Softwareentwicklung I (IB)

Methoden

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik
Hochschule München

Letzte Änderung: 18.03.2018 20:09

Inhaltsverzeichnis

Motivation

- **Methoden** werden in Klassen definiert, ebenso wie Objektvariablen
- Objektvariablen legen Eigenschaften („Attribute“) von Objekten fest, Methoden legen Operationen fest
- Anders formuliert: Objektvariablen beschreiben den Aufbau von Objekten, Methoden ihr Verhalten

Definition

- Methoden haben Namen, wie Objektvariablen
- Beispiel: Methode `print` der Klasse `Rational`:

```
class Rational {
    int numer;
    int denom;

    void print() {
        System.out.printf("%d/%d%n", numer, denom);
    }
}
```

- Methoden entsprechen Abläufen ⇒ mit (englischen) Verben benannt

Signatur und Rumpf

- Methodendefinition = (Methoden-)Kopf + (Methoden-)Rumpf

```
void print() /* bis hier: Kopf */ {
    // Rumpf
    System.out.printf("%d/%d%n", numer, denom);
}
```

- Allgemein

```
void name() /* bis hier: Kopf */ {
    // Innerhalb der geschweiften Klammern: Rumpf
    statement
    ...
}
```

- Klammern im Rumpf sind Pflicht, auch bei einer (oder überhaupt keiner) Anweisung

