Kapitel 7: Objektorientierung

Das Grundparadigma der objektorientierten Programmierung ist, dass ein Programm eine Ansammlung von *Objekten* ist, die miteinander interagieren. Jedes Objekt gehört zu einer bestimmten *Klasse*, und ist mit *Datenfeldern* und *Methoden* ausgerüstet. Der Wert der Datenfelder beschreibt den Zustand des Objektes, die Methoden dienen dazu, diesen Zustand zu verändern und mit anderen Objekten zu kommunizieren.

7.1 abstrakte Datentypen, Objekte, Klassen

Abstrakte Datentypen (ADT) sind ein Konzept aus der theoretischen Informatik, welches für die objektorientierte Programmierung eine ähnliche grundlegende Rolle spielt wie der λ-Kalkül für die funktionale Programmierung oder die Turing-Maschine für die imperative Programmierung. Formal definieren wir den Begriff der Signatur: Eine Signatur Σ ist ein Paar, bestehend aus einer Grundmenge M und einer Menge von Operationen und Prädikaten Φ:

$$\Sigma = (M, \Phi)$$

Zu jedem Element von Φ wird außerdem ihre Stelligkeit angegeben (zur Erinnerung: eine *n*-stellige *Operation* auf einer Menge M ist eine Funktion $M^n \rightarrow M$, eine *n*-stellige *Operation* auf einer Menge M ist eine Funktion $M^n \rightarrow \mathbb{B}$)
Beispiel für eine Signatur ist also etwa : $(\mathbb{N}_0, 0, s, +, *)$. Dabei sei `0` nullstellig, `s` einstellig, und `+` und `*` zweistellig.

$0 : \rightarrow \mathbb{N}_0$

$s : \mathbb{N}_0 \rightarrow \mathbb{N}_0$

$+ : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$

$* : \mathbb{N}_0 \times \mathbb{N}_0 \rightarrow \mathbb{N}_0$

Es erhebt sich die Frage, welche Datenobjekte vom Typ $\mathbb{N}_0$ es (mindestens) gibt. Unter der *Termalgebra* einer Signatur versteht man alle Objekte, die sich als Ergebnis wohlgeformter Terme auf Grund der Signatur darstellen lassen. Beispiele sind

- `0, s(0), s(s(0)), s(s(s(0))), …`
- `0+0, (0+0)+0, …`
- `s(0)+s(0), s(s(0))+s(s(s(0))), …`
- `s(s(0))*(s(s(0))*(s(s(0)))), (0*s(0))+s(0), …`

Die Objekte, die sich so darstellen lassen, nennt man die *Termalgebra* der Signatur.

Nicht alle Terme geben verschiedene Werte: z.B ist s(0)+s(0) gleich zu s(s(0)) (manchmal ist 1+1=2). Daher nimmt man eine Äquivalenzklassenbildung durch Angabe von (allgemeingültigen) Gesetzen vor:

- `x+0=x`
- `x+s(y)=s(x+y)`
- `x*0=0`
- `x*s(y)=x+(x*y)`

Damit lässt sich obige Aussage beweisen (1+1=2):
`s(0)+s(0) =? s(s(0))`

```
s(0)+s(0) = s(0 + s(0))
          = s(s(0 + 0))
          = s(s(0))
```

Ein abstrakter Datentyp (ADT) T besteht aus einer Signatur **Σ** und einer Menge von algebraischen Gesetzen (Gleichungen) **Γ**

$T = (\Sigma, \Gamma)$

Wenn die Gesetze ausschließlich aus Gleichungen zwischen Elementen von **M** bestehen, spricht man auch von einer *Varietät* (engl.: variety). Oft lässt man auch Bedingungen und Ungleichungen zu, dies ist aber eine Erweiterung des ursprünglichen Konzepts. Die Gesetze beschreiben, was für alle Objekte dieses Typs gelten soll.

Beispiel: ADT IntSet „Menge ganzer Zahlen“
Signatur: $(\mathtt{IntSet}, \emptyset, \in, \subseteq, \cup, \cap, -)$

$\emptyset : \rightarrow \mathtt{IntSet}$
$\cup : \mathtt{IntSet} \times \mathtt{IntSet} \rightarrow \mathtt{IntSet}$
$\in : \mathbf{Z} \times \mathtt{IntSet} \rightarrow \mathtt{boolean}$

Gesetze: z.B.

