Kovariante Probleme – Beispiel (1)

```
abstract class Futter { ... }

class Gras extends Futter { ... }

class Fleisch extends Futter { ... }

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

Kovariante Probleme – Beispiel (2)

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

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 anhand 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
```