

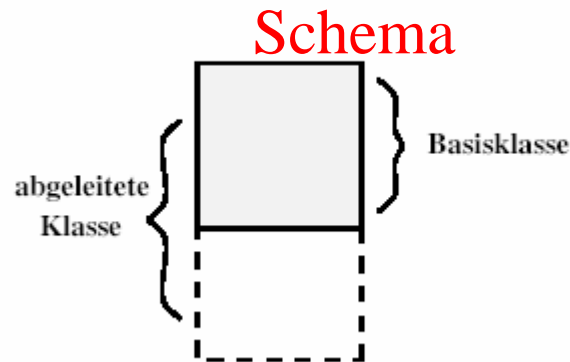
Höhere objektorientierte Konzepte

Vererbung, dynamisches Binden, abstrakte
Basisklassen, Interfaces, generisches
Programmieren, Aktionsobjekte

Foliensatz von A. Weber zur Vorlesung Informatik I, Bonn, 2002/03
Überarbeitet von W. Küchlin zu Informatik I, Tübingen 2003/04

Vererbung und abgeleitete Klassen

- **Vererbung** (*inheritance*) ist ein Mechanismus, mit dem die „**ist-ein**“ (*is-a*)-**Beziehung** zwischen **Datentypen** ausgenutzt werden kann
- Sei der **Datentyp A** ein **Untertyp** (*subtype*) eines **Obertyps** (*supertype*) **B**
 - Jedes **a** aus **A** ist (auch) ein **B**
 - Die Klasse **A** ist eine **Erweiterung** der Klasse **B**
 - Die Klasse **A** hat **alle Eigenschaften** von **B** und **eventuell** noch **weitere zusätzliche**
- Methoden von **B** können in **A** wieder verwendet werden
 - **Reale Vererbung**: **A** verwendet tatsächlich die **Implementierung** einer Methode aus **B** wieder
 - **Virtuelle Vererbung**: **A** verwendet die **Methoden-Schnittstelle** wieder, überschreibt aber die **Implementierung**. Dies gibt mehr Flexibilität

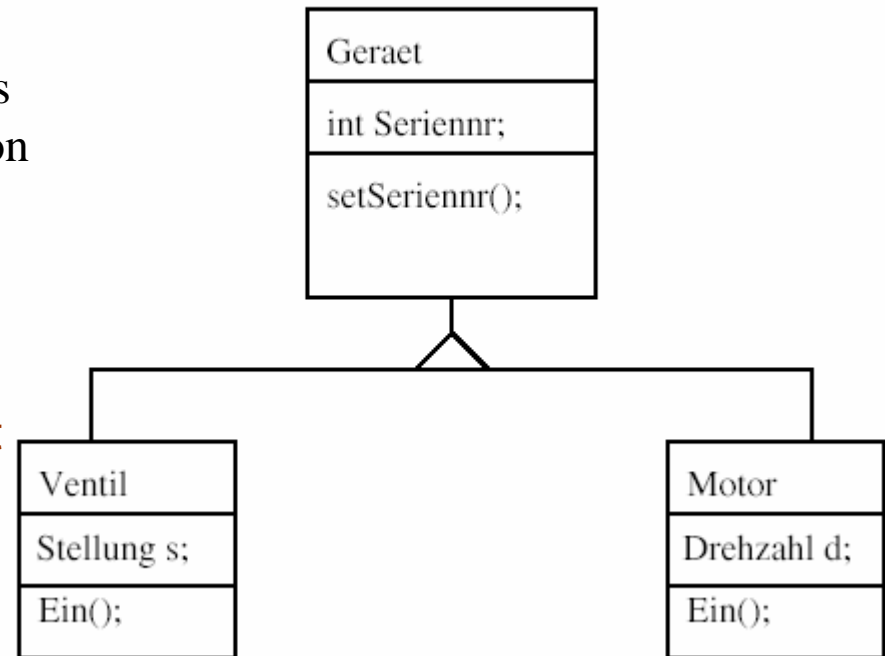


Vererbung und abgeleitete Klassen

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

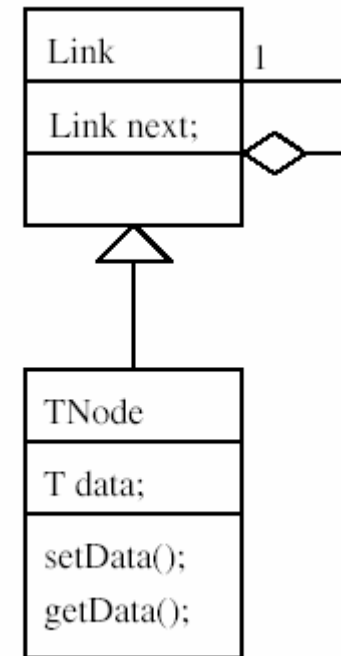


Klassendiagramm

Vererbung und abgeleitete Klassen

- **Beispiel B:** Extraktion von Gemeinsamkeiten aus TNode

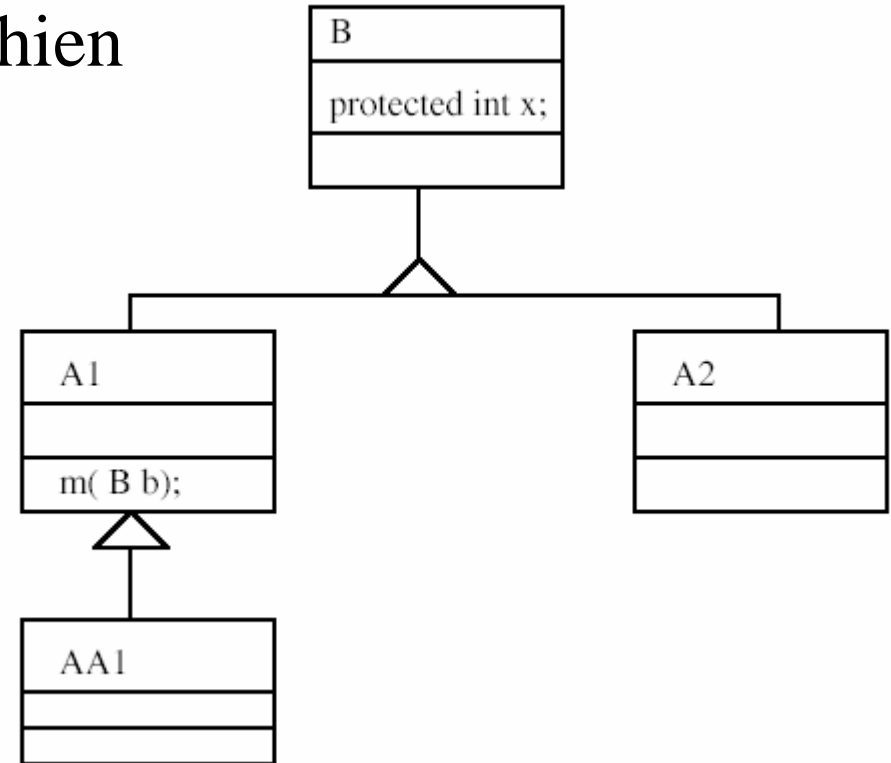
```
class Link {  
    Link next;  
}  
class TNode extends Link {  
    T data;  
    public void setData(T d){  
        data = d;  
    }  
    public T getData() {  
        return data;  
    }  
}
```



