Beispiel für überladene Methode

```
class Gras extends Futter { ... }

abstract class Tier {
    public abstract void friss (Futter x);
}

class Rind extends Tier {
    public void friss (Gras x) { ... }
    public void friss (Futter x) {
        if (x instanceof Gras) friss ((Gras)x);
        else erhoeheWahrscheinlichkeitFuerBSE();
    }
}
```

Überladen = statisches Binden

nur deklarierte Typen für Überladen entscheidend:

```
Rind   rind = new Rind();
Futter gras = new Gras();
rind.friss(gras);          // Rind.friss (Futter x)
rind.friss((Gras)gras);    // Rind.friss (Gras x)
```

Achtung: deklarierten Typ von `rind` betrachten:

```
Tier   rind = new Rind();
Futter gras = new Gras();
rind.friss(gras);          // Rind.friss (Futter x)
rind.friss((Gras)gras);    // Rind.friss (Futter x) !!
```

Empfehlungen für Überladen

- Überladen generell fehleranfällig $\Rightarrow$ eher vermeiden
- Überladen in verschiedenen Klassen schwer sichtbar $\Rightarrow$ Überladen von Methoden aus Oberklasse stets vermeiden
- Unterscheidung zwischen statischem und dynamischem Binden soll nicht entscheidend sein
  $\Rightarrow$ für je zwei überladene Methoden soll gelten:
  - Unterscheidung an Hand einer Parameterposition, wobei Parametertypen keinen gemeinsamen Untertyp haben
  - oder alle Parametertypen einer Methode spezieller als die der anderen, und allgemeinere Methode verzweigt nur auf speziellere, falls möglich

Multimethoden = dynamisches Binden

Multimethoden entsprechen syntaktisch überladenen Methoden, aber Bindung erfolgt an Hand dynamischer Typen

dadurch oft einfachere Programme möglich:

```
class Rind extends Tier {
    public void friss (Gras x) { ... }
    public void friss (Futter x) {
        erhoeheWahrscheinlichkeitFuerBSE();
    }
}
```

Achtung: NICHT in Java !!!
(in Java nur Überladen mit statischem Binden)

Simulation von Multimethoden (1)

```
abstract class Tier {
    public abstract void friss (Futter futter);
    ...
}
class Rind extends Tier {
    public void friss (Futter futter) {
        futter.vonRindGefressen(this);
    }
}
class Tiger extends Tier {
    public void friss (Futter futter) {
        futter.vonTigerGefressen(this);
    }
}
```

Simulation von Multimethoden (2)

```
abstract class Futter {
    public abstract void vonRindGefressen (Rind rind);
    public abstract void vonTigerGefressen (Tiger tiger);
}
class Gras extends Futter {
    public void vonRindGefressen (Rind rind) { ... }
    public void vonTigerGefressen (Tiger tiger)
        { tiger.fletscheZaehne(); }
}
class Fleisch extends Futter {
    public void vonRindGefressen (Rind rind)
        { rind.erhoeheWahrscheinlichkeitFuerBSE(); }
    public void vonTigerGefressen (Tiger tiger) { ... }
}
```

Komplexität von Multimethoden

- Nachteil simulierter Multimethoden: Anzahl der Methoden

  M Tierarten, N Futterarten $\Rightarrow M \cdot N$ inhaltliche Methoden

  Generell für n Bindungen: $N_1, N_2, \ldots, N_n$ Möglichkeiten
  $\Rightarrow N_1 \cdot N_2 \cdots N_n$ inhaltliche Methoden
  insgesamt $N_1 + N_1 \cdot N_2 + \cdots + N_1 \cdot N_2 \cdots N_n$ Methoden

- echte Multimethoden verwenden daher Komprimierungstechniken und Vererbung

  Eindeutigkeit bei Vererbung muss garantiert werden:

```
void frissDoppelt (Futter x, Gras y) {...}
void frissDoppelt (Gras x, Futter y) {...}
void frissDoppelt (Gras x, Gras y) {...}   // notwendig
```

