

Praktische Informatik 1

Vererbung

Thomas Röfer

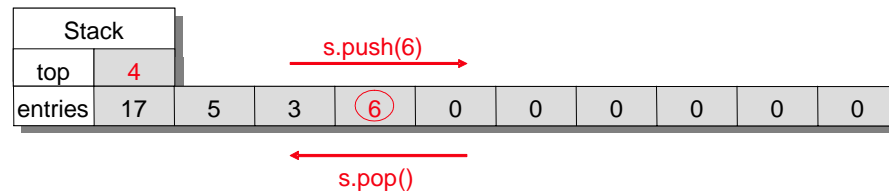
- Pakete
- Vererbung
- Frühes / spätes Binden
- Lokale Klassen
- Zugriffsschutz
- Abstrakte Klassen und Schnittstellen
- Anonyme Klassen
- Mehrfaches Erben
- Musterlösung, Übungsblatt

Rückblick „Listen, Stapel, Warteschlangen“

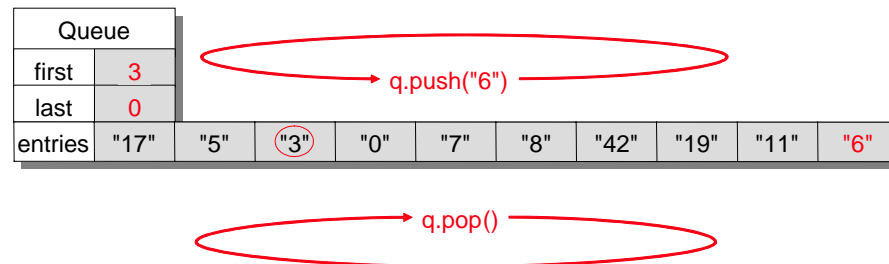
Abstrakte Datentypen

$create : \rightarrow Stack$
 $push : Stack \times Entry \rightarrow Stack$
 $top : Stack \rightarrow Entry$
 $pop : Stack \rightarrow Stack$
 $empty : Stack \rightarrow boolean$

Stapel



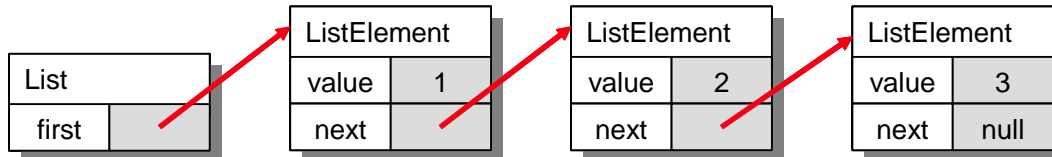
Warteschlange



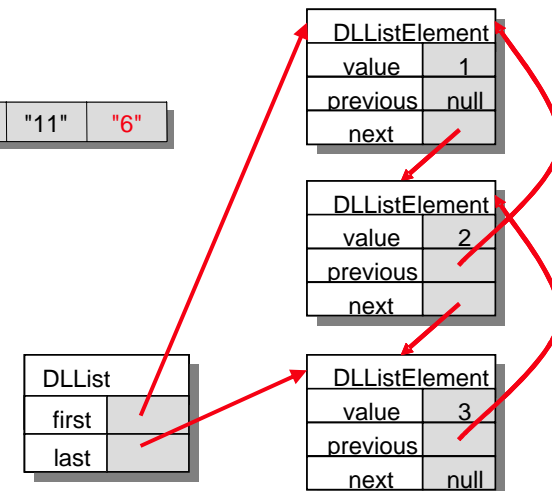
Algebraische A.D.

$top(push(s, e)) = e$
 $pop(push(s, e)) = s$
 $empty(create())$
 $\neg empty(push(s, e))$

Einfach verkettete Liste



Doppelt verkettete Liste



Pakete (Packages)

- Konzept
 - Klassen dienen zum Zusammenfassen von Methoden und Attributen
 - Pakete dienen zum Zusammenfassen von Klassen und weiterer Pakete
- Im Quelltext
 - Am Anfang: **package** *Paketname* { . *Paketname* }
 - Namensraum importieren:
import *Paketname* { . *Paketname* } . (* | *Klassenname*)
- Im Dateisystem
 - Pakete werden als Ordner repräsentiert
- In BlueJ
 - Ein neues Paket kann man mit „*Bearbeiten|Neues Paket ...*“ erzeugen
 - Pakete werden fast wie eigene Projekte behandelt