Klassendiagramm

Zugriffsschutz **protected** in Klassenhierarchien

- Zugriffsschutz **protected** hat Bedeutung in Klassenhierarchien
- Zugriff nur von abgeleiteten Klassen aus (In **Java**: innerhalb des Pakets sowieso voller Zugriff)
 - **m** kann auf **x** (= **this.x**) zugreifen
 - Zugriff **b.x** in **m** nur gültig, wenn aktueller Typ von **b** mindestens **A1**, also nur durch **((A1) b).x** oder **((AA1) b).x**
 - Zugriff **ungültig**, falls **b** vom Typ **A2**, oder nur genau vom Typ **B**



Konstruktoren in Klassen-Hierarchien

- In einer abgeleiteten Klasse **A** wird der **no-arg Konstruktor** der **Basisklasse B** mit **super()** bezeichnet
 - Ein Konstruktor mit einem Argument **arg** als **super(arg)** usf.
- Eine **Methode** der übergeordneten Klasse kann von der abgeleiteten Klasse aus mithilfe des Schlüsselworts **super** aufgerufen werden
 - wie z. B. in **super.m()**
 - Explizite Konstruktor-Aufrufe wie **super(arg)** oder **this(arg)**, immer nur als erste Anweisung im Konstruktor möglich.

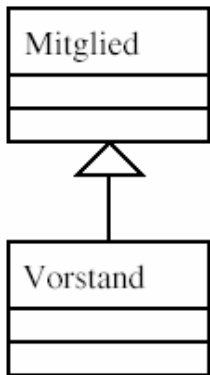
Beispiel: Hierarchie in einem Schachclub

- Basisklasse

```
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 getName() { return name;}
    // Methoden
    public String toString() {
        return "Name: " + name + ", Nummer: " + nummer;
    }
}
```

Vererbung und abgeleitete Klassen

- Abgeleitete Klasse **Vorstand**



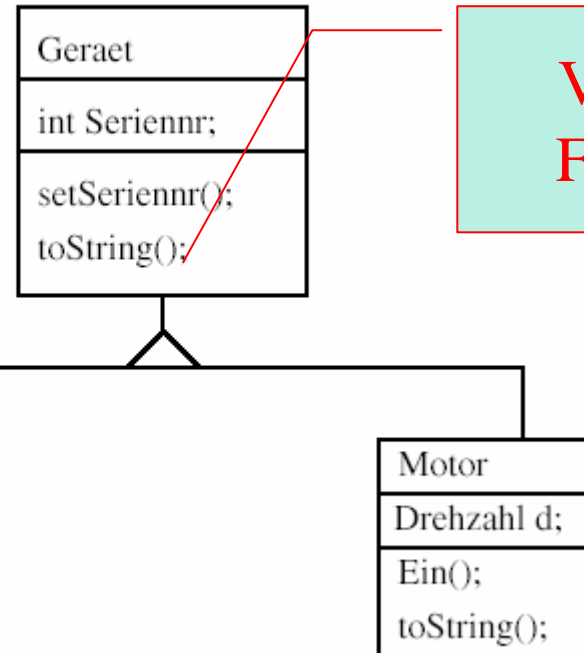
```
public class Vorstand extends Mitglied
{
    protected String amt; // Präsident, Kassenwart, ...
    // Konstruktoren
    public Vorstand(String n, int m, String a) {
        super(n, m); amt=a;
    }
    public String toString() {
        return ("Vorstandsmitglied: " + super.toString()
       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

**Konstruktor-
Aufruf der
Oberklasse**

**Aufruf einer
Methode der
Oberklasse**

Virtuelle Funktionen und dynamisches Binden

```
Ventil v = new Ventil();  
String s = v.toString();  
    // ein Ventil ist ein Gerät
```



Virtuelle
Funktion

Reimplementierung in
abgeleiteter Klasse
(überschreibt [überlagert]
Implementierung in
Basisklasse mit gleicher
Signatur)

Virtuelle Funktionen und dynamisches Binden

- Wird in **A** eine Methode **vm()** überschrieben, so ändert sich damit die Bedeutung eines *jeden* Aufrufs von **vm()**, der auf einem Objekt vom (genauen) Typ **A** stattfindet
 - I. a. **ändern** auch andere **Methoden** in **A** durch das Überschreiben von **vm()** implizit **ihre Bedeutung** (Semantik), falls sie Aufrufe von **vm()** enthalten
 - Gilt auch für solche Methoden, die **A** völlig unverändert von **B** geerbt hat und die evtl. schon implementiert wurden lange bevor das neue **vm()** geschrieben wurde
 - Ein **solcher Aufruf** ist **immer problematisch**, da er **nicht** in der **Methodenschnittstelle sichtbar** ist
 - *Fragile Base Class Problem*

Virtuelle Funktionen und dynamisches Binden

- Da Objekte **polymorph** sein können, kann ein Objekt vom **tatsächlichen abgeleiteten Typ A** auch als solches vom **Basistyp B** fungieren
- Eine **Objektreferenz xb** vom **Typ B** kann auf **Objekte beider Typen** zeigen
- Im Falle einer **virtuellen Methode vm()** hat ein **Aufruf xb.vm()** je nachdem eine **andere Bedeutung**
 - Es wird immer der Code ausgeführt, der dem **tatsächlichen Typ** des **Objektes** entspricht, auf das **xb** verweist
 - Zeigt **xb** momentan auf ein Objekt vom Typ **A**, so wird diejenige Variante von **vm()** ausgeführt, mit der **vm()** in **A** überschrieben wurde

Virtuelle Funktionen und dynamisches Binden

- Die **Auswahl** der **tatsächlich** ausgeführten Funktion **xb.vm()** erfolgt erst zur **Laufzeit** des Programms *dynamisch* anhand des *aktuellen Typs* des an **xb** zugewiesenen Objekts
 - und **nicht** schon zur **Übersetzungszeit** *statisch* anhand des *deklarierten Typs* von **xb**
 - der ein **Basistyp** (Obertyp) des *aktuellen Typs* sein kann
- Anders formuliert wird der **Code** von **vm()** **nicht** zur **Übersetzungszeit** (*statisch*) an den Namen **vm** gebunden, **sondern erst** zur **Laufzeit** (*dynamisch*)
 - Deshalb spricht man auch von **dynamischem Binden** oder **spätem Binden** (*dynamic binding, late binding*)

Virtuelle Funktionen und dynamisches Binden: Beispiel

- **Auswahl der Methode zur Laufzeit**

