter Datentyp → Klasse
  - abstrakter Datentyp → Abstrakte Klasse, die nur Felder und abstrakte Methoden definiert
  - „noch abstrakterer“ Datentyp → Interface

Interfaces: Beispiel

- Die Deklaration des im Paket **`java.util`** enthaltenen Interface **`Enumeration`** hat folgende 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

Vererbung: Verdecken und Überschreiben

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 Feldern

- Die Deklaration eines Feldes in einer Klasse verdeckt (hides / shadows) jedes Feld gleichen Namens in Oberklassen
  - Auch dann, wenn die Felder verschiedenen Typ haben!
  - Klassenvariablen („Klassenfelder“) können Instanzvariablen („Instanzfelder“) verdecken und umgekehrt
    - Dies geschieht nur beim Zugriff auf ein Objekt (der Zugriff auf Instanzvariablen über eine Klasse ist ja nicht möglich).

Verdecken von Methoden

- Eine Deklaration einer Klassenmethode verdeckt (hides / shadows) alle Klassenmethoden gleicher Signatur in Oberklassen
  - Instanzmethoden werden nie verdeckt sondern „überschrieben“ („overridden“)
  - Es ist nicht möglich, dass eine Klassenmethode eine Instanzmethode verbirgt oder überschreibt
  - … wohingegen eine Klassenvariable der Unterklasse eine Instanzvariable der Oberklasse verdecken darf!

Zugriff auf verdeckte Elemente

- Prinzip: Über ein Variable vom statischen Typ T kann man auf die Felder von T zugreifen
  - Der statische Typ bestimmt was „sichtbar“ ist!
- Nachfolgend die Anwendung des Prinzips für die möglichen Arten des Zugriffs auf Klassen- und Instanzelemente (Felder oder Methoden)

Zugriff auf verdeckte Klassenelemente

- Der Zugriff auf verdeckte Klassenelemente ist möglich indem man als Empfänger
  - … den qualifizierten Namen der Klasse verwendet

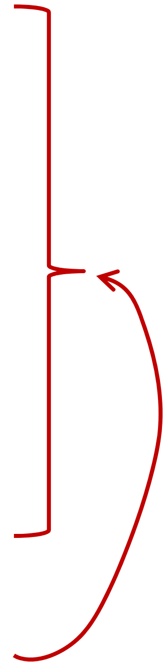
```
MyClass.var;
```

  - … oder einen Objektausdruck verwendet, dessen statischer Typ die entsprechende Klasse ist
    - Dieser wird bestimmt durch die Variablendeklaration

```
MyClass x;
x.var;
```

    - … oder durch eine explizite Typkonversion

```
T x;
((MyClass) x).var;
```



- Der Zugriff auf Klassenenlemente über Objektausdrücke gilt als schlechter Stil!

Zugriff auf verdeckte Instanzelemente

- Der Zugriff auf verdeckte Instanzvariablen der Klasse C ist möglich indem man als Empfänger eine Objektvariable verwendet, deren statischer Typ C ist
  - Dieser wird bestimmt durch die Variablendeklaration

```
C x;
x.var;
```

  - … oder durch eine explizite Typkonversion

```
T x;
((C) x).var;
```

- Instanzmethoden werden nie verdeckt sondern → „überschrieben“ („overridden“)

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

```
class T { }

class Link {
      public Link next;
      public Link prev;
}

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

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

verdeckt das Feld prev aus Link

überschreibt die Methode setData() aus TNode

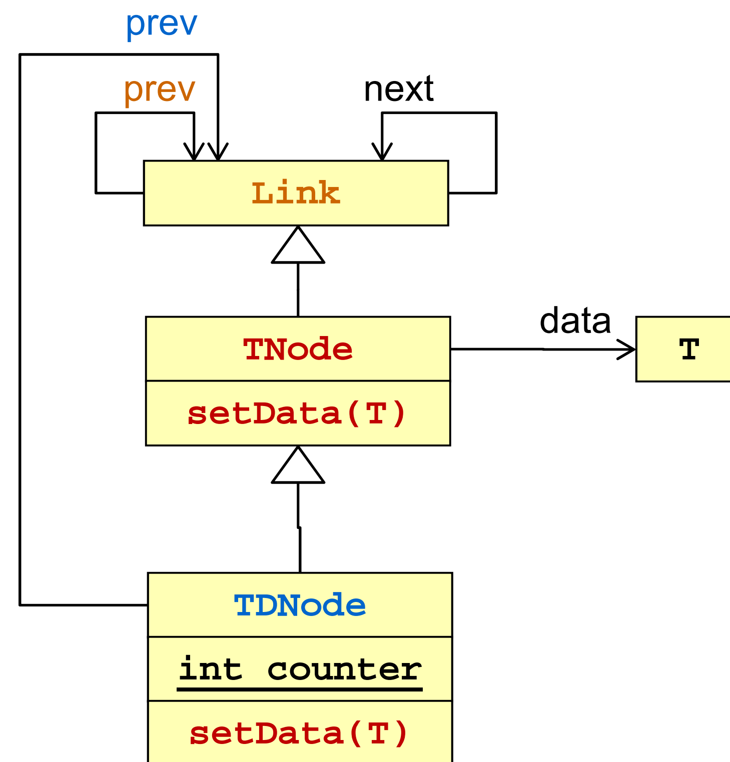
- Auf dieses Beispiel werden wir im Folgenden noch zurückkommen

Essenz des Codes als UML-Diagramm

Java Code

