

Programmierungsmethodik 1

Programmierertechnik

Methoden

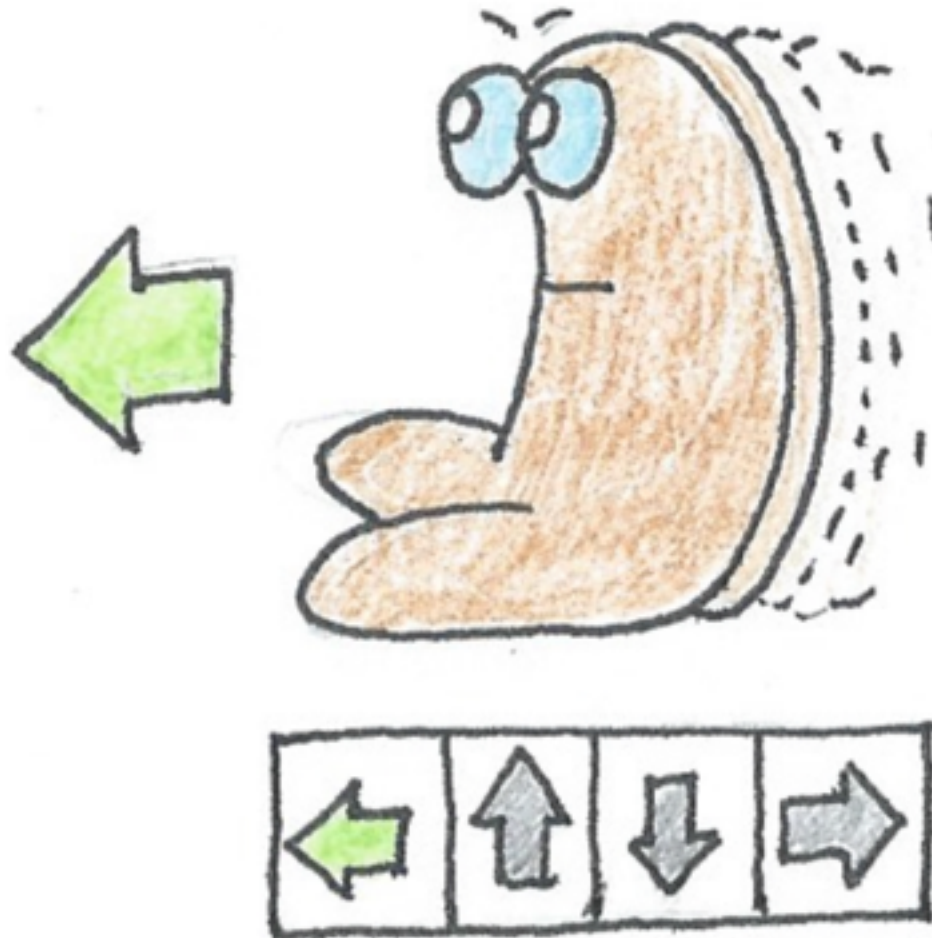
Wiederholung

- Klassen und Objekte
- Referenztypen
- Objektvariablen
- Vergleich und Lebensdauer
- UML

Ausblick



Worum gehts?



Agenda

- Einführung
- Argumente und Parameter
- Überladen
- ErgebnISRückgabe
- UML



Methoden

Methoden

- sind eigenständig benannte und einzeln ausführbare Anweisungsblöcke innerhalb einer Klasse
- werden in Klassen definiert
 - ebenso wie Objektvariablen
- Abgrenzung:
 - Objektvariablen legen Eigenschaften („Attribute“) von Objekten fest
 - Methoden legen Operationen auf diesen Objekten fest
- anders formuliert
 - Objektvariablen beschreiben den Aufbau von Objekten, Methoden ihr Verhalten

Methoden

- Methoden haben Namen, wie Objektvariablen
 - ebenfalls erster Buchstabe klein!
- Methoden beschreiben Abläufe
 - werden mit aussagekräftigen Verben benannt
- Beispiel
 - Methode `print()` der Klasse `Bruch`

```
class Bruch {  
    int zaehler;  
  
    int nenner;  
  
    void print() {  
        System.out.format("%d/%d\n",  
            zaehler, nenner);  
    }  
}
```


Definition

- Syntax
 - Methodenkopf (auch: "Signatur"):
 - <Ergebnistyp> <Methodenname>(<Parameterliste>)
 - Methodenrumpf:

```
{  
    <Anweisung>  
    ...  
}
```
- Sonderfälle
 - Typ void: Keine ErgebnISRückgabe!
 - Parameterliste (): Keine Parameter!
- Beispiel:

```
void print() {  
    System.out.println( ... );  
}
```

