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 ⇒ eher vermeiden
- Überladen in verschiedenen Klassen schwer sichtbar ⇒ Überladen von Methoden aus Oberklasse stets vermeiden
- Unterscheidung zwischen statischem und dynamischem Binden soll nicht entscheidend sein
  ⇒ 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!

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!
```

Ausnahmebehandlung in Java

```
class A {
    void foo() throws Help, SyntaxError { ... }
}
class B extends A {
    void foo() throws Help {
        if (helpNeeded())
            throw new Help();
    }
}
...
    try { ... }
    catch (Help e) { ... }
    catch (Exception e) { ... }
    finally { ... }
```

Einsatz von Ausnahmebehandlungen

- Ursachen unvorhergesehener Programmabbrüche finden
  (kaum vermeidbar, außer durch Wiederaufsetzen)

- kontrolliertes Wiederaufsetzen nach Fehlern
  (in der Praxis notwendig, sinnvolles Aufsetzen schwierig)

- vorzeitiger Ausstieg aus Sprachkonstrukten
  (fehleranfällig, vor allem wenn nicht lokal; vermeidbar)

- Rückgabe alternativer Ergebniswerte
  (oft Hinweis auf schlechte Programmstruktur; vermeidbar)

Einsatzbeispiele für Ausnahmen (1)

Ohne Ausnahmebehandlung:

```
while (x != null)
    x = x.getNext();
```

Mit Ausnahmebehandlung:

```
try {
    while (true)
        x = x.getNext();
}
catch (NullPointerException e) {}
```

fehleranfällig, da Ausnahme auch in `getNext` auslösbar

Einsatzbeispiele für Ausnahmen (2)

trickreiche Verwendung von Ausnahmen:

```
                                  try { throw x }
if (x instanceof T1) {...}        catch (T1 x) {...}
else if (x instanceof T2 {...}    catch (T2 x) {...}
...                               ...
else if (x instanceof Tn {...}    catch (Tn x) {...}
else {...}                        catch (Exception x) {...}
```

keine nicht-lokalen Ausnahmen (daher weniger fehleranfällig)

aber beide Varianten schwer wartbar

Einsatzbeispiele für Ausnahmen (3)

Ohne Ausnahmebehandlung:

```
public static String addA (String x, String y) {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else return "Error";
}
```

Mit Ausnahmebehandlung:

```
public static String addB (String x, String y)
        throws NoNumberString {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else throw new NoNumberString();
}
```

sinnvoller Einsatz, da Fehlerabfragen vermieden werden

Iterator (Cursor)

Zweck: sequentieller Zugriff auf Elemente eines Aggregats

Anwendungsgebiete:

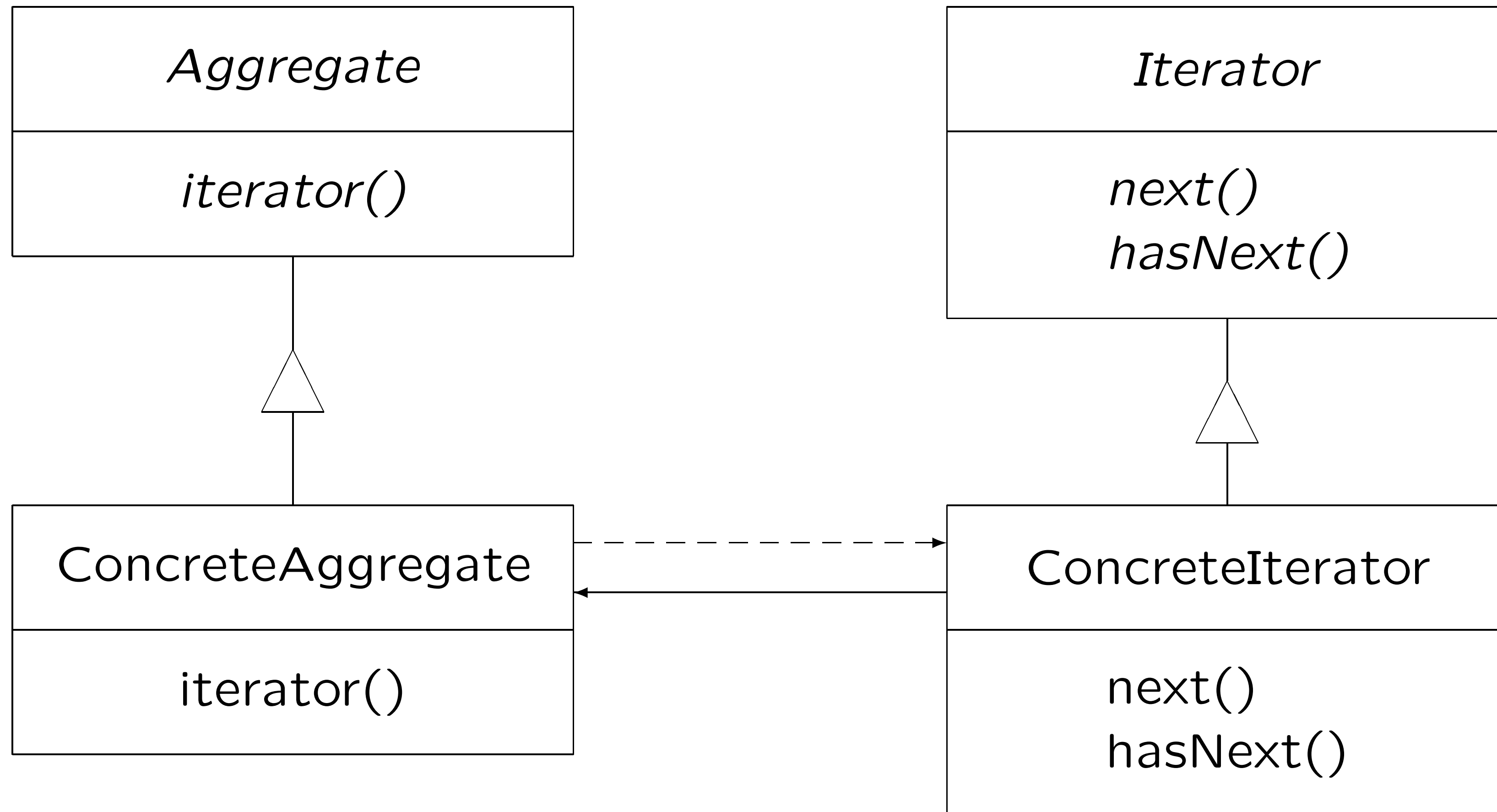
- Zugriff auf Aggregatinhalt

  innere Darstellung bleibt gekapselt

- mehrere Abarbeitungen des Aggregatinhalts

- einheitliche Schnittstelle für Abarbeitung verschiedener Aggregatstrukturen (polymorphe Iterationen)

Iterator: Struktur



Iterator: Eigenschaften

- unterstützen unterschiedliche Arten der Abarbeitung von Aggregaten (mehrere Iteratorklassen pro Aggregatklasse)
- vereinfachen Schnittstelle von „Aggregate“
- mehrere gleichzeitige Abarbeitungen möglich

Iterator: Implementierungshinweise

- externe Iteratoren flexibler, aber weniger einfach

  extern: Anwender holt nächstes Element (siehe Beispiele)

  intern: Iterator wendet Operation auf alle Elemente an

- interne Iteratoren besser wenn Beziehungen zwischen Elementen bei Abarbeitung zu berücksichtigen sind

- Algorithmus zum Durchwandern des Aggregats in Aggregat oder Iterator definiert

- Aggregatänderungen während der Abarbeitung berücksichtigen (robuster Iterator)

- auch auf leeren Aggregaten brauchbar