```
class T { }

class Link {
      public Link next;
      public Link prev;
}

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

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

UML Diagramm



Überschreiben und Verdecken: Zugriffsregeln

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

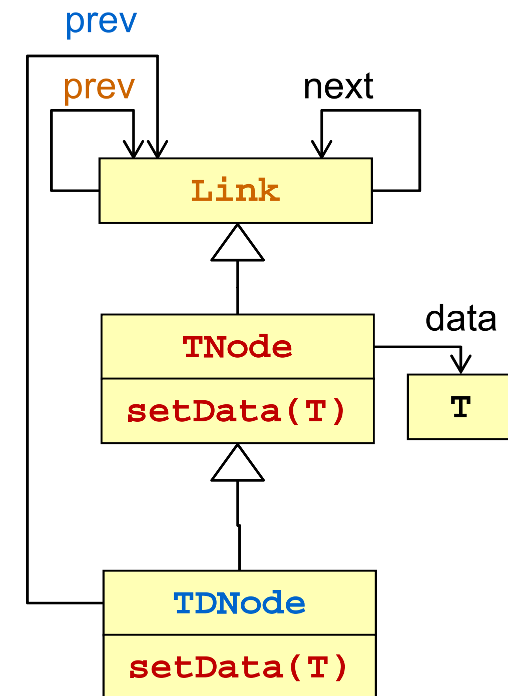
a) Ist der Zugriff für den Empfängerausdruck gültig?

- Kriterium: Was im statischen Typ des Empfänger-ausdrucks definiert ist, darf benutzt / aufgerufen werden!

b) Welches von mehreren Mitgliedern gleichen Namens in der Klassenhierarchie wird ausgewählt?

- Unterschiedliche Kriterien für Verdecken und Überschreiben

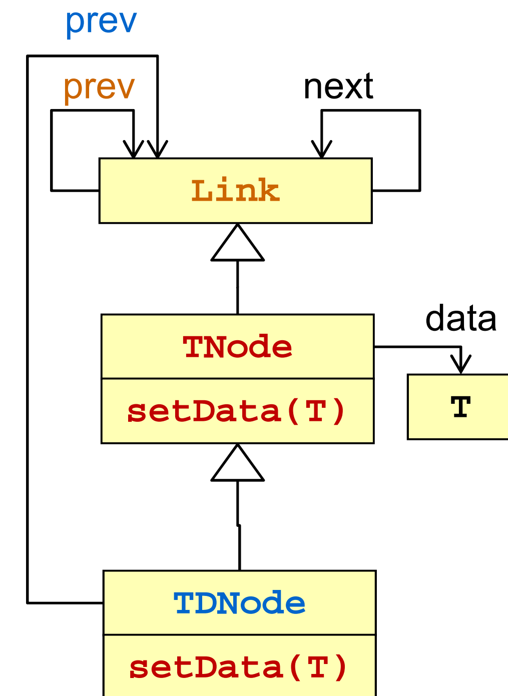
A) Gültigkeit des Zugriffs



- Ist ein Variablenzugriff oder Methodenaufruf gültig?
    - Kriterium: Was im statischen Typ des Empfängerausdrucks definiert ist, darf benutzt / aufgerufen werden!
    - Der statische Typ ist der deklarierte Typ oder der vom Compiler aus Deklarationen ohne Datenflussanalyse herleitbare Typ
- Beispiel

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

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

B) Auswahl des zugegriffenen Mitglieds



- Welches von mehreren Mitgliedern gleichen Namens in der Klassenhierarchie wird ausgewählt?
  - Verdecken
    - Der statische Empfängertyp bestimmt die sichtbare Signatur und Implementierung → statisches Binden
  - Überschreiben
    - Der statische Empfängertyp bestimmt die sichtbare Signatur
    - Der dynamische Empfängertyp bestimmt die Implementierung → dynamisches Binden

```
Link l = new TDNode();
l.prev;                  // Verdecken: Feld aus Link

TNode n = new TDNode();
n.prev;                  // Verdecken: Feld aus Link
(TDNode) n).prev;        // Verdecken: Feld aus TDNode
n.setData();             // Überschreiben: Methode aus TDNode
```