Methoden

- Klammern um den Rumpf sind Pflicht
- Methodendefinitionen sind nur in Klassen zulässig
 - nicht außerhalb einer Klassendefinition,
 - nicht innerhalb einer anderen Methodendefinition
- Anzahl, Reihenfolge und Anordnung von Methodendefinitionen in einer Klasse sind beliebig

Aufruf (Ausführung)

- Zielobjekt muss bei Aufruf der Methode angegeben werden
- Methodenaufruf syntaktisch ähnlich zu Objektvariablenzugriff:
 - <Zielobjekt>.<Methodenname>(<Argumente>)
- runde Klammern markieren Methodenaufruf
 - fehlen bei Zugriff auf Objektvariablen
- Beispiel: Bruch initialisieren, dann ausgeben:

```
Bruch bruch = new Bruch();  
bruch.zaehler = 1;  
bruch.nenner = 9;  
bruch.print();
```

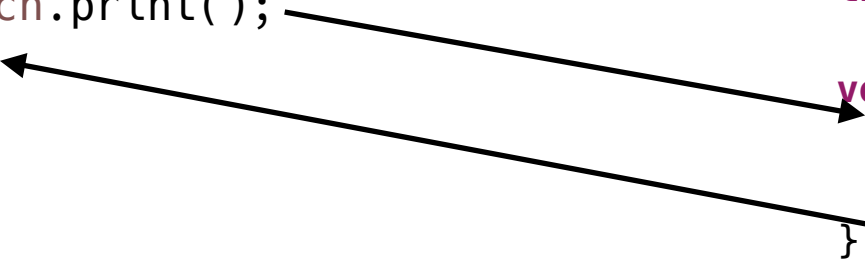
Call-Sequence

- Call-Sequence ist Ablauf eines Methodenaufrufs in mehreren Einzelschritten
- Ablauf der Call-Sequence:
 - Aufrufendes Programm („Aufrufer“, engl. caller) unterbrechen
 - Methodenrumpf durchlaufen
 - Aufrufer nach dem Aufruf fortsetzen
- mehrere Aufrufe
 - Aufrufer wird jedes Mal unterbrochen, immer derselbe Methodenrumpf wird ausgeführt

Call-Sequence

```
Bruch bruch = new Bruch();  
bruch.zaehler = 5;  
bruch.nenner = 10;  
bruch.print();
```

```
class Bruch {  
    int zaehler;  
  
    int nenner;  
  
    void print() {  
        System.out.format("%d/%d\n",  
                           zaehler, nenner);  
    }  
}
```

The diagram consists of two black arrows. The first arrow originates from the `bruch.print();` line in the first code block and points to the `void print() {` line in the second code block. The second arrow originates from the closing curly brace of the `print()` method in the second code block and points back to the `bruch.print();` line in the first code block, indicating the return path.

Methoden

- Methodenrumpf = Block
- Gültigkeitsbereich lokaler Deklarationen = Methodenrumpf
- Lebensdauer lokaler Variablen
 - jeweils ein Aufruf einer Methode
 - Gegensatz Objektvariablen: Lebensdauer wie Objekt
- Beispiel: Methode vereinfache() zum Kürzen eines Bruchs:

```
void vereinfache() {  
    int gcd = berechneGgt(zaehler, nenner);  
    zaehler /= gcd;  
    nenner /= gcd;  
}
```

Zugriff aus einem Methodenrumpf

- Zugriff auf Objektvariablen des eigenen Objektes
 - Angabe eines Zielobjekts nicht nötig
- Beispiel
 - vereinfache(): Objektvariablen zaehler, nenner wie lokale Variablen ansprechbar
- ebenso: Aufruf von Methoden des eigenen Objektes ohne Angabe eines Zielobjektes
- Methoden erreichen jede Objektvariable der eigene Klasse
 - unabhängig von der Anordnung der Definitionen

Namenskollisionen

- Namen von lokalen Variablen und Objektvariablen kollidieren nicht
- Nachteil
 - lokale Variablendeklaration „verdeckt“ eine gleichnamige Objektvariable
- Vorteil
 - Benennung von lokalen Variablen ohne Rücksicht auf Objektvariablen möglich

Beispiel: Namenskollisionen

```
public class BeispielNamensKollision {
```

```
    int variable = 23;
```