```
Mitglied[] member = new Mitglied[3];
member[0] = new Mitglied("Udo", 1245);
member[1] = new Mitglied("Klaus", 1246);
member[2] = new Vorstand("Eva", 720, "Präsidentin");

for (int i=0; i<member.length; i++) {
    System.out.println(member[i].toString());
}
```

- Welcher Code ausgeführt wird, kann erst zur Laufzeit bestimmt werden
- Bei jedem Aufruf kann das verschiedener Code sein

- Virtuelle 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 Basisklasse kann **nur als Ableitungsbasis** und **nicht als Typ** von Objekten auftreten, 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*)

Virtuelle Funktionen und Finale Methoden

- In Java ist jede Instanzmethode, die nicht weiter gekennzeichnet ist, eine potentiell überschreibbare Funktion
- Es gibt aber auch Instanzmethoden, die nicht überschreibbar sein sollen, z.B. aus Sicherheitsgründen
 - Zugriff kann auch etwas effizienter erfolgen, siehe später
- Solche 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

Virtuelle Funktionen und Finale Methoden

- **Bemerkung:**
 - In C++ dagegen sind alle Instanzmethoden **implizit final**
 - Da Aufrufe finaler Methoden ohne zusätzlichen Aufwand gegenüber Funktionen in C geschehen können
 - Siehe später
 - 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**

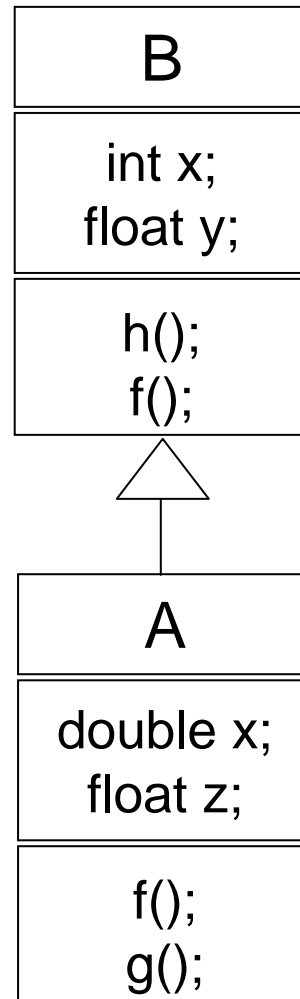
Virtuelle Funktionen und Finale Methoden

- Soll von einer Klasse **F** insgesamt **nicht** mehr **abgeleitet** werden können, so deklariert man **F** durch Angabe des **Schlüsselworts** **final** als **finale Klasse** (*final class*)
 - Die vorliegende Implementierung von **F** ist damit endgültig
 - Es wird kein Objekt vom Typ **F** geben, das einige Methoden anders definiert oder zusätzliche Attribute oder Methoden hat
 - Alle **Methoden** einer **finalen Klasse** sind damit **implizit** schon **final**
 - Eine **abstrakte Klasse** kann **nicht final** sein, da ihre Implementierung sonst nie ergänzt werden könnte

Realisierung des dynamischen Bindens

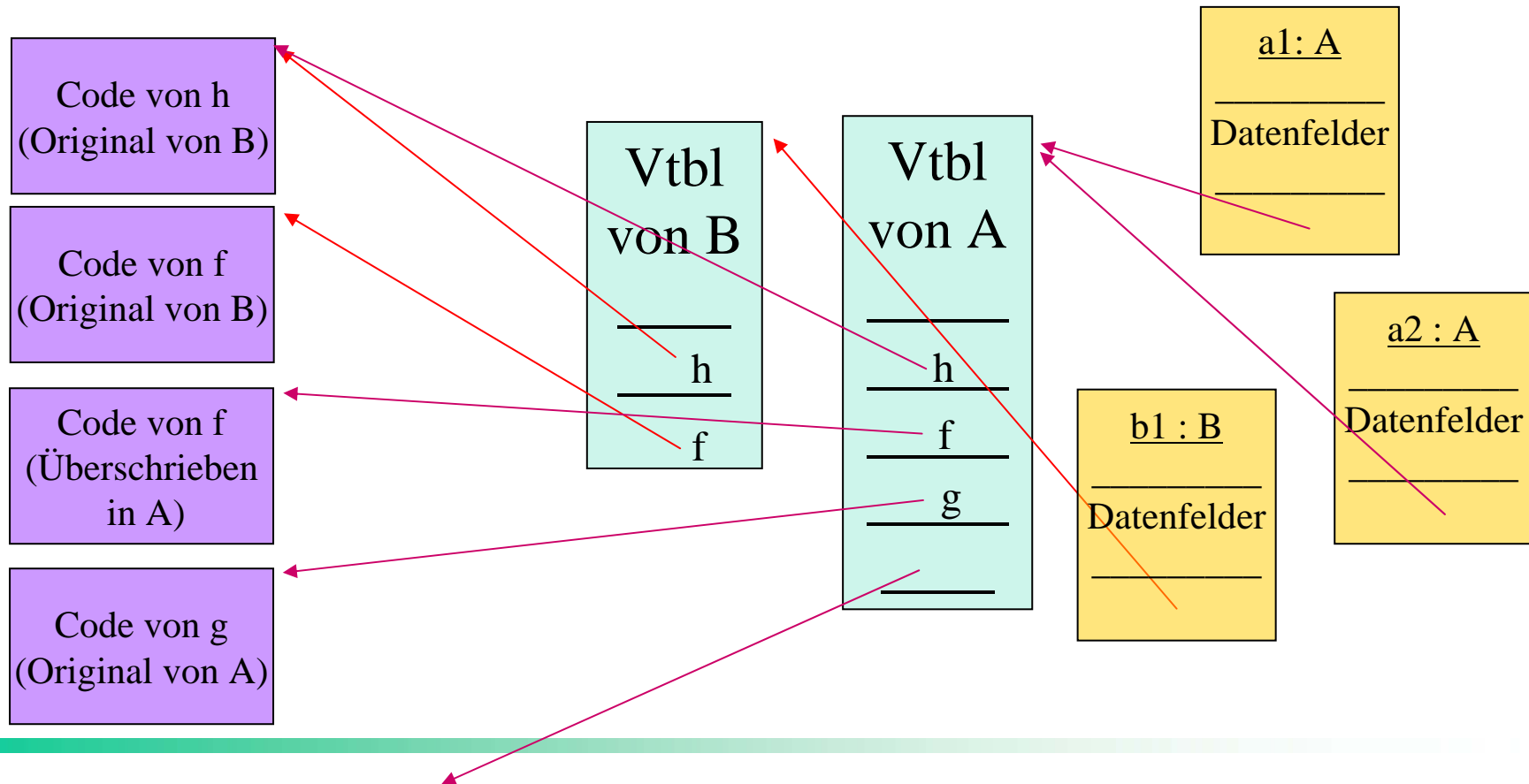
- Eine Funktion **f** wird durch einen **Funktionszeiger** (*function pointer*) repräsentiert
 - Darunter kann man sich die Anfangsadresse des Funktionscodes im Speicher vorstellen, die die Funktion ja eindeutig repräsentiert
- Die Funktionszeiger für **finale** und **statische Methoden** (und Funktionen in **C**) können **direkt** in den aufrufenden Instruktionen vorkommen
- Die Funktionszeiger für **virtuelle Funktionen** werden über eine **Tabelle** angesprochen. Dort können sie transparent ausgetauscht werden.

- Beispiel:



Realisierung des dynamischen Bindens

- Bei **virtuellen Funktionen** gibt es **Indirektion** über **Tabelle der virtuellen Funktionen** (*virtual function table*; *vtbl*)



Typanpassungen in Klassenhierarchien

- Ähnlich wie bei Elementartypen gibt es auch für Variable von Klassentypen die Möglichkeit der **Typanpassung** oder **Typerzwingung** (*type coercion*)
- Jedes Objekt einer abgeleiteten (Unter-)Klasse **A** kann auch als Objekt jeder Oberklasse **B** angesehen werden (Polymorphie)
 - Deswegen ist die **Zuweisung**
`B b = new A();`
uneingeschränkt gültig
- Umgekehrt gilt dies natürlich nicht, denn nicht jedes Objekt der Klasse **B** ist auch ein Objekt der Klasse **A**
 - **Nicht jedes Gerät ist ein Ventil**

Bedeutung der Typerzwingung a priori nicht klar; allgemein verwendete „Umwandlungsfunktion“: **Identität auf Objekt** („kann angesehen werden als“)!
Nur **Änderung des deklarierten Typs!** (D.h. **Referenzvariable vom Obertyp kann auf Objekt vom Subtyp verweisen.**)

Typanpassungen in Klassenhierarchien