Überschreiben und Verdecken : Beispiel

```
class T { }

class Link {
     Link next;
     Link prev;
}

class TNode extends L
     T data;
     void setData (T d
          data=d;
     }
}

class TDNode extends
     Link prev;
     static int counte
     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 Signatur im statischen Typ des Empfängers definiert und sichtbar sind
2. Auf verdeckte Felder und Methoden kann man über Referenzvariablen vom passenden statischen Typ zugreifen
3. Instanzmethoden überschreiben („overriding“), Klassenmethoden verdecken („shadowing“)
4. Es werden immer die dem tatsächlichen Objekt zugehörigen Varianten von Instanzmethoden auf diesem ausgeführt (dynamisches Binden)
5. Statische und finale Methoden dürfen nicht überschrieben werden

Vererbung: “finale” Methoden und Klassen

Abstrakte Methoden

Finale Methoden

Überschreibbare versus 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 → Finale Methoden
  - Weil man sich aus Sicherheitsgründen darauf verlassen muss, dass die Implementierung nicht ersetzt werden kann
  - Der Aufruf kann auch etwas effizienter erfolgen

Finale Methoden

- Syntax
  - Finale Methoden (final methods) sind mit dem Schlüsselwort `final` gekennzeichnete Instanzmethoden
  - Klassenmethoden sind in Java per Definition nicht überschreibbar (da statisch gebunden)
- Semantik
  - Finale Methoden werden an Unterklassen vererbt
  - Sie dürfen in den Unterklassen nicht überschrieben werden
  - Die mit `final` gekennzeichnete Implementierung ist von da abwärts die letzte Implementierung der Methode in der Klassenhierarchie

Finale Klassen

- Eine finale Klasse (final class) ist eine die keine Unterklassen haben darf
  - Dies wird durch Angabe des Schlüsselworts `final` deklariert (`final class C … {…}`)
  - Alle Methoden einer finalen Klasse sind implizit final
  - Beispiel: Die vordefinierte Klasse Array ist final → Von Reihungen darf man nicht ableiten
- Eine abstrakte Klasse darf nicht final sein
  - Da ihre Implementierung sonst nie ergänzt werden könnte!

Überschreibbare und Finale Methoden in C++