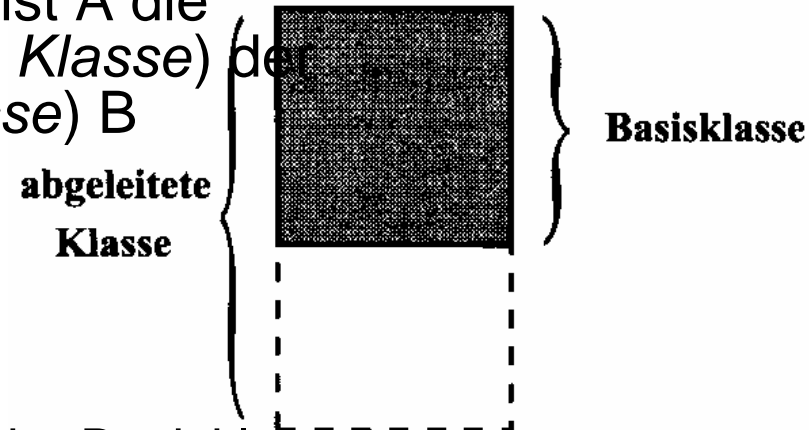
```
java.lang.System.out.println("Hallo")
```

java.lang.System.out.println("Hallo")

PaketPaketKlasseAttributMethode

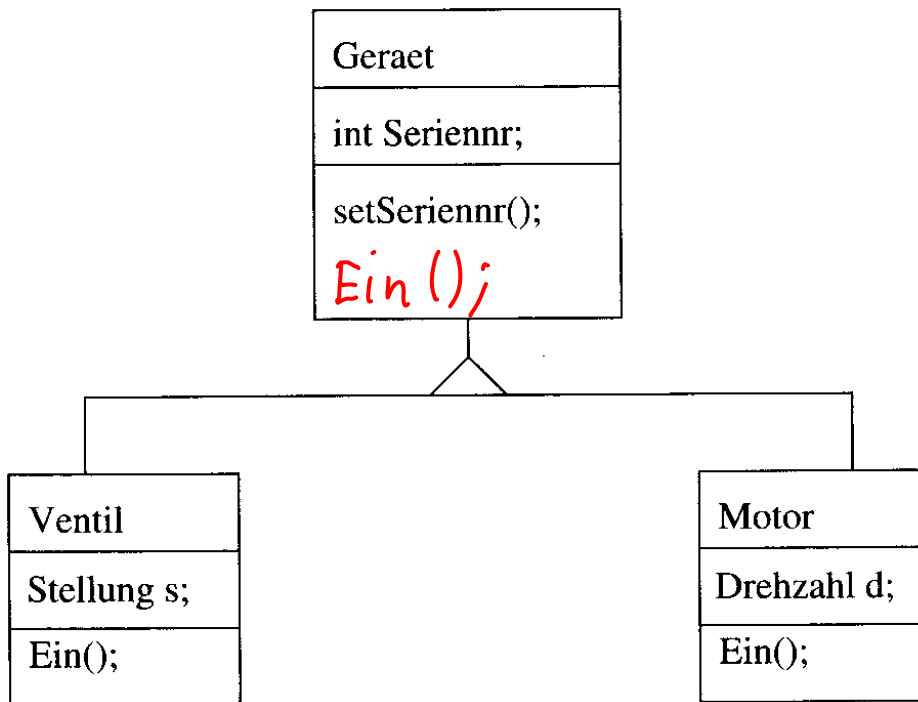
Vererbung und abgeleitete Klassen

- Erlaubt die Modellierung der is-a Relation zwischen Klassen
- Wenn Klasse A von Klasse B erbt, ist A die *Unterklasse (subclass / abgeleitete Klasse)* der *Oberklasse (superclass / Basisklasse)* B
- Ziele
 - Wiederverwendung von Code
 - Unterstützung von *Polymorphie*
- Sichtweisen
 - Abgeleitete Klasse ist Erweiterung der Basisklasse
 - Basisklasse ist allgemeiner, abgeleitete Klasse ist spezieller
- Reale Vererbung / virtuelle Vererbung
 - Eine Methode wird *real* vererbt, wenn die abgeleitete Klasse die Methode wiederverwendet
 - Eine Methode wird *virtuell* vererbt, wenn die abgeleitete Klasse die Methode *überschreibt*