- Hat man z. B. eine **gemischte Liste** von Objekten der Klassen **B** und **A**, so benutzt man zum Listendurchlauf eine Referenzvariable **xb** vom Typ **B**, denn jedes Objekt kann an **xb** zugewiesen werden
- Manchmal will man sich aber nicht auf Methodenaufrufe des Kontraktes von **B** beschränken, sondern man will spezielle Methoden, die nur in **A** nicht aber in **B** zur Verfügung stehen, aufrufen, falls **xb** tatsächlich auf ein Objekt der Klasse **A** zeigt
- In diesem Fall kann man durch eine **Typanpassung** (*type cast*) den Typ von **xb** zu **A** **verengen** (*narrowing*)
 - Die **Java-Syntax** ist dabei **dieselbe** wie für die **Typanpassung** bei **Elementartypen**

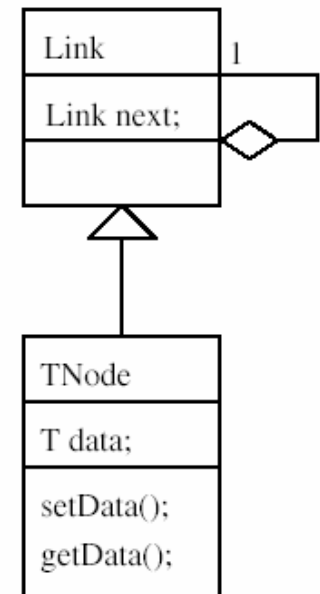
Typanpassungen in Klassenhierarchien

- **Beispiel:**

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

Type cast
Operator

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



Typanpassungen in Klassenhierarchien

- Die Typanpassung „nach oben“ in der Klassenhierarchie ist also immer korrekt
 - Wir sprechen auch von **Ausweitung** (*widening*), **Aufwärtsanpassung** (*up casting*) oder **sicherer Anpassung** (*safe casting*)
- Die Typanpassung einer Referenzvariable „nach unten“ in der Klassenhierarchie ist nur dann korrekt, wenn die Variable auf ein Objekt der entsprechenden Klasse (oder einer Unterklasse davon) zeigt
 - Wir sprechen auch von **Verengung** (*narrowing*), **Abwärtsanpassung** (*down casting*) oder **unsicherer Anpassung** (*unsafe casting*)
 - Falls eine unzulässige Verengung versucht wird, wird eine (ungeprüfte) Ausnahme vom Typ **ClassCastException** ausgeworfen
 - Dies kann durch eine vorherige Prüfung
`if (xb instanceof A) ((A) xb).method();`
mit dem Operator **instanceof** verhindert werden

- **Bemerkung:**
 - Von Reihungen kann man nicht ableiten
 - Ein Reihungstyp ist dann ein Obertyp eines anderen Reihungstyps, wenn dies für die Typen der Komponenten gilt
 - Ist **B** Oberklasse von **A**, dann ist **B[]** Oberklasse von **A[]**,
und **B[] xb = new A[n];** ist eine gültige Zuweisung

Überschreiben und Verdecken

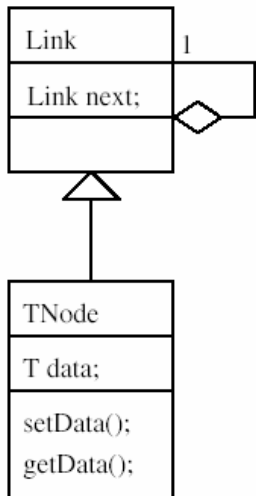
- Wegen der **Polymorphie** muss bei Ausdrücken für den Zugriff auf **Mitglieder** (**Felder** oder **Methoden**) geregelt werden, **welche Felder** oder **Methoden** in der **Klassenhierarchie** jeweils **gemeint** sind
 - Es gibt
 - Klassenvariablen, Klassenmethoden
 - Instanzvariablen, Instanzmethoden
- Ein **Zugriff** auf ein **Mitglied** kann nur durch einen in der **entsprechenden Klasse** **gültigen Namen** erfolgen
- Die Angabe der Klasse geschieht bei **Klassenvariablen** und **Klassenmethoden** entweder über eine Referenzvariable vom Typ der entsprechenden Klasse
 - in der Art
x.name
 - oder über einen **qualifizierten Namen** (*qualified name*), bei dem man die zugehörige Klasse in der Art
Klasse.name
vor dem Namen angibt

Überschreiben und Verdecken

- Der Zugriff auf **Instanzvariablen** und **Instanzmethoden** geschieht dagegen **immer** über eine **Objektreferenz**
 - da für einen Zugriff ja ein konkretes Objekt vorliegen muss
- Beim Zugriff über eine Objektreferenz gibt dabei der **deklarierte Typ** der **Objektreferenz** an, in **welcher Klasse** der **Name gültig** sein muss
- Bei einem Zugriff auf ein Mitglied sind daher zwei Fragen zu lösen:
 1. Ist der **Name** in der angegebenen Klasse **gültig**?
 2. **Welches** von **gegebenenfalls mehreren Mitgliedern gleichen Namens** in der **Klassenhierarchie** wird bei dem **Zugriff ausgewählt**?

Überschreiben und Verdecken

- **Beispiel:** Der folgende Code illustriert die Gültigkeit von Mitgliedsnamen