$x \cup y = y \cup x$
$x \cap \emptyset = \emptyset$
...

Konkrete Mengen ganzer Zahlen (z.B. {1,2,3} oder {x | x%2=0} sind Instanzen des abstrakten Typs. Die Gesetze legen fest, was für alle Instanzen gelten soll. Beispiel:

$\{1,2,3\} \cup \{4,5\} = \{4,5\} \cup \{1,2,3\}$
$\{x \mid x\%2=0\} \cap \emptyset = \emptyset$

Beispiel: Paare ganzer Zahlen Z×Z:
ADT ZZ = (**ZZ, first, second, conc**)

$\mathbf{first, second : ZZ} \rightarrow \mathbb{Z}$
$\mathbf{conc : Z \times Z \rightarrow ZZ}$

Gesetze:

first(conc(x,y))=x
second(conc(x,y))=y

Instanzen:

⟨3,-5⟩, ⟨0,0⟩, ⟨-12,12⟩, …
first(⟨3,0⟩)=3
conc(1,2)=⟨1,2⟩

Beispiel: Sequenzen (parametrisiert mit Basistyp σ)
ADT seq⟨σ⟩: (seq⟨σ⟩, empty, isEmpty,
prefix, first, rest,
postfix, last, lead):

empty : → seq⟨σ⟩
isEmpty : seq⟨σ⟩ → boolean

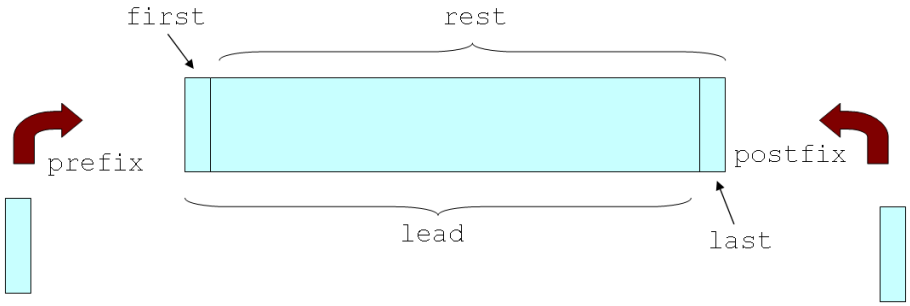
```
prefix: σ × seq⟨σ⟩ → seq⟨σ⟩
first: seq⟨σ⟩ → σ
rest: seq⟨σ⟩ → seq⟨σ⟩

postfix: seq⟨σ⟩ × σ → seq⟨σ⟩
last: seq⟨σ⟩ → σ
lead: seq⟨σ⟩ → seq⟨σ⟩

isEmpty(empty) = true
isEmpty(prefix(a,x)) = false
isEmpty(postfix(x,a)) = false

first(prefix(a,x))= a
rest(prefix(a,x))= x

last(postfix(x,a))= a
lead(postfix(x,a))= x
```



Eigenschaften

```
first(rest(prefix(a, prefix(b, empty)))) = b
rest(rest(prefix(a, prefix(b, empty)))) = empty
last(lead(postfix(a, postfix(b,empty)))) = b
…
prefix(a,empty) =? postfix(empty,a)
```

Typerweiterung
abgeleitete Operation +:

```
+: (seq⟨σ⟩ ∪σ) × (seq⟨σ⟩ ∪σ)→ seq⟨σ⟩
prefix(a,x)=a+x
postfix(x,a)=x+a
x+empty=x
empty+x=x
x+(y+z)=(x+y)+z
```

+ ist also ein “überladener” Operator, der für mehrere verschiedene Eingabetypen eine Sequenz liefert. Er kann wie folgt rekursiv definiert werden:

```
x+empty=x
x+postfix(y,a) = postfix(x+y,a)
```

```
empty+x=x
prefix(a,x)+y = prefix(a,x+y)
```

Stapel und Schlangen

Stapel („stack“): einseitiger Zugriff

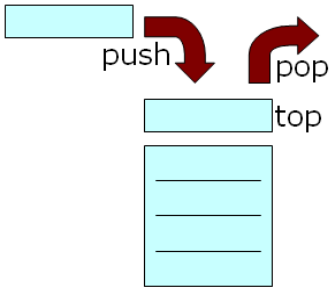
`empty, isEmpty, first, rest, prefix` **– oder –** `empty, isEmpty, last, lead, postfix`