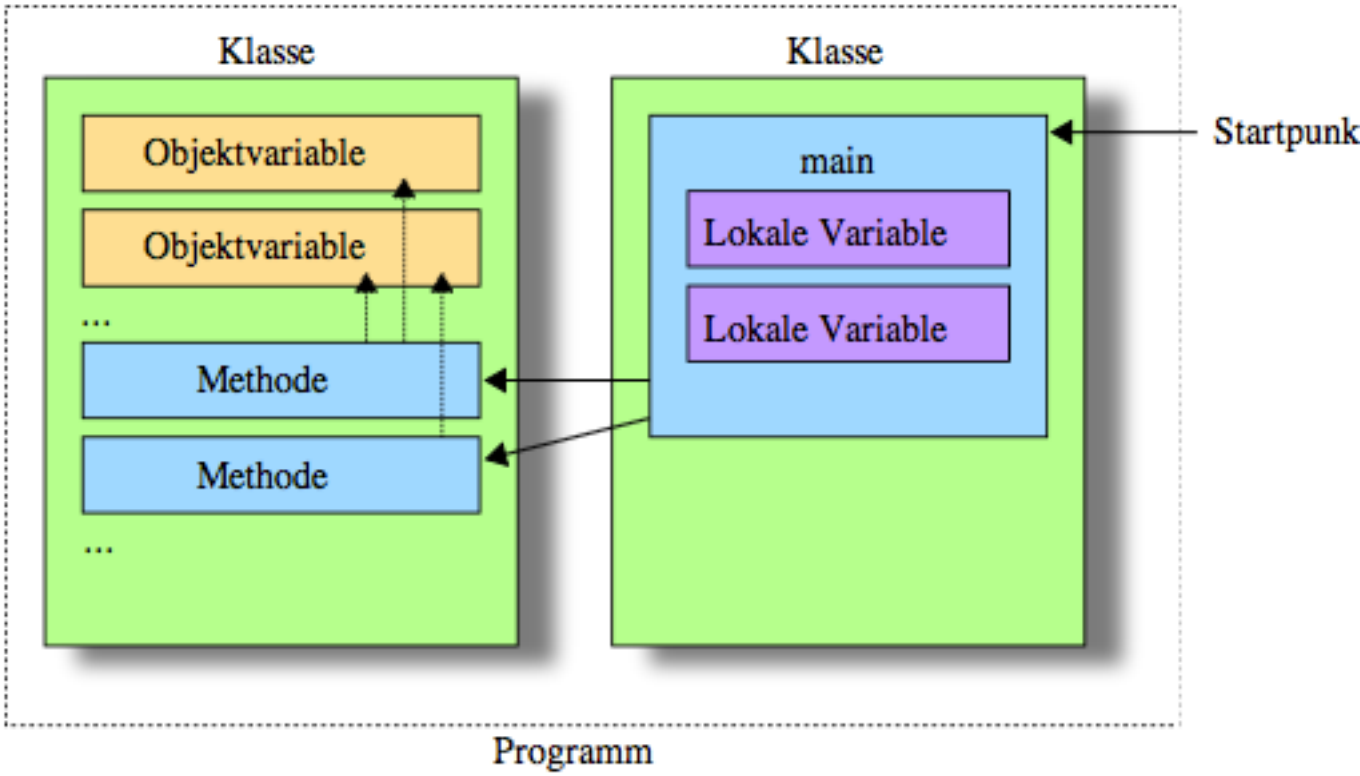
Methodenaufruf

- Methode wird mit Zielobjekt aufgerufen; Ohne Zielobjekt kein Aufruf
- Methodenaufruf syntaktisch ähnlich zu Elementzugriff: Zielobjekt`.`Methodenname`();`
- Runde Klammern markieren Methodenaufruf, fehlen bei Objektvariablenzugriff
- Beispiel: Bruch initialisieren, dann mit `print` ausgeben:

```
Rational r = new Rational();
r.numer = 1;
r.denom = 9;
r.print();      // Methodenaufruf, gibt 1/9 aus
```

Aufbau Klasse

- Methodendefinitionen nur in Klassen zulässig,
  - nicht außerhalb einer Klassendefinition,
  - nicht innerhalb einer anderen Methodendefinition
- Anzahl, Reihenfolge und Anordnung von Methodendefinitionen in einer Klasse beliebig
- Programm besteht aus mehreren Klassen, Klasse besteht aus Objektvariablen und Methoden:



Call-Sequence

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

```
Rational r = new Rational();
r.numer = 1;
r.denom = 9;
r.print();      // gibt 1/9 aus
r.numer = 5;
r.print();      // gibt 5/9 aus
```

Ablauf schematisch:

Application	Rational	*Ausgabe*
`main()`	`print()`	
`Rational r =`		
`new Rational();`		
`r.numer = 1;`		
`r.denom = 9;`		
`r.print();` →	`System.out.printf( ... );`	`1/9`
	←	
`r.numer = 5;`		
`r.print();` →	`System.out.printf( ... );`	`5/9`
	←	
…		

Methodenrumpf als Block

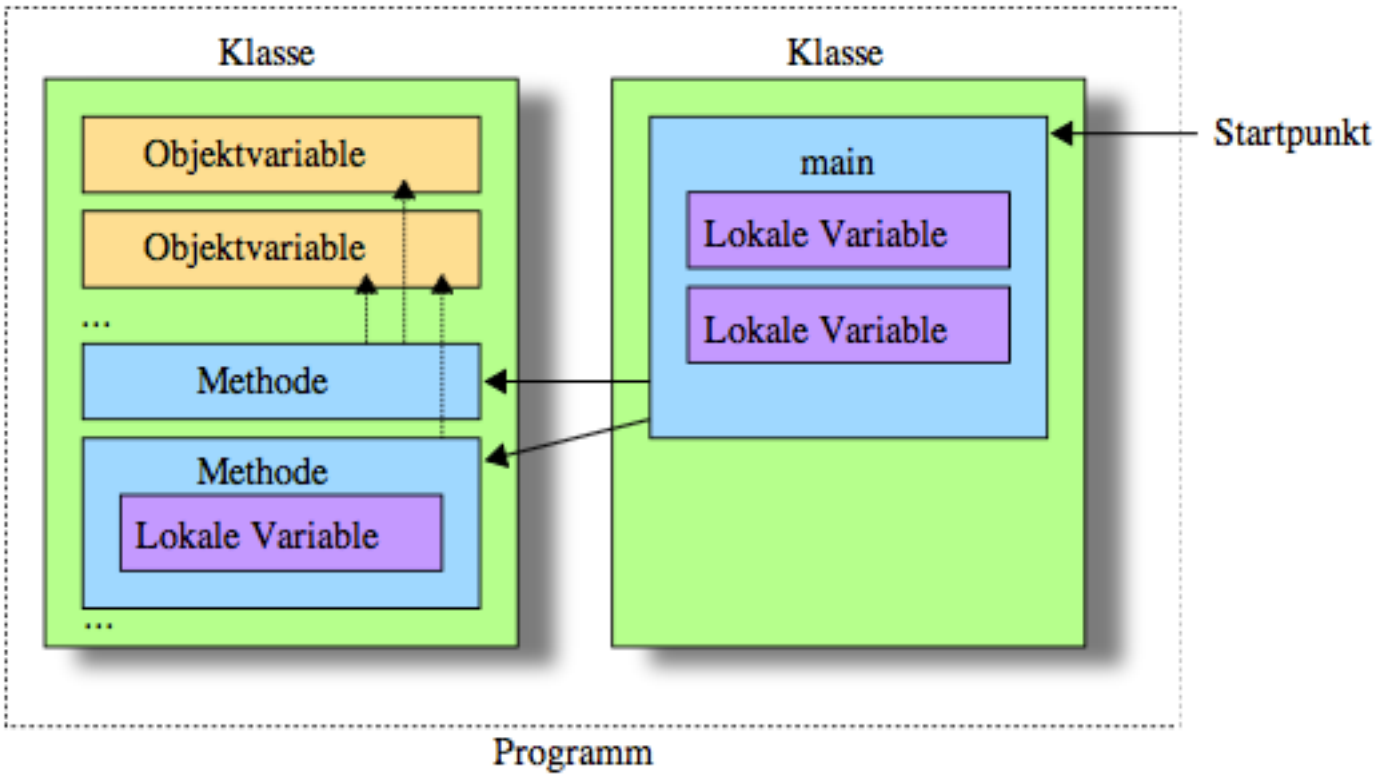
- Methodenrumpf = Block
- Gültigkeitsbereich lokaler Definitionen = Methodenrumpf
- Lebensdauer lokaler Variablen: jeweils ein Aufruf (Gegensatz Objektvariablen: Lebensdauer wie Objekt)
- Beispiel: Methode `reduce` zum Kürzen eines Bruchs:

```
class Rational {
  int numer;
  int denom;
  void reduce() {
    int gcd = ...;
    numer /= gcd;
    denom /= gcd;
  }
  ...
}
```