```

Link l = new TNode(); // OK: a TNode is a Link
T d = l.data;        // ERROR: no data field in
                    //           a Link!
T d = l.getData();  // ERROR: no getData method
                    //           in a Link!
TNode n = (TNode) l; // OK: l is instance of a TNode.
T d = n.data;        // OK: TNode has a data field.
T d = n.getData();  // OK: TNode has a getData method
l = n.next;         // OK: a TNode is a Link with next
    
```

Überschreiben und Verdecken

- Bezüglich der Deklaration von Mitgliedern gleichen Namens sind folgende Regeln zu beachten
 - Die Deklaration eines Feldes in einer Klasse verdeckt oder verbirgt (*hide*) jedes Feld gleichen Namens in einer Oberklasse
 - und zwar auch dann, wenn die Variablen verschiedenen Typ haben
 - Sowohl Klassenvariablen als auch Instanzvariablen können verdeckt werden
 - Über Referenzvariablen vom entsprechenden Typ kann man auf verdeckte Felder zugreifen

Überschreiben und Verdecken

- Die **Definition** einer **Instanzmethode überschreibt** (*override*) alle Instanzmethoden mit **gleicher Signatur** in **Oberklassen**, sofern sie aus der abgeleiteten Klasse **zugänglich** sind
 - **private Methoden** der Oberklasse werden in diesem Sinne nicht überschrieben
- Man spricht von **implementieren** statt von überschreiben, wenn die Methode der Oberklasse **abstrakt** ist, die der **abgeleiteten** Klasse aber nicht
- Es ist aber in **Java nicht** erlaubt, Klassenmethoden oder als **final** deklarierte Instanzmethoden zu überschreiben
 - dies führt zu einem Fehler bei der Übersetzung
- Eine Definition einer Klassenmethode **verdeckt** oder **verbirgt** (*hide*) alle Methoden gleicher Signatur in Oberklassen, falls sie zugänglich waren
 - private Methoden der Oberklasse werden in diesem Sinne nicht verborgen
- Es ist aber **nicht erlaubt**, dass eine **Klassenmethode** eine **Instanzmethode** verbirgt
 - wohingegen eine **Klassenvariable** eine **Instanzvariable verdecken** darf

- Für den **Zugriff** gilt in **Java** grundsätzlich:
 - Beim **Zugriff auf ein Feld** ist der *deklarierte Typ* der Objektreferenz entscheidend
 - Er legt im Zweifelsfall fest, welches von mehreren Feldern mit gleichen Bezeichnern in der Klassenhierarchie gemeint ist
 - Eine **explizite Typkonversion** beim Zugriff gilt als **lokale (Re-)Deklaration** des Typs
 - Beim **Aufruf einer Instanzmethode** bestimmt der *aktuelle Typ* des Objekts die **zugehörige Variante der Methode**, die auf dem Objekt **aktiviert** wird
 - Auf **verborgene Klassenvariablen** und **Klassenmethoden** kann man immer mit einem **qualifizierten Namen** zugreifen
 - Alternativ kann man über eine Referenzvariable zugreifen, die mit genau dem Typ der Klasse deklariert (oder zu dem Typ konvertiert) wurde, in der das Feld oder die Klassenmethode deklariert sind

- **Bemerkung:**

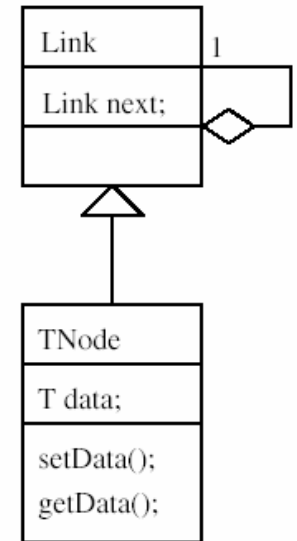
- Die Zugriffsregeln sind also ein weiterer Sachverhalt, der in die Entwurfsentscheidung einfließen muss, ob ein Feld
 - **direkt**
 - oder **über Selektoren** zugänglich gemacht wird
 - **Felder und Methoden** werden **hier verschieden behandelt**
- **Verschiedene Felder gleichen Namens** sind (deshalb) 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 ursprünglichen Kontrakt ja nicht, auf den sich **A** verlassen hatte und muss deshalb erlaubt sein

Überschreiben und Verdecken : *Beispiel*

- Die Entwickler von **TNode** brauchen einen **TDNode** und fügen deshalb eine Referenz **Link prev;** zu dem **TNode** hinzu
- Gleichzeitig fügen die Entwickler von **Link** ebenfalls eine Referenz **Link prev;** zur Klasse **Link** hinzu, da dies allgemein gewünscht wurde
 - Hierdurch soll der mühselig entwickelte neue **TDNode** aber nicht unbrauchbar werden
 - Außerdem haben die Entwickler von **TDNode** gleichzeitig die Methode **setData()** aus **TNode** überschrieben
 - Sie zählt jetzt die Zugriffe in einer Klassenvariable **counter**

```
class Link {  
    Link next;  
    Link prev;  
}
```

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



Überschreiben und Verdecken : *Beispiel*

- Die **aktuellen Zugriffe** im folgenden **Programmfragment** sind dann die jeweils im **Kommentar** angegebenen:

```
{
    T d = new T();
    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(d);           // setData name 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**

Überschreiben und Verdecken

- Auf die bisher beschriebene Art kann eine überschriebene Instanzmethode auf einem Objekt von außen nicht mehr aufgerufen werden
 - da ja ausschließlich der Objekttyp selbst die Methodenauswahl bestimmt
 - Im Beispiel ruft auch `n.setData()` die Methode aus `TDNode` auf, denn dies ist die Klasse des Objekts, auf das `n` verweist
 - Auch ein qualifizierter Aufruf `TNode.setData()` ist **nicht möglich**, denn hier fehlt ein Objekt
- In den Instanzmethoden der abgeleiteten Klasse selbst (und überall, wo die Verwendung von `this` erlaubt ist) kann aber das Schlüsselwort `super` benutzt werden, um auf Felder oder Methoden in der unmittelbaren Oberklasse zuzugreifen
 - Beim Zugriff auf ein Feld `name` in einer Methode von `A` ist `super.name` einfach eine Abkürzung für `((B) this).name`
 - Wenn `B` die unmittelbare Oberklasse ist