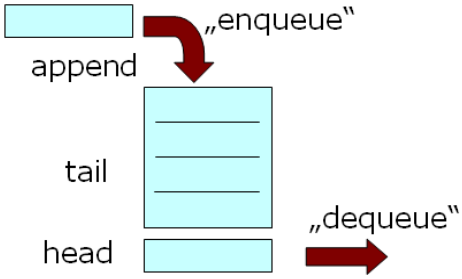
top ≅ `first/last`, *pop* ≅ `rest/lead`, *push* ≅ `prefix/postfix`

Schlange („queue“):

`empty, isEmpty, first, rest, postfix` **– oder –** `empty, isEmpty, last, lead, prefix`

head ≅ `first/last`, *tail* ≅ `rest/lead`, *append* ≅ `postfix/prefix`





Multimengen

```
bag⟨σ⟩ :
empty : → bag⟨σ⟩
isEmpty : bag⟨σ⟩ → boolean
insert: bag⟨σ⟩ × σ → bag⟨σ⟩
delete: bag⟨σ⟩ × σ → bag⟨σ⟩
elem: σ × bag⟨σ⟩ → boolean
any: bag⟨σ⟩ → σ

isEmpty(empty) = true
isEmpty(insert(x,a)) = false

insert(insert(x,a),b)= insert(insert(x,b),a)

delete(empty,a) = empty
delete(insert(x,a),a) = x
delete(insert(x,b),a) = insert(delete(x,a),b)

elem(a, empty) = false
elem(a, insert(x,a)) = true
elem(a,insert(x,b) = elem(a,x) (a≠b)
elem(any(x),x) = true (x≠empty)
```

Beispiel-Algorithmus für Multimengen

card: $\text{bag}\langle\sigma\rangle \times \sigma \rightarrow N_0$

```
int card (bag⟨σ⟩ x, σ a){
 if (isEmpty(x)) return 0;
 else {
  σ b = any(x);
  return card(delete(x,b)) + ((a=b)?1:0);
 }
}
```

7.2 Klassen, Objekte, Methoden, Datenfelder, Konstruktoren

Eine Klasse ist die Java-Realisierung eines abstrakten Datentyps. In der objektorientierten Programmierung sind Klassen das grundlegende Strukturierungsmittel.

Klasse = Datenfelder + Methoden

Datenfelder sind die Zustandsvariablen des Objekts, Methoden die Operationen, Funktionen oder Prozeduren zur Zustandsänderung. Objekte sind Instanzen von Klassen, ähnlich wie Elemente Bestandteile einer Menge sind. Die folgenden Begriffe werden fast synonym verwendet (mit minimalen, subtilen Unterschiede!)

- **Klasse** ~ Typ ~ Art
- **Objekt** ~ Instanz ~ Exemplar
- **Datenfeld** ~ Objektattribut ~ Instanzvariable ~ Eigenschaft (passives Merkmal)
- **Methode** ~ Funktion/Prozedur ~ Operation ~ Fähigkeit (aktives Merkmal)
- **Parameter** ~ Argument ~ Eingabewert

Beispiel für eine Klasse: Philosophen sitzen um einen Tisch, denken, werden hungrig und essen:

```
class Philosoph {
  boolean hungrig;
  void denken() {...};
  void essen() {...};
}
```

Klassen bestehen also im Wesentlichen aus der Deklaration von Daten- und Funktionsteilen (vgl. Turing/von-Neumann Analogie zwischen Daten und Programmen). Datenfelder bestimmen durch ihren aktuellen Wert während einer Berechnung den Zustand der abgeleiteten Objekte, Methoden definieren die Aktionsmöglichkeiten der Objekte.

Beispiel: das Objekt `Sokrates´ ist eine Instanz der Klasse `Mensch´, kann im Zustand `hungrig´ sein und beherrscht die Methode `denken´ mit Ergebnistyp `Erkenntnis´ (void).