- In C++ sind alle Instanzmethoden implizit final
  - Statisch gebunden und nicht überschreibbar
- Virtuelle Funktionen müssen in C++ durch das Schlüsselwort `virtual` gekennzeichnet werden
  - Dynamisch gebunden und überschreibbar
- Instanzmethoden in C++, die nicht als virtuelle Funktionen gekennzeichnet (also implizit final) sind können verdeckt werden, während dies bei als final gekennzeichneten Methoden in Java nicht möglich ist

universitätbonn

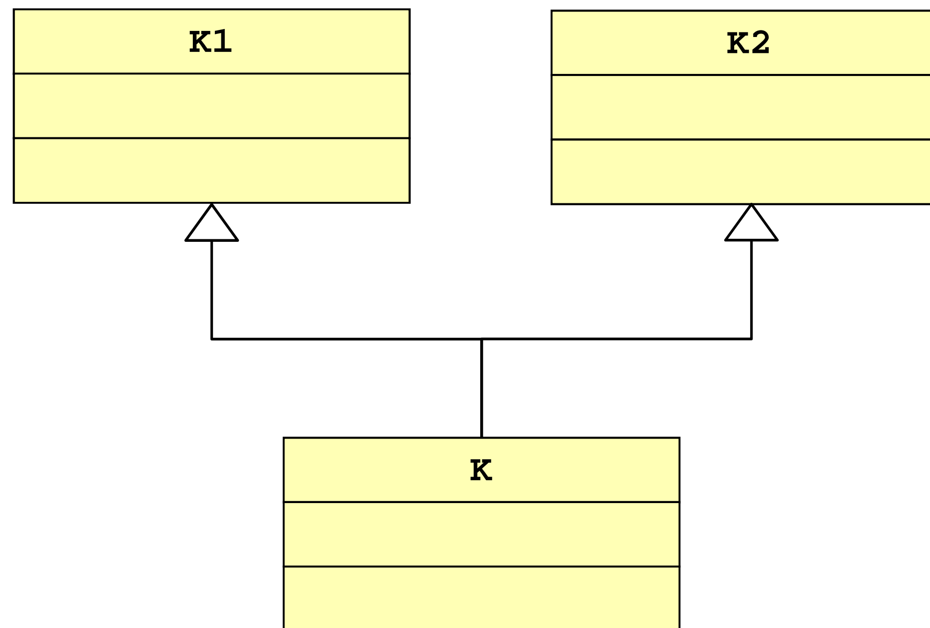
Multiple Vererbung

Definition

Probleme

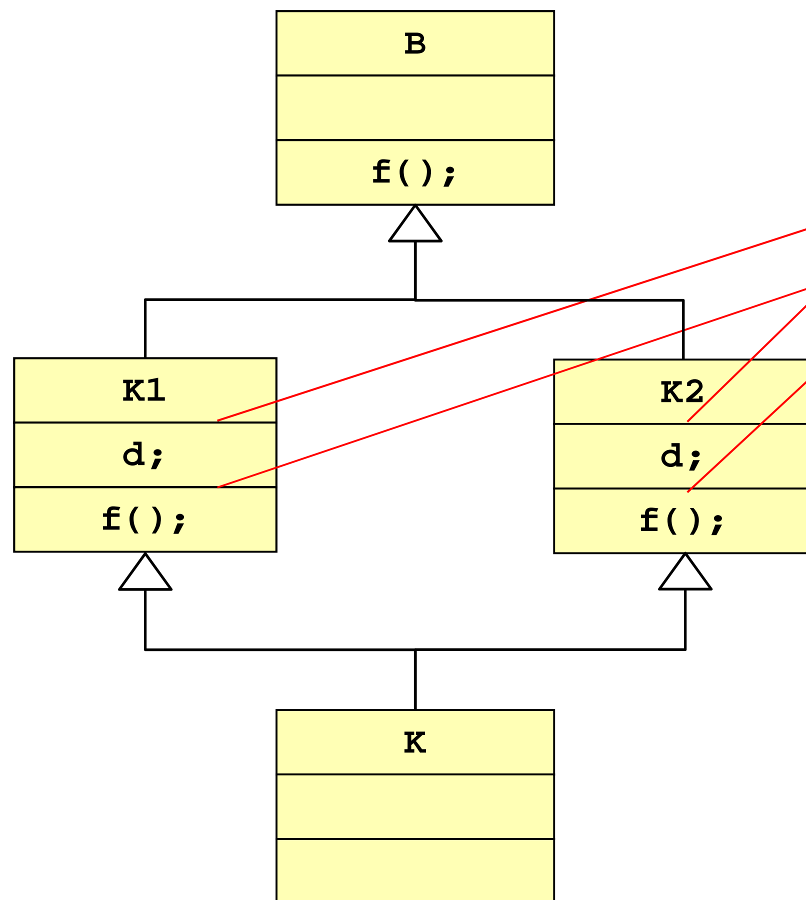
Mehrfachvererbung

- 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 zwischen Klassen nur Einfachvererbung
- Diese Einschränkung verhindert Konflikte zwischen
  - nicht-konstanten Attributen oder
  - lokal nicht überschriebenen Methoden

  mit gleichem Namen / gleicher Signatur, die von mehreren Oberklassen geerbt werden

- Java erlaubt „Mehrfachvererbung“ zwischen Interfaces
  - Dies ist keine echte Vererbung sondern nur eine Subtypbeziehung, da keinerlei Implementierung „geerbt“ wird (lediglich die Typinformation der Methodensignaturen)

Zusammenfassung

Checkliste für Prüfunsvorbereitung

Chekliste: Können Sie folgende Konzepte erklären?

- Objekte
  - Kapselung
  - Objektidentität
  - Aliasing / Sharing
  - Flaches / tiefes Clonen
- Typen
  - Primitive Typen / Obejkttypen
  - Subtypbeziehung
  - Ersetzbarkeitsprinzip
  - Polymorphismus
  - Statischer / Dynamischer Typ
- Klassen
  - Klassenvariablen versus Instanzvariablen
  - Klassenmethoden versus Instanzmethoden
  - Konstruktoren
  - Selbstbezug (this)
- Vererbung
  - Vererbung versus Spezialisierung
  - Vererbung versus Subtypbeziehung
  - Finale Klassen und Methoden
  - Abstrakte Klassen und Methoden
  - Interfaces versus abstrakte Klassen versus konkrete Klassen versus abstrakte Datentypen
- Zugriff auf Felder und Methoden
  - Sichtbarkeiten
  - Aufrufsignatur
  - Kandidatensignaturen
  - Rolle des statischen Empfängertyps
  - Signatursubtyp
  - Statisches / dynamisches Binden
  - Verdecken / Überladen / Überschreiben
  - Prozeduraufrufe versus Nachrichten