Objektvariable

```
    void methode() {
```

```
        int variable = 42;
```

lokale Variable

```
        System.out.println(variable);
```

```
        System.out.println(this.variable);
```

```
    }
```

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

```
        BeispielNamensKollision nce = new BeispielNamensKollision();
```

```
        nce.methode();
```

```
    }
```

```
}
```

Bindung von "innen-nach-außen", also lokale Variable

this = aktuelles Objekt, also Objektvariable

Selbstreferenz

- reserviertes Wort `this` ist eine Referenz auf das eigene Objekt
 - liefert das eigene Objekt als Zielobjekt
- automatisch definiert, immer verfügbar
- nützlich u.a. um verdeckte Objektvariablen zu erreichen

Übung: Methoden

- Schreiben Sie eine Methode verdopple(), die den Wert des Bruchs verdoppelt



Argumente und Parameter

Argumente und Parameter

- Parameter dienen zur Übergabe von Daten vom Aufrufer an die Methode
- zwei Sprachelemente sind gekoppelt:
 - Die Methode definiert Parameter (Übergabe-Variablen)
 - Der Aufrufer liefert Argumente (Werte) für die Parameter
- Methodenkopf-Definition mit ausführlicher Parameterliste:
`<Ergebnistyp> <Methodenname>(<Typ1> <Variablenname1>, <Typ2>
<Variablenname2>, ...)`
- Methodenaufruf-Syntax mit Argumenten
`<Zielobjekt>.<Methodenname>(<Argument1>, <Argument2>, ...)`

Beispiel für Parameter

- Methode erweitere zum Erweitern eines Bruchs mit Parameter faktor
 - faktor: Faktor, mit dem Zähler und Nenner erweitert werden sollen
- Der Aufrufer muss bei jedem Aufruf ein kompatibles Argument angeben

```
bruch.print(); // liefert 5/9  
bruch.erweitere( 2 );  
bruch.print(); // liefert 10/18
```

```
void erweitere(int faktor) {  
    zaehler *= faktor;  
    nenner *= faktor;  
}
```

Parameterübergabe

- Parameter und Argumente werden vom Compiler bei jedem Aufruf paarweise abgeglichen
 - pro Parameter ist ein Argument (Wert) erforderlich
 - zu viele oder zu wenige Argumente: wird nicht übersetzt
 - beliebig komplizierte Ausdrücke sind als Argumente zulässig
 - diese werden erst ausgewertet, dann wird der Ergebnis-Wert übergeben
 - Typ jedes Arguments muss kompatibel zum entsprechenden Parameter sein
- Verwendung der Parameter im Methodenrumpf
 - genauso wie (automatisch initialisierte) lokale Variablen
- Parameter
 - dritte Art von Variablen, neben lokalen Variablen und Objektvariablen

Call-Sequence mit Parametern

- Erweiterung der einfachen Call-Sequence parameterloser Methoden
- Einzelschritte beim Aufruf einer Methode:
 - Werte aller Argumente von links nach rechts berechnen
 - Parameter erzeugen (lokale Variablen!)
 - Parameter mit Argumentwerten initialisieren
 - Aufrufendes Programm („Aufrufer“) unterbrechen
 - Methodenrumpf durchlaufen
 - Parameter zerstören (lokale Variablen!)
 - Aufrufer nach dem Aufruf fortsetzen

Mehrere Parameter

- Klasse Bruch:

```
void initialisiere(int zaehler, int nenner) {  
    this.zaehler = zaehler;  
    this.nenner = nenner;  
}
```

- Aufruf mit passender Anzahl an Argumenten:

```
Bruch bruch = new Bruch();  
bruch.initialisiere(18, 24);
```

- Unzulässige Aufrufe

```
bruch.initialisiere(18);  
bruch.initialisiere(18, 24, 42);
```

Primitive Typen als Parameter

- versteckte Wertzuweisung bei der Parameterübergabe
 - Initialisierung von Variablen
 - Werte primitiver Typen werden kopiert
- implizite und explizite Typumwandlungen wie bei "normalen" Wertzuweisungen
- Beispiele

```
bruch.erweitere( 3.14 ); // Fehler: falscher Typ  
bruch.erweitere( (int)3.14 ); // ok
```