- Lokale Variable `gcd` gilt nur im Rumpf, existiert für jeweils einen Aufruf
- Aufruf von `reduce`:

```
Rational r = new Rational();
r.numer = 6;
r.denom = 9;
r.print();  // gibt 6/9 aus
r.reduce();
r.print();  // gibt 2/3 aus
```

- Programm besteht aus mehreren Klassen, Klasse besteht aus Objektvariablen und Methoden, Methoden enthalten lokale Variablen



Lokale Variable in Methode

Zugriff aus einem Methodenrumpf

- Zugriff auf Objektvariablen des eigenen Objektes ohne Angabe eines Zielobjekts
- Beispiel: `reduce`:
  Objektvariablen `numer`, `denom` 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

Zugriff aus einem Methodenrumpf – Beispiel

- Beispiel: Neue Methode `printReduced` gibt einen Bruch gekürzt aus:

```
class Rational {
  int numer;
  int denom;

  void print() { ... }

  void reduce() { ... }

  void printReduced() {
    reduce();       // Methode des eigenen Objektes
    print();    // Methode des eigenen Objektes
  }
  ...
}
```

Zugriff aus einem Methodenrumpf – Beispiel (2)

- Aufruf: Anwender sieht nur „äußeren“ Aufruf, nicht die beiden untergeordneten:

```
Rational r = new Rational();
r.numer = 6;
r.denom = 9;
r.printReduced();   // gibt 2/3 aus
```

Namenskollisionen

- Namen von lokalen Variablen und Objektvariablen kollidieren nicht
- Anlass: Benennung von lokalen Variablen ohne Rücksicht auf Objektvariablen
- Nachteil: Lokale Definition „verdeckt“ Objektvariable
- Beispiel:

```
class BlackMagic {
  int numer = 1;                // Objektvariable
  ...
  void voodoo() {
    int numer = 2;              // lokale Variable
    System.out.println(numer);  // gibt 2 aus
  }
}
```

- Technisch möglich, aber schlechter Stil
- In der Praxis selten ein Problem: Zugriffe auf Objektvariablen ohnedies besser auf einzelne Methoden beschränkt

Namensräume

- Namen müssen innerhalb eines **Namensraumes** eindeutig sein
- Getrennte Namensräume:
  - Objektvariablen einer Klasse
  - Methoden einer Klasse
  - Lokale Variablen und Parameter einer Methode
  - Klassennamen im Programm
- Beispiel: jedes „`foo`“ in in einem anderen Namensraum ⇒ keine Kollisionen

```
class foo {       // Klassenname
  int foo;        // Objektvariable

  void foo() {    // Methode
    int foo = 1; // Methodenrumpf, lokale Variable
  }
}
```

Selbstreferenz mit `this`

- Reserviertes Wort `this` = (Referenz auf) eigenes Objekt
- Automatisch definiert, immer verfügbar
- Öffnen Zugang zu verdeckten Objektvariablen. Beispiel:

```
class BlackMagic {
  int numer = 1;
  ...
  void voodoo() {
    int numer = 2;
    System.out.println(numer);      // gibt 2 aus
    System.out.println(this.numer); // gibt 1 aus
  }
}
```