Objekte können von anderen Objekten erzeugt und aufgerufen werden:

```
public static void main(String[] args) {
  Philosoph sokrates;
  sokrates = new Philosoph();
  sokrates.denken();
  if(sokrates.hungrig) sokrates.essen();
```

```
    System.out.println(sokrates.hungrig);
  }
```

Objektorientierte Modellierung

Klassen modellieren ein Konzept eines Anwendungsbereiches oder eine Gemeinsamkeit mehrerer Dinge (platonische „Idee“); Objekte modellieren die konkreten Akteure oder Gegenstände des betrachteten Bereiches. Berechnungen entstehen dadurch, dass Objekte miteinander und mit dem Benutzer kommunizieren, das heisst,

- sich gegenseitig aufrufen
- Nachrichten austauschen
- neue Objekte erzeugen

Wurzel der Interaktion ist die `public class Main` mit der Methode

`public static void main(String[] args) {...}`

Beispiele

Klasse	Objekt	Datenfeld	Methode
Mensch	`sokrates`	`hungrig`	`denken`
Auto	`herbie`	`tankinhalt`	`fahren`
Planet	`erde`	`masse`	`drehen`
Kreis	`kreis1`	`radius`	`farbeAendern`

Wichtig: Klassen können hierarchisch strukturiert sein!

Lebewesen – Mensch – Berliner
Fahrzeug – Auto – Porsche
usw.

Darüber hinaus gibt es statische Datenfelder (Schlüsselwort static): Diese beschreiben Attribute, die alle Objekte der Klasse gemeinsam haben

Beispiel: `static int anzahl_Finger = 10;`
als Attribut von `class Philosoph`

Solange kein Objekt einer Klasse erzeugt wurde, muss nur der Speicherplatz für statische Variablen angelegt werden. Der Speicherplatz für Objekte wird erst beim Aufruf von `new` reserviert. Mit ***new*** wird (dynamisch) ein neues Objekt erzeugt. Man nennt diesen Vorgang auch “*instanziieren*” und das Objekt eine „Instanz“ der Klasse. z.B.

```
Philosoph aristoteles = new Philosoph();
```

- Dadurch werden die objektspezifischen Datenfelder angelegt, d.h., es wird ausreichend Speicherplatz im Adressraum reserviert.
- Dieser Speicherplatz wird automatisch wieder freigegeben, wenn das Objekt nicht mehr erreichbar ist („garbage collection“)
- Primitive Datenobjekte (Zahlen oder Zeichen) müssen nicht erzeugt werden, sie existieren sowieso.

Vom **Instanziieren** zu unterscheiden ist das ***Initialisieren*** von Objekten: den Datenfeldern werden Anfangswerte zugewiesen; z.B., int x = 0;

Eine nichtinitialisierte Variable enthält u.U. einen unvorhersagbaren Wert (was zufällig an dieser Stelle im Speicher stand…)

Für die Erzeugung von neuen Objekten kann jede Klasse Konstruktor-Methoden zur Verfügung stellen, die genauso wie die Klasse selbst heißen (im Beispiel `Philosoph()`), und in denen die Datenfelder der Klasse initialisiert werden können.

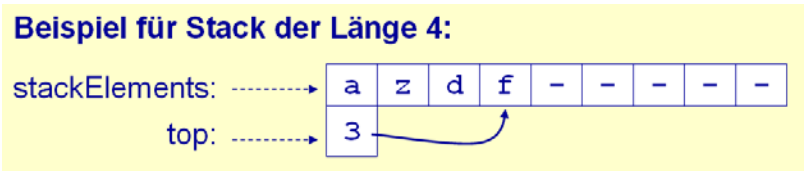
```
        void Philosoph (){
              hungrig = true;
        }
```

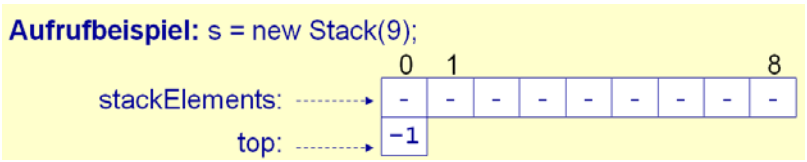
Der Konstruktor wird bei der Erzeugung eine Instanz der Klasse automatisch aufgerufen und dient dazu, das betreffende Objekt zu initialisieren. Der Konstruktor hat denselben Namen wie die Methode selbst und kann auch Eingabeparameter enthalten, liefert jedoch kein Ergebnis.

```
      class Student { ...
      String matrikelnummer;
      int scheine;
      Student (String matrNr){           // Konstruktor- Methode
        matrikelnummer = matrNr; ...
        scheine = 0;
      }
      ...
```