Referenztypen als Parameter

- Referenztypen sind als Parameter zulässig
- Beispiel
 - Methode addiereDazu erwartet anderes Bruch-Objekt als Parameter, addiert this zu dem Parameterobjekt

```
void addiereDazu(Bruch andererBruch) {  
    zaehler = zaehler * andererBruch.nenner +  
        andererBruch.zaehler * nenner;  
    nenner = nenner * andererBruch.nenner;  
    vereinfache();  
}
```

- aus der Sicht von addiereDazu ist andererBruch ein anderes Objekt
- Ansprechen der eigenen Objektvariablen ohne Zielobjekt
- Ansprechen der fremden Objektvariablen mit Zielobjekt andererBruch

Aliasing bei Referenzparametern

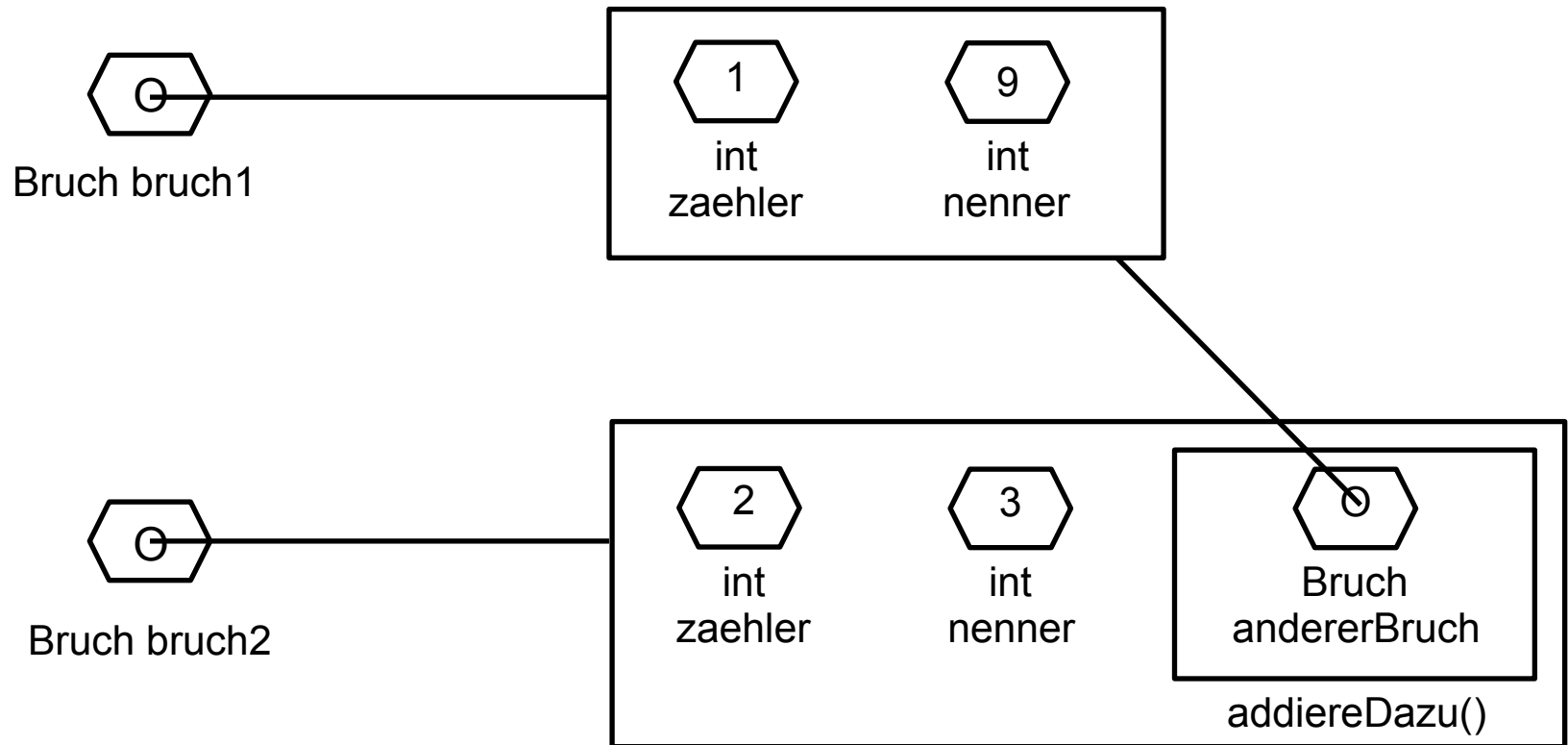
- Nicht das Objekt des Aufrufers, sondern Referenz (Zeiger) wird kopiert
 - daher: Aliasing - mehrere Referenzen auf dasselbe Objekt - bei der Übergabe von Objekten
 - wie bei Wertzuweisungen von Referenztypen