Überschreiben und Verdecken

- 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++;
    }
}
```

- Eine Klasse, welche mindestens eine abstrakte Funktion besitzt, ist eine **abstrakte Klasse** (*abstract class*)
 - Das heißt, es kann **keine Objekte** von **diesem Typ** geben
- Solche **abstrakten Klassendefinitionen** dienen der **Vereinbarung** von **gemeinsamen Daten** und **Funktionsschnittstellen** von bestimmten Klassen
- **Abgeleitete Klassen**, die **nicht alle** der abstrakten Funktionen **realisieren**, sind **selbst wieder abstrakte Klassen**
- **Abstrakte Klassen** können (definitionsgemäß) **nicht final** sein, da einige virtuelle Funktionen noch nicht endgültig feststehen
- In **Klassendiagrammen** kennzeichnen wir **abstrakte Klassen**, in dem wir ihren Namen *kursiv* setzen
- In **Java** werden **abstrakte Klassen** durch das **Schlüsselwort **abstract**** gekennzeichnet

- **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();
}
```

- 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 zur Klasse selbst gehören und konstant sind
 - Implizit ist jedes Feld eines Interfaces als **public**, **static** und **final** deklariert
- In **Java** beginnen Definitionen von **Interface-Klassen** mit dem **Schlüsselwort** **interface**
- Das Schlüsselwort **abstract** braucht **nicht benutzt** zu werden, um die Methoden des Interfaces als abstrakt zu kennzeichnen, da dies bei **allen Methoden** eines **Interface** (**implizit**) der Fall ist

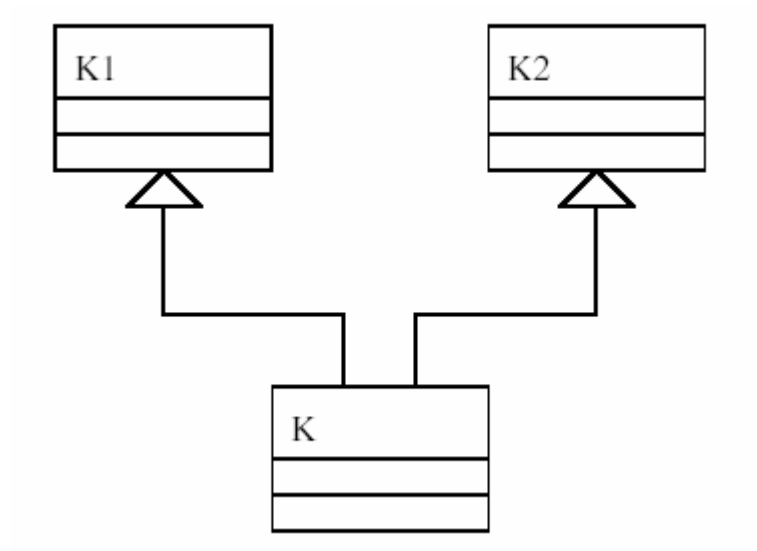
- **Beispiel:** Die Deklaration des im Paket `java.util` enthaltenen Interface `Enumeration` ist von folgender Form:

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

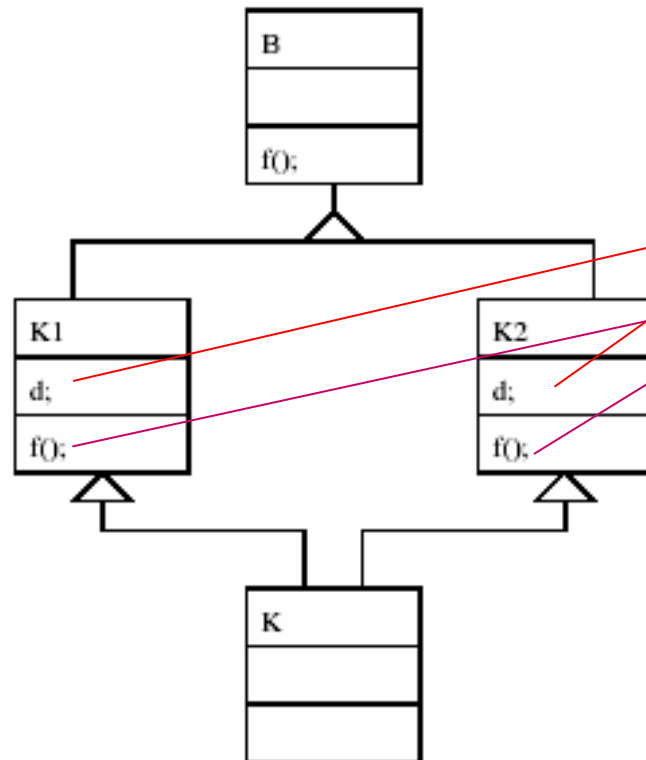
- Wenn eine **Klasse** die **abstrakten Methoden** eines **Interface** „erben“ (und **realisieren**) soll, so wird das Schlüsselwort **implements** benutzt
- Im Gegensatz zu der **Vererbung** von **Klassen**, bei der in **Java** nur Einfach-Vererbung zulässig ist, können auch mehrere Interfaces mittels der **implements-Klausel** aufgeführt und realisiert werden
- Eine Klasse, die ein Interface realisiert, muss *alle* der im Interface genannten Methoden implementieren

- Auch wenn es keine Objekte geben kann, die einer abstrakten Klasse oder einem Interface entsprechen, so können **Variable** oder **Parameter** von **Funktionen** mit dem **Typ** eines **Interface's** **deklariert** werden
 - Ein **Interface** oder eine abstrakte Klasse ist also in **Java** ein **legaler Referenztyp** (*reference type*)
- Will man **generisch programmieren** (siehe später), so ist es oftmals vorteilhaft, etwa einen **Parameter** mit dem **Referenztyp** eines **Interface** benutzen zu können
 - Aufgerufen wird eine solche Funktion dann mit einem Parameter, der eine nicht-abstrakte Klasse als Typ besitzt, die das Interface implementiert, also seine Schnittstelle „erbt“

- Ein Beispieldiagramm zu Mehrfachvererbung



- Eines der Probleme bei Mehrfachvererbung



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

- Java erlaubt **Mehrfachvererbung** nur von **Interfaces**, sonst ist nur Einfachvererbung erlaubt
 - Die Einschränkung in Java verhindert 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

Generische Datentypen

- In Java ist jede Klasse automatisch eine Unterklasse der Klasse **Object**
 - ohne dass dies eigens mit dem Schlüsselwort **extends** gekennzeichnet werden muss
- Datentypen wie Listen, Stacks, etc., die über Elementen vom Typ **Object** definiert sind, heißen **generisch** (*generic*) (bezüglich der Elementtypen)
 - Zum Beispiel lassen sich aus einer Liste über **Object** nun Listen über allen Klassentypen generieren, da die Zuweisung **Object data = d;** für jedes **d** eines Klassentyps gültig ist und die Objekte zur Laufzeit wieder zu ihrer Objekt-Klasse spezialisiert werden können
 - Da es für jeden elementaren Datentyp auch eine **Hüll-Klasse** (*wrapper class*) gibt, können solche Datentypen als allgemein generisch in Java betrachtet werden