Vererbung – Beispiel

*Geraet g = new Motor();
g.Ein();*

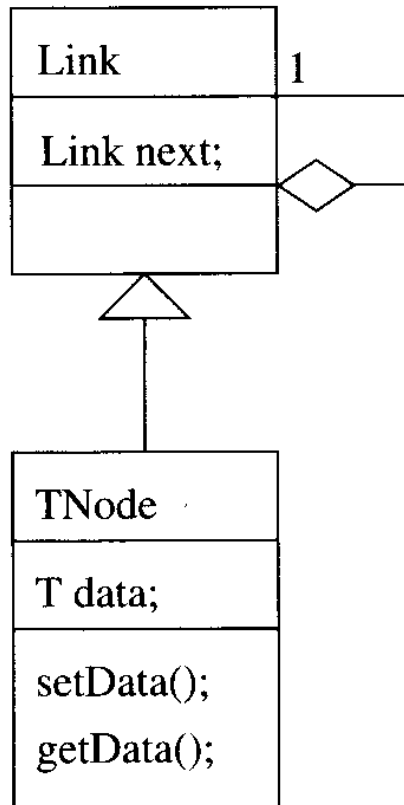


```
class Geraet {
    int Seriennr;
    void setSeriennr() {...}
}
```

```
class Ventil extends Geraet {
    Stellung s;
    void Ein() {...}
}
```

```
class Motor extends Geraet {
    Drehzahl d;
    void Ein() {...}
}
```

Vererbung – Beispiel



```
class Link {
    Link next;
}

class TNode extends Link {
    T data;

    void setData(T d) {
        data = d;
    }

    T getData() {
        return data;
    }
}
```

Überschreibung

- Überschreibung von Methoden
 - Eine abgeleitete Klasse definiert eine Methode mit *derselben Signatur* wie ihre Basisklasse
 - Überschreiben durch Methode mit *derselben Parameterliste* aber *anderem Rückgabotyp* nicht erlaubt
 - Wurde eine Methode als *final* deklariert, kann sie nicht überschrieben werden
- Überschreibung von Attributen
 - Der Zugriff auf Attribute erfolgt durch *frühe Bindung* (also zur Kompilierzeit)

Überschreibung

```
class A {  
    int fn1(int i) {...}  
    int fn2(int i) {...}  
    final int fn3(int i) {...}  
}  
  
class B extends A {  
    int fn1(int i) {           // Überschreibung  
        return 2 * super.fn1(i); // Aufruf der Basisklassenmethode  
    }  
  
    int fn1(double d) {...} // Überladung  
  
    double fn2(int i) {...} // Fehler: gleiche Parameterliste, anderer Rückgabetyp  
  
    int fn3(int i) {...}     // Fehler: Überschreibung nicht erlaubt  
}
```

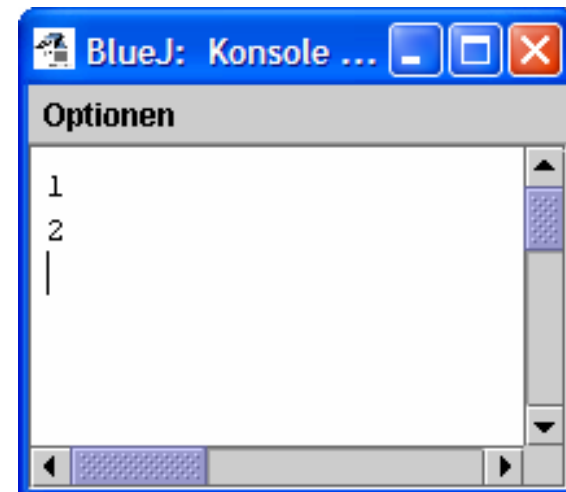
Virtuelle Methoden und spätes Binden

- Virtuelle Methoden
 - Methoden bezeichnet man als *virtuell*, wenn man über eine Referenz vom Typ einer Basisklasse eine Methode einer abgeleiteten Klasse aufrufen kann
 - In Java sind alle Methoden *virtuell*
- Spätes Binden
 - Wenn erst zur Laufzeit eines Programms entschieden werden kann, welche (*virtuelle*) Methode aufgerufen wird, bezeichnet man dies als *spätes* oder *dynamisches Binden*
- Virtuelle Methodentabellen
 - Zur Realisierung des späten Bindens enthält jedes Java-Objekt eine versteckte Referenz auf die *virtuelle Methodentabelle* seiner Klasse
 - Darin stehen die Adressen der Methoden, die für ein Objekt dieser Klasse aufgerufen werden müssen