- Beispiel:

```
Bruch bruch1 = new Bruch();  
Bruch bruch2 = new Bruch();  
bruch1.initialisiere(2, 3);  
bruch2.initialisiere(1, 9);  
bruch1.addiereDazu(bruch2);
```

- im Rumpf von addiereDazu: Argument des Aufrufers (bruch2) und der Parameter der Methode (andererBruch) referenzieren dasselbe Objekt

Beispiel: Eintritt in die addiereDazu()-Methode



Seiteneffekte

- addiereDazu liest Objektvariablen des Parameterobjektes, verändert aber nur eigene Objektvariablen
- böswillige Version von addiereDazu()
 - schreibt in das Parameterobjekt!

```
void addiereDazu (Bruch andererBruch) {  
    ...  
    andererBruch.zaehler = 0;  
}
```

- für den Aufrufer nicht erkennbar: Methodenaufruf verändert das Argument!

```
bruch2.print(); // 1/9  
bruch1.addiereDazu(bruch2);  
bruch2.print(); // Nenner von s ist jetzt 0
```

- also: schreibende Zugriffe auf fremde Objektvariablen vermeiden

Übung: Parameter

- Schreiben Sie eine Methode `subtrahiereDavon()`.
- Die Methode hat einen Parameter (`andererBruch`) vom Typ `Bruch`.
- In der Methode sollen sie beiden Brüche subtrahiert werden, das Ergebnis überschreibt den `Bruch` selbst.



Überladen von Methoden

Überladen von Methoden

- engl. overloading
- mehrere Methoden mit gleichem Namen, aber unterschiedlichen Parameterlisten
 - entscheidend: unterschiedliche Parameteranzahl und/oder Typ
 - Namen der Parameter sind ohne Bedeutung
- Überladen ist zulässig
 - sinnvoll für verwandte Methoden mit ähnlichem Zweck

Überladen von Methoden

- mehrere Methoden initialisiere mit gleichem Bezeichner zur Wertzuweisung an einen Bruch

```
void initialisiere(int zaehler, int nenner) {  
    this.zaehler = zaehler;  
    this.nenner = nenner;  
}
```

```
void initialisiere(int wert) {  
    this.zaehler = wert;  
    this.nenner = 1;  
}
```

Überladen von Methoden

- Aufruf
- die passende überladene Methode wird aufgrund der Argumentliste des Aufrufers ausgewählt
- Beispiele:

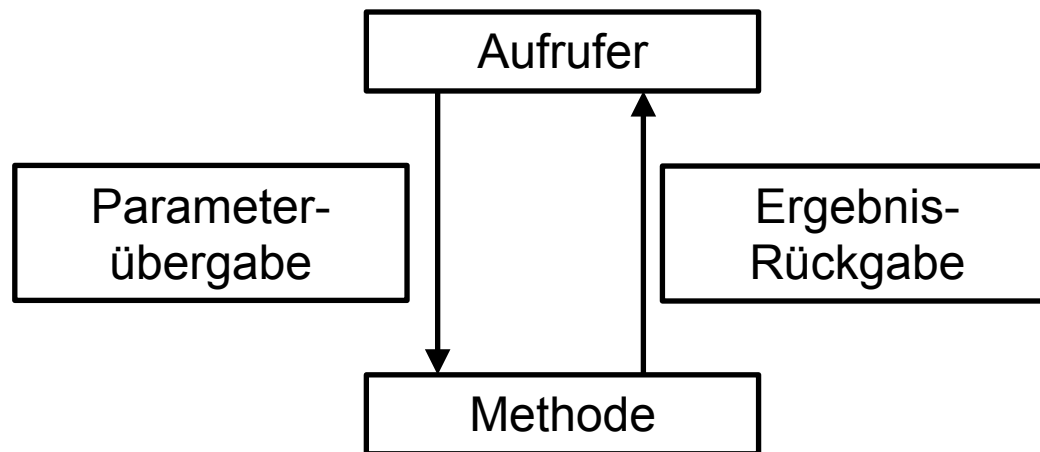
```
bruch.initialisiere(2); // → initialisiere(int)
bruch.initialisiere(2, 1); // → initialisiere(int, int)
bruch.initialisiere(2, 1, 0); // Fehler
```
- überladene Methoden führen zu Polymorphismus