- **Beispiel:**

Wir erhalten eine generische Klasse **List** (mit der generischen Knotenklasse **Node**) wenn wir in unseren Klassen **TList** und **TNode** aus den Typ **T** durch **Object** ersetzen.

```
public class ObjectNode {
    Object data; // Datenelement
    Node next; // Zeiger auf Listenzelle
    // ...
}
public class List {
    private ObjectNode head; // Kopf der Liste
    // ...
}
```

Generisches Programmieren

- Mit dem **Konzept** des **generischen Programmierens** möchte man ein und **denselben Code** für **Daten verschiedener Typen** verwenden
 - Man macht damit die Gemeinsamkeit verschiedener Datentypen deutlich, also z. B. von **Stacks** über **Integer** und **Stacks** über **Float**
- Man generiert gewissermaßen Code für Stacks über **Integer** und **Float** automatisch mit dem Code für Stacks über **Object**
- Dieses Konzept kann als Verallgemeinerung des Konzepts der Prozedurparameter angesehen werden, da dort ein und derselbe Code für verschiedene Werte eines Typs verwendet wird und hier für verschiedene Typen

- Zu **generischen Datentypen** gehören auch **generische Methoden**
 - Diese arbeiten auf allen Spezialisierungen des generischen Typs, müssen aber dazu evtl. zur Laufzeit oder Übersetzungszeit spezialisiert werden
- Für die Anpassung zur Laufzeit haben wir schon virtuelle Funktionen mit dynamischem Binden kennen gelernt
 - Das **Programmieren mit virtuellen Funktionen** stellt also schon eine **erste Art des generischen Programmierens** dar

- Wegen seiner großen Bedeutung heben wir dieses Programmierprinzip hier nochmals hervor

Die folgende Funktion **scan** implementiert den generischen Listendurchlauf.

```
public class List {  
    private Node head; // Kopf der Liste  
    // ...  
    void scan () {  
        for (Node cursor=head; cursor!=null; cursor=cursor.next) {  
            // Aktion (zur Zeit leer)  
            //etwa  
            System.out.println(cursor.data.toString());  
        }  
    }  
}
```

Virtuelle Funktion **toString**
generische Methode

Diese Methode **scan** funktioniert offensichtlich völlig gleich, egal von welchem aktuellen Typ die Datenelemente der Knoten sind.

- Viele Algorithmen zum Suchen und Sortieren können als generische Algorithmen implementiert werden, die durch einen Vergleichsoperator parametrisiert sind
- In Java wird dazu ein Interface **Comparable** im Paket **java.lang** zur Verfügung gestellt, in dem eine Vergleichsmethode **compareTo** wie folgt spezifiziert ist:
 - **compareTo** liefert eine Zahl kleiner als 0, gleich 0, oder größer als 0 zurück, je nachdem, ob das Objekt, auf dem **compareTo** läuft, kleiner, gleich, oder größer als das Argument-Objekt ist
 - „3-wertiger Vergleich“
 - Vgl. **String-Klasse**
 - Wenn das Objekt, mit dem verglichen wird, nicht den gleichen Typ besitzt, dann wird ein **Ausnahmeobjekt** geworfen
 - D.h. dies ist die Spezifikation im Interface; eine Implementierung hat diese Spezifikation zu erfüllen

- Beispiel

Test mit instanceof
Operator

Referenztyp einer
Schnittstellen-Klasse

```
public class List {
    private Node head; // Kopf der Liste
    // ...
    /**
     * Returns true iff m is smaller than all elements
     * of type Comparable in this list.
     */
    public boolean isMin(Comparable m) {
        for (Node cursor=head; cursor!=null;
             cursor=cursor.next) {
            if (cursor.data instanceof Comparable) {
                Comparable o = (Comparable) cursor.data;
                if (o.compareTo(m) <= 0) return false;
            }
        }
        return true;
    }
}
```

- In **C++** gibt es keine gemeinsame Urklasse wie **Object**
- Im Gegensatz zu **Java** unterstützt **C++** generisches Programmieren durch das Konzept der **Klassen-Muster** (*template classes*)
 - Auch Klassen-Schablonen genannt
 - Dies sind Klassen, die mit einem Typ parametrisiert sind
- Templates werden in Java 1.5 eingeführt

- Folgendes ist das Fragment einer generischen Knotenklasse in C++ (**kein** Java-Code)

```
template<class T> class Node { //C++ Code !
  T data; // generic data field
  Node* next; // pointer to next Node
public:
  T get_data() { return(data); }
  set_data (T d) { data = d; }
  // ..
}
```

Instantisierung zu Knoten über **int**:

```
Node<int> n; // C++
```

- Verwendung von Template-Klassen findet zunehmend Verwendung in C++
 - Etwa in der **Standard Template Library (STL)**
 - Generische Listen, Stacks, etc.
- Zur **Übersetzzeit** werden die **verschiedene Instanzen** des „generischen Programms“ vom **Compiler** generiert
 - **Vorteil:** Keine Indirektion über Virtual Function Table notwendig
 - Gerade bei relativ einfachen Operationen etwa auf Listen kann Indirektion zu einer wesentlichen Erhöhung der Laufzeit führen
 - Als „Hausnummer“: Verdoppelung der Laufzeit
 - **Nachteil:** Übersetzer erzeugt (i.A.) separaten Code für jede Typ-Instanz
 - Aufblähung des übersetzten Codes (**code bloat**)

Vergleich generisches Programmieren in Java und C++

- In **Java** generisches Programmieren via Ableitung von einer Basisklasse **Object** und virtuelle Funktionen
- **Vorteile**
 - Kein neues Sprachkonzept notwendig
 - Keine Duplizierung des Codes
 - Auch nicht nach Übersetzung
- **Nachteile**
 - Etwas langsamer zur Laufzeit, da immer bei Methodenaufruf eine Indirektion über Virtual Function Table notwendig
 - Keine klare Kennzeichnung als generische Methoden
 - In Java 1.5 sprachliche Kennzeichnung vorgesehen
 - Als „syntaktischer Zucker“ (syntactic sugar)
- In **C++** generisches Programmieren via **Template classes**
 - Java Methode eingeschränkt möglich, wenn eigene Basisklasse definiert werden kann
- **Vorteile von Template Classes**
 - Keine Indirektion über Virtual Function Table notwendig bei Methodenaufruf
 - Klare Kennzeichnung von generischen Programmen
- **Nachteile von Template Classes**
 - Übersetzter Code wird i.A. dupliziert für verschiedene Typen
 - Code bloat
 - Source Code aber nur einmal „generisch“ vorhanden
 - Zusätzliches Sprachkonzept
 - Interaktion von Template Klassen (die auch verschachtelt sein können) und virtuellen Funktionen kann sehr kompliziert werden