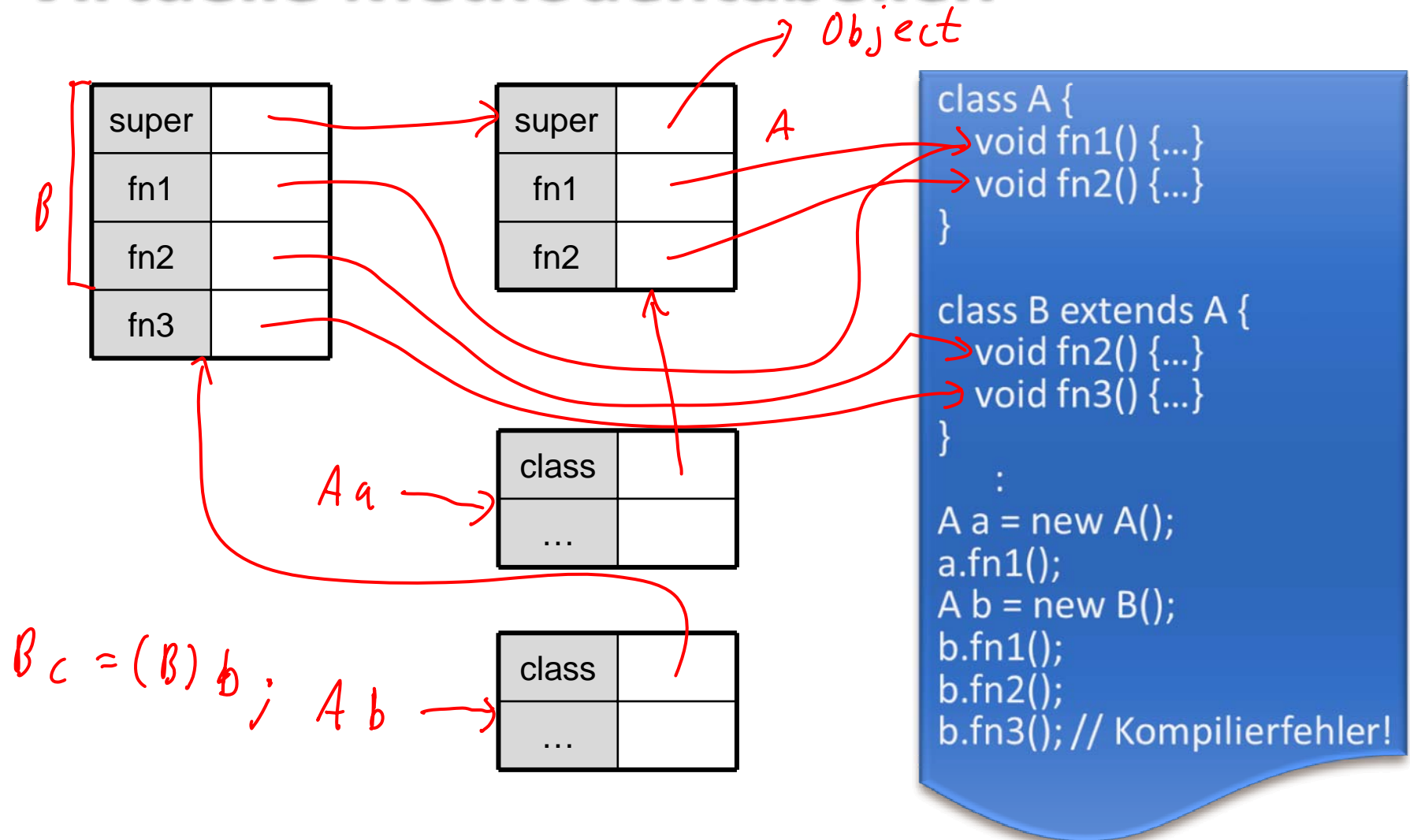
Frühes Binden / Spätes Binden

```
class A {  
    int value;  
  
    A() {value = 1;}  
  
    int getValue() {return value;}  
}  
  
class B extends A {  
    int value;  
  
    B() {value = 2;}  
  
    int getValue() {return value;}  
}
```

```
A a = new B();  
System.out.println(a.value);  
System.out.println(a.getValue());
```