Ergebnisrückgabe

Ergebnisrückgabe

- Parameterübergabe transportiert Information vom Aufrufer zur Methode
- Ergebnisrückgabe liefert Information von der Methode zurück zum Aufrufer



- eine Methode kann beliebig viele Parameterwerte annehmen, aber nur einen Ergebniswert liefern

Ergebnisrückgabe

- Definition der Ergebnisrückgabe findet im Rahmen der Methodendefinition statt:
 - Typ des Ergebniswertes wird im Methodenkopf definiert
 - vor dem Methodennamen
 - return-Anweisung im Methodenrumpf beendet die Methode sofort und liefert den Ergebniswert an den Aufrufer

- Syntax:

```
<Ergebnistyp> <Methodenname> (...) {  
    ...  
    return <Ausdruck>;  
}
```

- Typ von <Ausdruck> in der return-Anweisung muss kompatibel zu <Ergebnistyp> im Methodenkopf sein
- Ergebniswert, den der Methodenaufruf liefert, kann in beliebigen Ausdrücken verwendet werden

Ergebnisrückgabe

- Beispiel: Berechne die Gleitkommadarstellung des Bruchs und liefere sie zurück

```
double getWert() {  
    return (double) zaehler / (double) nenner;  
}
```

Ergebnisrückgabe

- mehrere return-Anweisungen sind im Rumpf erlaubt
- Methode wird sofort beendet, sobald zur Laufzeit die erste return-Anweisung erreicht wird
- statische Reihenfolge der return-Anweisungen ist unerheblich, konkreter Ablauf zur Laufzeit entscheidet

Ergebnislose Methoden

- Rückkehr ohne Ergebnis: Angabe des Pseudo-Typs void
 - überhaupt kein Wert
- automatische Rückkehr am Ende des Methodenrumpfes oder Rückkehr mit return-Anweisung ohne Ausdruck
- Beispiel:

```
void initialisiere(int zaehler, int nenner) {  
    this.zaehler = zaehler;  
    this.nenner = nenner;  
}
```

Ergebnisrückgabe

- Bei überladenen Methoden
- der Ergebnistyp wird beim Überladen von Methoden ignoriert
- Überladen mit unterschiedlichem Ergebnistyp bei gleichen Parameterlisten ist daher unzulässig!
- Beispiel:

```
int getZaehler() {  
    ...  
}
```

```
double getZaehler() {  
    ...  
}
```

Übung: Ergebnissrückgabe

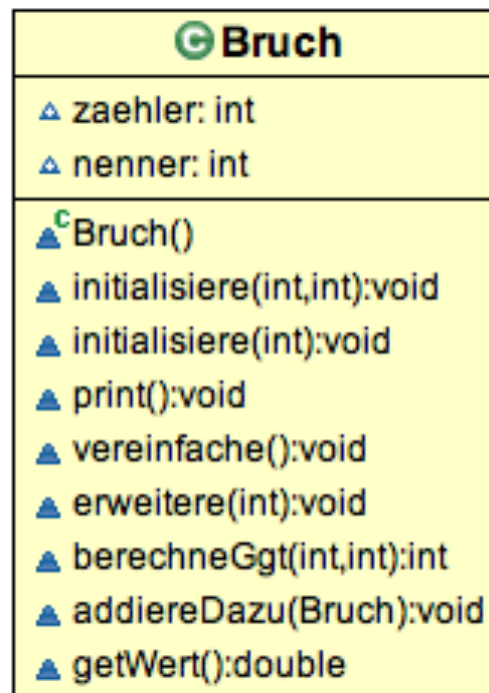
- Schreiben Sie eine Methode `istKleiner` mit zwei Parametern vom Typ `int`: `zaehler`, `nenner`
- Die Methode soll einen Wahrheitswert zurückliefern
 - wahr, wenn der Bruch selbst kleiner ist, als der Bruch, der sich aus den Parametern ergibt
 - falsch, wenn der Bruch selbst größer/gleich ist, als der Bruch, der sich aus den Parametern ergibt
- Schreiben Sie eine zweite Methode `istKleiner`, die nur einen Parameter für den Zähler hat, der Nenner wird als 1 angenommen.
 - Verwenden Sie die erste Methode zur Implementierung der zweiten



UML

UML

- Methoden-Signatur im dritten Block des UML-Klassen-Diagramms
- keine Rümpfe
- Beispiel:



Zusammenfassung

- Methoden
- Argumente und Parameter
- Überladen
- ErgebnISRückgabe
- UML