Klassen dienen zur Realisierung von abstrakten Datentypen. Dabei sind nur die Zugriffsfunktionen (in der Signatur) sichtbar („public“). Beispiel: Klasse „Stack“ zur Realisierung von Stapeln (nach Bothe):

```
class Stack {
    private char[] stackElements;
    private int top;
    public Stack(int n) {
        stackElements = new  char [n];
        top = -1; }
    public boolean isempty() {
        return top == -1; }
    public void push(char x) { // Methode
        if (top + 1 == stackElements.length){
          System.out.println (“Stack ist voll”);
          return;
        }
        top++; stackElements[top] = x; }
    public char top() {
        if (isempty()) {
          System.out.println("Stack leer");
          return ' ';
        } else
          return stackElements [top];
    }
    public void pop() {
        if (isempty())
          System.out.println("Stack leer");
        else
          top--;
    }
}
```





Anwendung der Klasse: z.B. zum Überprüfen von Klammerstrukturen:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Klammerstruktur {
  public static void main(String[] args) {
    System.out.println("Geben Sie eine Klammerstruktur ein");
    BufferedReader inputReader =
      new BufferedReader(new InputStreamReader(System.in)) ;
    String eingabeZeile = "";
    try {eingabeZeile = inputReader.readLine();}
      catch (IOException e){}
    int n = eingabeZeile.length();
    Stack s = new Stack(n);
      for (int i = 0; i < n; i++) {
        char ch = eingabeZeile.charAt(i);
          if (ch == '(' || ch == '{' || ch == '[')
            s.push(ch);
          else if (ch==')' && s.top() =='(' ||
            ch=='}' && s.top() =='{' ||
            ch==']' && s.top() =='[')
              s.pop();
          else if (ch==')' || ch=='}' || ch==']' ) {
            System.out.println("schließende Klammer passt nicht"); return;
          }
        }
        if (!s.isempty())
          System.out.println("schließende Klammer fehlt");
        else
          System.out.println("Klammerstruktur korrekt");
  }
}
```

(Achtung: Auch hier prüfen wir nicht ob der Keller leer ist.)
Anwendungsbeispiel ist z.B. der Term (()([]{})) oder der Term

```
if ( x < ( a [ ( i ) ] / 2 )) { x += f () ;}
```

Natürlich kann man auch mehrere Objekte einer Klasse erzeugen:

```
Stack s7 = new Stack(7);
Stack s9 = new Stack(9);
```

Dadurch werden verschiedene Instanzen der Klasse, jeweils mit eigenen Instanzvariablen und –methoden, erzeugt. Der Zugriff erfolgt über Qualifikatoren:

```
s7.push('a');
s9.push('b');
```

Modifikatoren:
Klassen, Datenfelder und Operationen können durch *Modifikatoren* näher attributiert werden. Diese betreffen vorwiegend die Zugriffsmöglichkeiten auf die betreffende Komponente. Folgende Modifikatoren sind erlaubt:

- ***final*** (für ein *Datenfeld*): Das Datenfeld ist eine Konstante, hat also für alle Objekte der Klasse stets den gleichen, unveränderlichen Wert.
- ***final*** (für eine *Operation*): Die Operation kann in Unterklassen nicht verändert (überschrieben) werden.
- ***static*** : Die Komponente ist *klassenbezogen*, d.h. für alle Objekte der Klasse in gleicher Weise verfügbar und wertgleich. Für eine Operation bedeutet das: Die betreffende Operation darf nur ***static***-Datenfelder benutzen.
- ***private*** : Die Komponente darf ***nur innerhalb der aktuellen Klasse*** benutzt werden (aber von beliebigen Objekten der Klasse).
- ***public*** : Die Komponente ist ***öffentlich***, darf also unbeschränkt von außen benutzt werden.
- ***protected*** : Auf die Komponente kann man ***nur innerhalb des Pakets*** zugreifen, das die betreffende Klasse enthält, und zusätzlich von deren Unterklassen.
- ***package*** (default-Einstellung): Zugriff wie bei ***protected***, aber auf das ***aktuelle Paket*** beschränkt.

Auch bei der Verwendung von Modifikatoren sollte man die Sichtbarkeitsregel „so lokal wie möglich, so global wie nötig“ berücksichtigen.

7.3 Vererbung, Polymorphismus, dynamisches Binden

Beim Vergleich von Ingenieurswesen und Software-Engineering fällt auf, dass Ingenieure ihren Gegenstandsbereich hierarchisch strukturieren. Ein Beispiel aus dem Maschinenbau ist die hierarchische Strukturierung eines KFZ gemäß dem Aufbau:

Fahrzeug – Getriebe – Zahnrad.

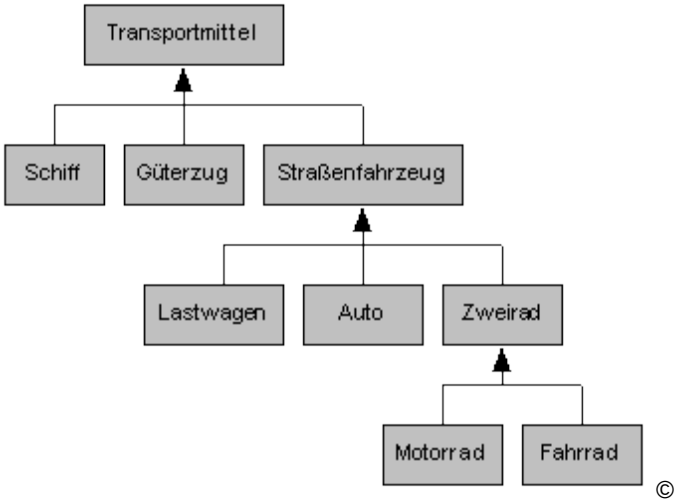
Innerhalb der Betrachtungsebene „Zahnrad“ gibt es eine Typenreihen-Strukturierung gemäß der Weltsicht („Ontologie“): Es gibt runde und ovale Zahnräder, Zahnräder bei denen die Zähne innen oder außen sind, usw. Auf der Betrachtungsebene „Fahrzeug“ gibt es die Spezialisierungen (z.B.) Volkswagen, Nutzfahrzeuge, Caddy, Life, TDI 1.9 DPF.
Die Einteilung entspricht dem, was man in der Informatik „*Vererbungsstruktur*“ nennt; es gibt verschiedene Zahnrad-Bautypen, die sich voneinander im Detail unterscheiden; es gibt aber auch gewisse Gemeinsamkeiten, die uns veranlassen, von einem „Zahnrad“ zu sprechen. Genauso: es gibt verschiedene KFZ-Marken, aber alle haben 4 Räder und einen Hubraum.
Wenn ein Ingenieur ein Getriebe konstruiert, überlegt er sich eine Dekomposition in Komponenten (bzw. einen Bauplan zum Zusammensetzen aus Einzelteilen) und schlägt in Katalogen mit Zahnkranz-Bautypen nach oder modifiziert bestehende Bautypen geeignet.
In der objektorientierten Programmierung besteht Softwaredesign aus der Dekomposition des Problems in einzelne Klassen (bzw. der Strukturierung in eine Klassenhierarchie) und dem Zusammenfügen bzw. der Modifikation vorhandener Bibliotheksklassen. Genau wie ein Fahrzeug aus vielen Einzelteilen besteht, ist objektorientierte Software aus vielen Klassen zusammengesetzt.
Technisch bedeutet dies, dass Klassendefinitionen auf verschiedene Weise hierarchisch strukturiert sein können:

- Klassen dürfen *lokale Klassen* als Deklaration enthalten
- Klassen können zu *Paketen* zusammengefasst werden
- Zwischen Klassen kann eine *Erbschaftsbeziehung* (Super/Subklassenhierarchie) bestehen

- Zwischen abstrakten und konkreten Klassen kann eine *Implementierungsrelation* bestehen

Vererbung ist also **das** fundamentale Konzept der oo- (objektorientierten) Programmierung.

Als Beispiel betrachten wir eine Hierarchie von Fahrzeugtypen:



© http://www.javabuch.de/

Diese Hierarchie könnte objektorientiert etwa wie folgt nachempfunden werden:

```
public class Fahrzeug {
  private int tachostand;
  public Fahrzeug() {
    tachostand = 0;
  }
  public int gibTachostand() {
    return tachostand;
  }
  public void fahre(int strecke) {
    tachostand += strecke;
  }
}
```

Eine Erweiterung dieser Klasse wäre dann etwa:

```
class LKW extends Fahrzeug {...}
```

Durch diese Definition sind alle Datenfelder und Methoden von `Fahrzeug` auch für `LKW` verfügbar (Ausnahme: private Datenfelder und Methoden). Wir sagen, dass die Klasse `LKW` die Attribute der Klasse `Fahrzeug` *erbt*. `LKW` kann darüber hinaus zusätzliche Variablen oder Methoden enthalten oder Methoden abändern

- z.B. `nutzlast` oder `transportiere`
- z.B. `fahre` mit zusätzlichem Argument `ladung`

Der Zugriff auf die Methoden erfolgt mit Member-Selektor (Punkt), z.B. `lkw1.ladung`

Polymorphismus

Auf oberster Ebene gilt: Jede Klasse ist von der allgemeinsten Klasse Object abgeleitet und erbt von dieser gewisse Eigenschaften

- `Object clone()`
- `boolean equals(Object)`
- `String toString()`
- `...`

In den meisten Fällen muss jedoch die generische („ererbte“) Definition der Methoden abgeändert werden. In abgeleiteten Klassen dürfen alle ererbten Methoden neu definiert werden (Ausnahmen: private, final). Dies realisiert den Umstand, dass im Speziellen zusätzliche Maßnahmen gegenüber dem Allgemeinen durchgeführt werden.
Beispiel: ein LKW kann nicht rückwärts fahren, daher prüft fahre zunächst ob das Argument positiv ist:

```
public void fahre(int strecke){
  if (strecke > 0) super.fahre(strecke);
}
```

Die speziellere Methode *überlagert* die allgemeinere aus der übergeordneten Klasse. Unter *Polymorphie* (griechisch: „Vielgestaltigkeit“) versteht man die Tatsache, dass unterschiedliche Methoden in verschiedenen Klassen oder auch Methoden mit verschiedenen Parametern innerhalb einer Klasse gleich bezeichnet sein dürfen. Beispiele:

- \+ für Int-, Float-Addition, String-Konkatenation
- neue Methode `fahre` in Klasse LKW mit zusätzlichem Argument `last` prüft zunächst ob die angegebene Nutzlast überschritten wurde:

  ```
  public void fahre(int strecke, int last) {
    if (last > nutzlast)
      System.out.println("überladen!");
    else fahre(strecke); }
  ```

In jedem Fall muss an Hand der Signatur oder des Objekttyps entscheidbar sein, welche Methode gemeint ist.

```
public class LKW extends Fahrzeug {
  private int nutzlast;
  public LKW(int nutzlast){
    this.nutzlast = nutzlast;
  }
  public void fahre(int strecke){
    if (strecke > 0) super.fahre(strecke);
  }
  public void fahre(int strecke, int last) {
    if (last > nutzlast)
      System.out.println("überladen!");
    else fahre(strecke);
  }
  public void transportiere (int last, int start, int ziel)
  { /* belade(last); fahre(ziel - start, last); */ }
}
```

Dynamisches Binden
Es ist nicht immer statisch entscheidbar, welche Methode gerade gemeint ist:

```
Fahrzeug kfz = new LKW(...);
kfz.fahre (... ladung); ...
```

```
if (...) kfz = new PKW();
kfz.fahre(... personen);
```

Das bedeutet, dass erst zur Laufzeit entschieden werden kann, welche Methode jetzt eigentlich gemeint ist. Diesen Effekt nennt man „dynamisches Binden“. Zur Realisierung erzeugt der Compiler zusätzlichen Code zur Prüfung (Ausnahmen: private, static, final).

```
public class Fuhrpark {
  private static Fahrzeug f;

  public static void sampleMethod() {
    f = new LKW(20000); // ein 20-Tonner
    f.fahre(-100); // welches fahre?
    //f.fahre(100, 10000); // nicht erlaubt
    f = new LKW(4); // ein 4-Sitzer
    ((LKW)f).fahre(123,5);
  }
}
```

Bindungsregeln und Objektinitialisierung

- Beim Erzeugen eines Objektes mittels `new` werden zunächst die Konstruktoren der vererbenden Klassen (Eltern, Großeltern etc.) aufgerufen.
- Die eigenen Merkmale können mit dem Deskriptor `this`, die der jeweiligen Elternklasse mit super aufgerufen werden.
- Eventuelle Typumwandlung (*Casting*) durch vorgestelltes `(Typ)`