Virtuelle Methodentabellen



Konstruktoren

- Standardkonstruktor
 - Wenn man keine Konstruktoren definiert, erzeugt Java automatisch einen Konstruktor ohne Parameter (nur dann)
- Konstruktionsreihenfolge
 - Objekte werden immer von der Basisklasse zur abgeleiteten Klasse konstruiert
 - Um einen anderen als den Standardkonstruktor der Basisklasse auszuführen, muss die erste Zeile eines Konstruktors *super(...)* enthalten
 - Ist dies nicht so, führt Java automatisch *super()* am Anfang aus
- Hinweis
 - Ein Konstruktor kann einen anderen aufrufen
 - Innerhalb eines Konstruktors einer Basisklasse man Methoden einer abgeleiteten Klasse aufrufen bevor deren Konstruktor abgearbeitet wurde!

```
class A {  
    int num;  
  
    A(int num) {  
        this.num = num;  
    }  
  
    A() {  
        this(10);  
    }  
}
```


Standardkonstruktor – Beispiele

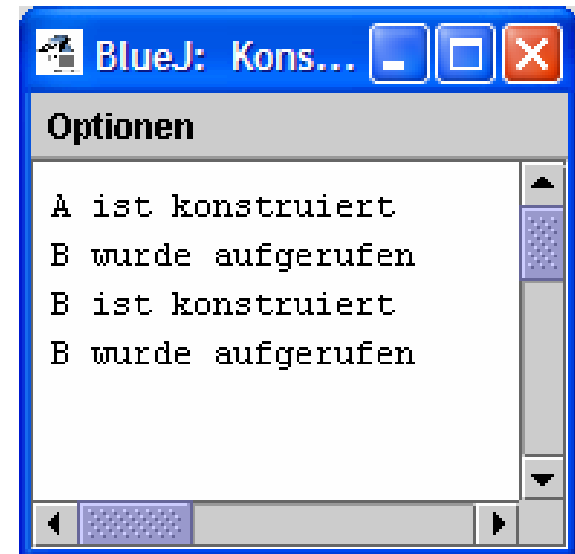
```
class A {  
    A(int a) {  
        System.out.println(a);  
    }  
}  
  
class B extends A {  
    B(double d) {  
        super((int) d);  
    }  
  
    B() {  
        // Fehler: A() nicht definiert  
    }  
}
```

```
class A {  
    int a;  
}  
  
class B extends A {  
    int i;  
  
    B(int i) {  
        this.i = i;  
        a = 17;  
    }  
}
```

Konstruktionsreihenfolge – Beispiel

```
class A {  
    A() {  
        System.out.println("A ist konstruiert");  
        print();  
    }  
  
    void print() {  
        System.out.println("A wurde aufgerufen");  
    }  
}  
  
class B extends A {  
    B() {  
        System.out.println("B ist konstruiert");  
        print();  
    }  
  
    void print() {  
        System.out.println("B wurde aufgerufen");  
    }  
}
```

B b = new B();



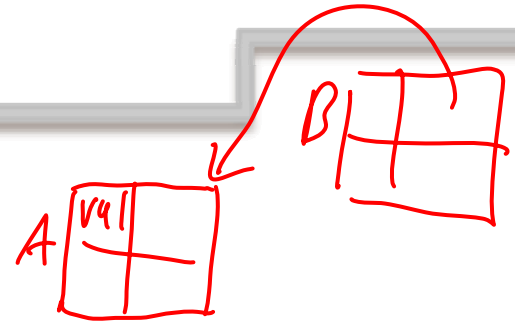
Typumwandlung in Klassenhierarchie

- In Richtung Basisklasse
 - Automatische Typumwandlung
 - *class B extends A {...*
A a = new B();
- In Richtung abgeleitete Klasse
 - Durch explizite Typumwandlung (Casting)
 - *B b = (B) a;*
 - Falls nicht möglich, wird *ClassCastException* erzeugt
 - *A a1 = new A();*
B b1 = (B) a1;
- Typabfrage zur Laufzeit
 - *if (a1 instanceof B) {*
b1 = (B) a1;
}