Nebenläufige Programmierung

- mehrere gleichzeitig ablaufende (= nebenläufige) Threads
- gleichzeitige und überlappte Zugriffe auf Variablen
  $\rightarrow$ Fehler wenn inkonsistente Zustände existieren (häufig!)
- *Synchronisation* soll gleichzeitige und überlappte Zugriffe vermeiden während Zustände inkonsistent sein können
- Programmierer muss sich um Synchronisation kümmern
- Synchronisation ist häufige Fehlerquelle

Beispiel für fehlende Synchronisation

```
public class Zaehler {
    private int i = 0, j = 0;
    public void schnipp() { i++; j++; }
}
```

- überlappte Aufrufe von `schnipp` in mehreren Threads
  $\rightarrow$ möglicherweise `i` $\neq$ `j`
- auch bei nur einer Variablen werden Aufrufe „vergessen“
- Problem bei einfachem Testen kaum feststellbar

Einfache Synchronisation in Java

```
public class Zaehler {
    private int i = 0, j = 0;
    public synchronized void schnipp() { i++; j++; }
}
```

- während der Ausführung einer `synchronized` Methode kann kein anderer Thread eine `synchronized` Methode desselben Objekts ausführen – „mutual exclusion“
- weitere (fast) gleichzeitige Aufrufe von `schnipp` blockiert $\rightarrow$ andere Threads warten bis ausgeführte Methode fertig
- `synchronized` Methoden sollen nur kurz laufen

Synchronisierte Blöcke in Java

```
public class Zaehler {
    private int i = 0, j = 0;
    public synchronized void schnipp() {
        synchronized(this) { i++; }
        synchronized(this) { j++; }
}   }
```

- kurzfristig `i` $\neq$ `j` möglich, aber kein Aufruf „vergessen“
- *Lock* für Thread auf Argument von `synchronized` (= `this`)
- nur dieser Thread darf auf `synchronized` Blöcke und Methoden desselben Objekts zugreifen, andere müssen warten
- `synchronized` Methoden setzen Lock immer auf `this`

Threads warten auf Objektzustände

```
public class Druckertreiber {
    private boolean online = false;
    public synchronized void drucke (String s) {
        while (!online) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        ... // schicke s zum Drucker
    }
    public synchronized void onOff() {
        online = !online;
        if (online) notifyAll();
    }
}
```

Erzeugen neuer Threads

```
public class Produzent implements Runnable {
    private Druckertreiber t;
    public Produzent(Druckertreiber _t) { t = _t; }
    public void run() {
        String s = ....
        for (;;) {
            ...           // produziere neuen Wert in s
            t.drucke(s); // schicke s an Druckertreiber
}   }   }
........
        Druckertreiber t = new Druckertreiber(...);
        for (int i = 0; i < 10; i++) {
            Produzent p = new Produzent(t);
            new Thread(p).start();
        }
```

Synchronisation und Bibliotheksklassen

- kein einheitliches Schema – manchmal Client für Synchronisation verantwortlich, manchmal Server

  → Dokumentation (Zusicherungen) sehr wichtig

- z.B. Synchronisation in `Vector` nicht beeinflussbar

- z.B. Client muss für Synchronisation in `List` sorgen:

  `List` von `synchronized` Methoden/Blöcken aus verwenden

  oder Listen folgendermaßen erzeugen:

```
List x = Collections.synchronizedList(new LinkedList(...));
```

Probleme bei Synchronisation

- Synchronisation kann Threads blockieren und dabei die Ausführung von Programmen (sehr stark) verzögern
- „deadlock“ und „livelock“ als Extremfälle
  (= unendliche Verzögerung, „liveness properties“)
- solche Probleme derzeit nur durch Testen auffindbar
- nebenläufige Programmierung schwierig und fehleranfällig
  → auf Einfachheit der Synchronisation achten
  → `wait`, `notify` und `notifyAll` möglichst vermeiden
- Erfahrung wichtig (z.B. spezielle Entwurfsmuster)