- In der Regel überflüssig, abgesehen von Konstruktoren und Settern

Parameter

Argumente und Parameter

- **Parameter** zur Übergabe von Information vom Aufrufer an Methode

- Zwei Sprachelemente gekoppelt:
  1. Methode nennt Parameter
  2. Aufrufer liefert **Argumente** für die Parameter
- **Parameterliste** im Methodenkopf schematisch:

```
void name(type1 name1, type2 name2, ...) {
    ...
}
```

- Beliebig viele Parameter zulässig (bisher: keine Parameter, leere Liste)

Beispiel

- Methode `extend` zum Erweitern eines Bruchs mit Parameter `f` = Faktor, mit dem erweitert werden soll:

```
class Rational {
  void extend(int f) { // Kopf mit Parameter int f
    numer *= f;
    denom *= f;
  }
  ...
}
```

- Aufrufer muss bei jedem Aufruf ein kompatibles Argument angeben: `r.extend(2);`

Übergabe

- Argumente und Parameter vom Compiler bei jedem Aufruf paarweise abgeglichen
- Ein Argument pro Parameter erforderlich (zu viele oder zu wenige Argumente: wird nicht übersetzt)
- Typ jedes Arguments kompatibel zum entsprechenden Parameter
- Beliebig komplizierte Ausdrücke als Argumente zulässig, werden erst ausgerechnet, dann übergeben
- Verwendung der Parameter im Methodenrumpf: vergleichbar mit automatisch initialisierten lokalen Variablen
- Parameter = dritte Art von Variablen, neben lokalen Variablen und Objektvariablen

Call-Sequence mit Parametern

- Erweiterung der einfachen Call-Sequence parameterloser Methoden

- Einzelschritte:
  - (A) **Aufrufende Methode**
  - (B) **Aufgerufene Methode**

  1. (A) Werte aller Argumente von links nach rechts berechnen
  2. (B) Parameter erzeugen
  3. (B) Argumentwerte aus Schritt 1 an die Parameter zuweisen
  4. (A) Ablauf unterbrechen
  5. (B) Rumpf durchlaufen
  6. (B) Parameter von Schritt 2 freigeben
  7. (A) Ablauf fortsetzen

Beispiel

- Ablauf der Call-Sequence bei einem Aufruf der Methode `extend`

  ```
  r.extend(3 * 7);
  ```

  in Zeitlupe:

  1. Wert des Argumentes berechnen: `3 * 7 → 21`
  2. Parameter `int f` gemäß Parameterliste erzeugen
  3. Parameter initialisieren `f = 21`
  4. Aufrufer unterbrechen
  5. Methodenrumpf durchlaufen: `numer *= 21;`
     `denom *= 21;`
  6. Parameter `f` freigeben
  7. Aufrufer fortsetzen

Mehrere Parameter

- Liste von Parametern im Methodenkopf (Komma zwischen je zwei Parametern)
- Beispiele:
  - Methode `set` zum Setzen von Zähler und Nenner eines Bruches
  - Methode `setZero` zum Rücksetzen eines Bruches auf null

```
class Rational {
  void set(int n, int d) { // 2 Parameter
    numer = n;
    denom = d;
  }
  void setZero() {         // 0 Parameter
    numer = 0;
    denom = 1;
  }
}
```

Aufruf

- Aufruf mit jeweils passender Anzahl Argumente:

```
Rational r = new Rational();
r.set(2, 3);                // r = 2/3

Rational s = new Rational();
s.setZero();                // s = 0/1
```

- Unzulässige Aufrufe mit falscher Anzahl Parameter:

```
r.set(2);       // Fehler: zu wenig Argumente
r.setZero(0);   // Fehler: zu viele Argumente
```

Primitive Typen als Parameter

- Versteckte Wertzuweisung bei Parameterübergabe
- Werte primitiver Typen werden kopiert
- Implizite und explizite Typumwandlungen wie bei Wertzuweisungen

```
r.extend(3.14);          // Fehler: falscher Typ
r.extend((int)3.14);     // ok
```

Referenztypen als Parameter

- Referenztypen für Parameter zulässig
- Beispiel: Methode `mult` erwartet anderes `Rational`-Objekt als Parameter, vervielfacht `this` mit dem Parameterobjekt

```
class Rational {
  ...
  void mult(Rational that) {
    numer *= that.numer;
    denom *= that.denom;
  }
}
```