Lokale Klassen

- Eine Klasse ist *lokal*, wenn sie innerhalb einer anderen Klasse definiert wird
- Statische lokale Klassen
 - Eine *statische lokale Klasse* unterscheidet sich außer in ihrer Sichtbarkeit nicht von globalen Klassen
 - Statische Methoden und Attribute der übergeordneten Klasse liegen im Sichtbarkeitsbereich der lokalen Klasse
- Objekt-lokale Klassen
 - Objekte von *nicht-statischen, lokalen Klassen* enthalten eine versteckte Referenz auf das Objekt der übergeordneten Klasse, in dessen Kontext sie erzeugt wurden
 - Damit haben sie Zugriff auf dessen Methoden und Attribute, d.h. diese befinden sich im Sichtbarkeitsbereich der lokalen Klasse

Lokale Klassen – Beispiel



```
class A {  
    int value;  
  
    class B {  
        void print() {  
            System.out.println(value);  
        }  
    }  
  
    static class C {  
        void print() {  
            System.out.println("Hello");  
        }  
    }  
}
```

```
A() {  
    value = 2;  
    B b = new B();  
    b.print();  
    C c = new C();  
    c.print();  
}  
  
static void hello() {  
    C c = new C();  
    c.print();  
    B b = new B(); // Fehler!  
}  
}
```

Zugriffsschutz

- Dient der Kapselung der Funktionalität einer Klasse
- Versteckt die interne Implementierung der Funktionalität
- Je kleiner die öffentliche Schnittstelle, desto flexibler kann die interne Implementierung angepasst werden, ohne andere Klassen ändern zu müssen
- Steigert die Wartbarkeit von Software

Zugriffsmodifizierer (access modifiers)

- *private*
 - Klassen, Methoden und Attribute können nur innerhalb derselben Klasse genutzt werden
 - Kann nicht vor globalen Klassen stehen
- *protected*
 - Klassen, Methoden und Attribute können nur innerhalb derselben Klasse und von ihr abgeleiteten Klassen genutzt werden
 - Kann nicht vor globalen Klassen stehen
- Keine Angabe
 - Klassen, Methoden und Attribute können nur innerhalb desselben Pakets genutzt werden
 - Nennt man auch *package private*
- *public*
 - Klassen, Methoden und Attribute können überall genutzt werden

Indizierte Liste mit Zugriffsschutz

```
public class List {  
    private static class Element {  
        public int entry;  
        public Element next;  
  
        public Element(int e, Element n) {  
            entry = e;  
            next = n;  
        }  
    }  
  
    private Element first;  
  
    public List() {  
        first = null;  
    }  
}
```

```
    private Element at(int pos) {  
        Element current = first;  
        while (pos > 0 && current != null) {  
            current = current.next;  
            --pos;  
        }  
        return current;  
    }  
  
    public int get(int pos) {  
        return at(pos).entry;  
    }  
  
    public void set(int pos, int entry) {  
        at(pos).entry = entry;  
    }  
}
```


Abstrakte Klassen

- Eine Klasse ist abstrakt, wenn mindestens eine Methode *abstrakt* ist, d.h. ihre Signatur zwar angegeben, aber nicht implementiert ist
- Objekte abstrakter Klassen können nicht konstruiert werden, wohl aber Objekte abgeleiteter Klassen, bei denen alle abstrakten Methoden implementiert wurden
- Es kann Referenzen vom Typ einer abstrakten Klasse geben
- Abstrakte Klassen erlauben die Implementierung einer Teilfunktionalität, bei der bestimmte Details offen gelassen werden
- Durch die abstrakten Methoden werden abgeleitete Klassen dazu verpflichtet, diese zu implementieren

```
abstract class Geraet {  
    int seriennummer;  
    abstract void ein();  
}  
  
class Ventil extends Geraet {  
    Stellung s;  
    void ein() {...}  
}  
  
class Motor extends Geraet {  
    Drehzahl d;  
    void ein() {...}  
}  
  
Geraet m = new Motor();  
m.ein();  
Geraet g = new Geraet();  
// Fehler!
```