Generische Funktionsparameter

- **Ziel:** Erweiterung der generischen Listen um Funktionsparameter
 - Kann bei anderen generischen Datentypen analog erfolgen
- Bei **scan** soll etwa eine Funktion auf alle Listenelemente angewendet werden
 - Etwa bei Listen über Ganzzahldatentypen die Nachfolgerfunktion **succ**: $x \mapsto x+1$
 - Diese **Funktion** soll ein „Parameter“ von **scan** werden
 - Die Funktion **scan** ist dann **Funktion höherer Stufe**
 - **Funktion erster Stufe:** Parameter enthalten keine Funktionen
 - **Funktion $(n+1)$ -ter Stufe:** Enthält als **Parameter Funktion n -ter Stufe**

Funktionen höherer Stufe

- In **funktionalen Sprachen**:
 - Konzept Funktionen höherer Stufe direkt unterstützt
- In **C, C++**:
 - Zeiger auf Code von Funktionen kann als Parameter übergeben werden
 - Typüberprüfung kann in gewissem Rahmen vorgenommen werden, aber nicht vollständig sicher
- **Java** kennt diese beiden Konzepte **nicht direkt**
 - Sondern nur Konzept von Referenztypen
 - Etwa auf eine Schnittstellen Klasse
 - Hiermit können Funktionen höherer Stufe simuliert werden!
 - Aktionsobjekte

Generische Funktionsparameter

- Statt eines Funktionsparameters benutzen wir in Java ein **Aktionsobjekt** (*action object*) als Parameter von **scan**
- **Verschiedene Aktionen** brauchen dann **verschiedene Aktionsobjekte verschiedener Klassen**
- **Allen Objekten** muss aber die **Aufrufchnittstelle** (Signatur) der **Aktionsfunktion** **gemeinsam** sein
 - Da diese ja beim generischen Durchlauf aufgerufen werden muss
 - Diese **Gemeinsamkeit spezifizieren** wir in einem **Interface**, das **von allen Aktionsklassen** implementiert werden muss

Generische Funktionsparameter

- Da auch **abstrakte Klassen** und **Interfaces** einen **Referenztyp definieren**, können wir einen Parameter definieren, der etwa folgendes Interface als Typ besitzt

```
interface MapCarInterface {  
    /**  
     * Abstract function whose realizations  
     * operate on list nodes.  
     */  
    public void action(Node n);  
}
```

Name nach Funktion
mapcar in **LISP** gewählt,
die entsprechendes leistet

Wenn **p** ein formaler Parameter vom Typ **MapCarInterface** ist und **n** eine Variable vom Typ **Node**, dann können wir in den Methoden zum Listendurchlauf Anweisungen der Form

```
    p.action(n);  
verwenden.
```

Generische Funktionsparameter

- Implementierung der Funktion höherer Stufe **mapcar** dann als Methode in generischer Listenklasse wie folgt möglich

```
public class List {
    private Node head; // Kopf der Liste
    // ...
    /**
     * Wendet p.action auf jedes
     * Datenfeld der Liste an.
     */
    public void mapcar(MapCarInterface p) {
        // go over list
        for(Node cursor=head;
            cursor!=null;
            cursor=cursor.next)
            // apply function that works on the data (car)
            // part of the node
            p.action(cursor);
    }
}
```

Formaler Parameter vom Typ **MapCarInterface** wird bei Aufruf durch abgeleiteten Klassentyp instantiiert, der **action** implementiert

mapcar ist eine Realisierung von **scan**, in der die Aktion genommen wird, die in **MapCarInterface** spezifiziert ist

Generische Funktionsparameter

- Beispiel: Aktueller Typ sei folgender (**PrintAction** genannt)

```
public class PrintAction
    implements MapCarInterface {
    // ...
    /**
     * Print action on nodes.
     */
    public void action(Node n) {
        System.out.print(n.data.toString());
    }
}
```

Dann gibt das folgende Programmstück die Liste `l` auf `System.out` aus.

```
List l = // ... some list
PrintAction p = new PrintAction();
l.mapcar(p);
```

- Weiteres Beispiel
 - Änderung des Knoteninhalts

```
public class SuccessorAction
implements MapCarInterface {
/**
 * Changes a data field which is of
 * type Integer to its successor.
 */
public void action(Node n) {
    if (n.data instanceof Integer) { // check type
        // cast to actual type
        Integer tmp = (Integer) n.data;
        //assign successor to data
        n.data = new Integer(tmp.intValue()+1);
    }
}
}
```

Generische Funktionsparameter

- Diskurs: Alternative Implementierung bei nur lesendem Zugriff

```
interface ObjectActionInterface {  
    /**  
     * Abstract function whose realizations  
     * operate on data.  
     */  
    public void oAction(Object d);  
}
```

Object statt
Node;
(problematische)
Idee: greife direkt
auf **data** zu

Eine entsprechende Implementierung von **action** wäre dann aber **nicht** mehr möglich:
// Beispiel, das das Problem mit schreibenden
// Aktionen zeigt:
// Beim Funktionsaufruf wird eine Kopie
// des Referenzparameters hergestellt
public void oAction(Object d) {
 if (d instanceof Integer) {
 Integer tmp = (Integer) d;
 d = new Integer(tmp.intValue()+1);
 // d zeigt auf neues Objekt,
 // aber **nicht** die Referenzvariable,
 // deren Wert beim Aufruf an d zugewiesen
 // wird.
 }
}

Generische Funktionsparameter

- Weiteres Beispiel: Summe der Listenfelder

```
public class SumAction
    implements MapCarInterface {
    int sum=0; // accumulates the int values
    /**
     * Addiert den Wert des data Feldes zu
     * sum hinzu, falls data vom Typ Integer ist.
     */
    public void action(Node n) {
        if (n.data instanceof Integer) { // check type
            // cast to actual type
            Integer tmp = (Integer) n.data;
            // accumulate value in sum
            sum += tmp.intValue();
        }
    }
}
```

Das folgende Programmstück gibt die Summe der Elemente einer Liste `l` von Integer-Werten auf `System.out` aus.

```
List l = // ... some list of Integer
SumAction s = new SumAction();
l.mapcar(s);
System.out.println(s.sum);
```

Bemerkung: Im funktionalen Programmieren spricht man hier allgemein von einem **fold-Operator**, da das Ergebnis eine auf einen Skalar-Wert zusammengefaltete Liste ist. Der **Akkumulator** (im Beispiel `sum`) wird zu Beginn mit dem **neutralen Element** der **fold-Operation** initialisiert (im Beispiel ist `0` das **neutrale Element** von `+`).