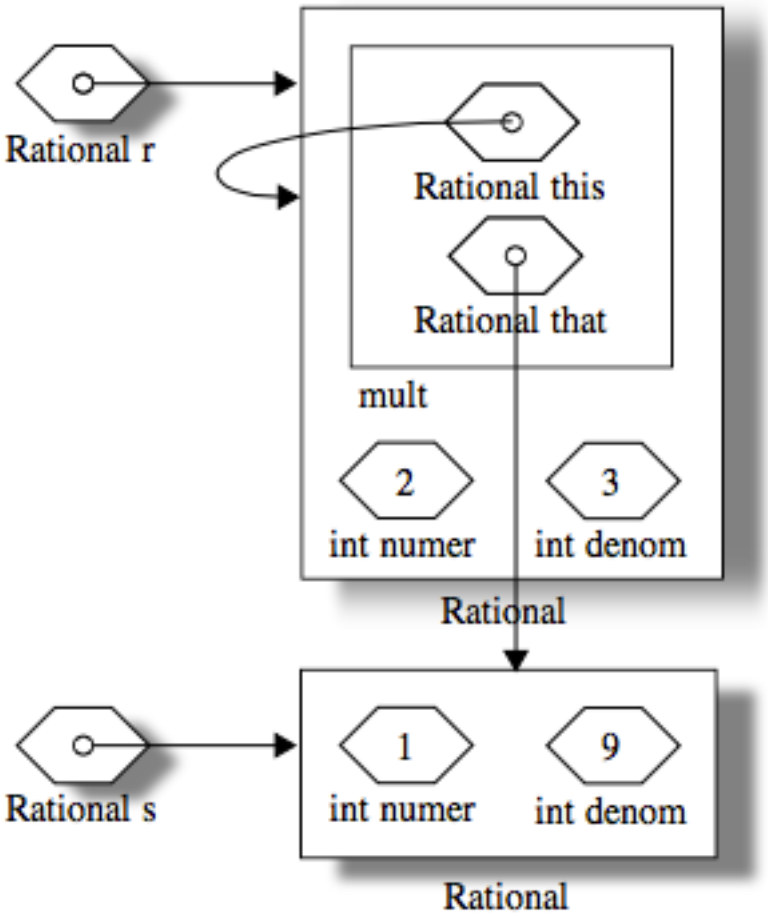
- Aus der Sicht von `mult`: `that` ist ein anderes Objekt
  - Ansprechen der eigenen Objektvariablen ohne Zielobjekt
  - Ansprechen der fremden Objektvariablen mit Zielobjekt `that`

Aliasing bei Referenzparametern

- Aliasing bei der Übergabe von Objekten wie bei Wertzuweisungen
- Objekt des Aufrufers wird nicht kopiert, sondern Referenz übergeben
- Im Rumpf von `mult`: Argument des Aufrufers und der Parameter der Methode referenzieren dasselbe Objekt.

```
Rational r = new Rational();
r.set(2, 3);
Rational s = new Rational();
s.set(1, 9);
r.mult(s);
```

- Beim Eintritt in `mult`:



this and that

- Bei `r.mult(s)` zwei verschiedene `Rational`-Objekte verwickelt:
  1. Zielobjekt des Methodenaufrufs `r` (`this`)
  2. Argument-Objekt `s` des Aufrufers, im Methodenrumpf als Parameter `p` erreichbar (Aliasing)

Fragwürdige Schreibzugriffe

- `mult` liest Objektvariablen des Parameterobjektes, verändert eigene Objektvariablen
- Nach `mult`: Parameterobjekt unverändert:

```
s.print();
r.mult(s);
s.print();      // gleiche Ausgabe wie vorher
```

- Böswillige Version von `mult`: Schreibt in das Parameterobjekt:

```
void mult(Rational that) {
  ...
  that.denom = 0; // böse!
}
```

- Für den Aufrufer nicht erkennbar: Methodenaufruf ruiniert das Argument!

```
s.print();
r.mult(s);
s.print();      // Nenner von s = 0!
```

- Keine schreibenden Zugriffe auf fremde Objektvariablen
- Besser: unveränderliche Klassen

`final`-Parameter

- in der Regel ist es eine schlechte Idee Parameter ändern zu wollen, aber es ist möglich
- Beispiel:

```
void silly(int a) {
  a++;
  this.a += a;
}
```

- das Gleiche kann etwas schöner auch so erreicht werden:

```
void notSoSilly(int a) {
  this.a += a+1;
}
```

- noch besser: Parameter als **unveränderlich** markieren:

```
void silly(final int a) {
  a++; // Fehler. Wird nicht übersetzt!
  this.a += a;
}
```

- Regel:

  "Machen Sie Parameter **immer** `final`, außer Sie haben einen guten Grund!"

Hier geht es weiter in Softwareentwicklung II (IB)