

# Datenkapselung

Softwareentwicklung II (IB)

---

Prof. Dr. Oliver Braun

Letzte Änderung: 01.05.2018 18:33

- **Modul** (engl. module) = Programmteil, in Java i.d.R. eine Klasse
- **Modularisierung** = Aufteilung in Module: Ziel bei der Konstruktion von (größeren) Programmen
- Reduktion der Abhängigkeiten zwischen Modulen: Module leichter einzeln entwerfen, implementieren, austauschen, erweitern, testen, korrigieren, ...
- **Datenkapselung** (engl. data hiding, data encapsulation): Maßnahme zur Reduktion der Abhängigkeiten

- Datenkapselung = Verbergen von Daten hinter Operationen
- Zugriff auf Daten nur über Operationen
- Java (und andere objektorientierte Sprachen):

---

Daten	↔	Objektvariablen
Operationen	↔	Methoden

---

- Anwender einer Klasse: Darf Methoden aufrufen, aber keine Objektvariablen (direkt) ansprechen

- Modifier **private** begrenzt Sichtbarkeit auf die eigene Klasse
- Zugriff von „außen“ wird nicht übersetzt

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

- Objektvariablen „existieren“ für Anwender nicht:

```
class Application {  
    public static void main(String[] args) {  
        Rational r = new Rational();  
        r.numer = 1; // Fehler - unbekannte Variable  
    }  
}
```

## Kein Zugriffsschutz innerhalb der eigenen Klasse

- Innerhalb der eigenen Klasse keine Einschränkung
- Beispiel: `print` funktioniert unverändert

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

- `private` wird vom Compiler beim Übersetzen berücksichtigt, nicht von der JVM zur Laufzeit
- `private` schottet Klassen gegeneinander ab, nicht Objekte!
- Konsequenz: Destruktives `mult` weiterhin zulässig
- Deshalb: Unveränderliche Klassen zusammen mit Datenkapselung einsetzen

- `private` für alle Definitionen, auch Methoden
- `private`-Methoden von außen nicht aufrufbar, nur von Methoden der eigenen Klasse
- Sinnvoll für Hilfsmethoden, die der Anwender nicht benutzen soll
- Beispiel: Berechnung des größten gemeinsamen Teilers in `Rational`

```
class Rational {  
    ...  
    void reduce() {  
        final int gcd = gcd(numer, denom);  
        numer /= gcd;  
        denom /= gcd;  
    }  
    private int gcd(int a, int b) {
```

- **private**-Objektvariable von außen nicht erreichbar
- Wert dennoch zugänglich machen: **Auskunftsmethode** (auch „Inspector-Methode“, „Accessor-Methode“, engl. `getter`) anbieten
- Definition eines Getter zu einer Objektvariablen

```
private type name;
```

nach dem Muster

```
type getName() {  
    return name;  
}
```

- Beispiele `getNumer` und `getDenom` für Objektvariablen `numer` und `denom`



- **private**-Objektvariablen in veränderlichen Klassen: von außen kein Zugriff, keine Veränderung möglich
- Um dennoch Modifikation zulassen: **Änderungsmethoden** (auch „Modifier-Methoden“, engl. setter) anbieten
- Definition eines Setter zu einer Objektvariablen

```
private type name;
```

nach dem Muster (hier **this** sinnvoll anwendbar):

```
void setName(type name) {  
    this.name = name;  
}
```

- Getter und Setter lassen sich mechanisch generieren

- `setNumer` und `setDenom` einer veränderlichen Fassung der Klasse `Rational`:

```
class Rational {  
    private int numer;  
    private int denom;  
    void setNumer(int numer) { // Setter fuer numer  
        this.numer = numer;  
    }  
    void setDenom(int denom) { // Setter fuer denom  
        this.denom = denom;  
    }  
    ...  
}
```

- Private Objektvariablen, Setter und Getter auf den ersten Blick unnötig kompliziert gegenüber ungeschützten, öffentlichen Objektvariablen
- Getter und Setter eröffnen neue Möglichkeiten:
  - Kontrolle der Werte
  - Fehlersuche
  - Abweichende Implementierung
  - Optimierungen
- **noch besser:** Setter und Getter nur dann definieren, wenn sie wirklich benötigt werden

- Objektvariable darf in der Regel nicht alle Werte des Typs annehmen
- Beispiel `Rational`: `denom` darf nicht 0 sein
- Setter können unerwünschte Werte abweisen (zum Beispiel durch Einsatz einer Exception):

```
void setDenom(int denom) {  
    if (denom == 0)  
        throw new ArithmeticException("zero denominator");  
    this.denom = denom;  
}
```

- Keine Möglichkeit zur Kontrolle bei freiem Zugriff

- Protokoll aller Änderungen einer Objektvariablen:

```
void setNumer(int numer) {  
    System.out.printf("calling setNumer(%d)%n", numer);  
    this.numer = numer;  
}
```

- Grundlage zur Fehlersuche mit Traces
- Freier Zugriff: keine Überwachung möglich

## Abweichende Implementierung

- Setter und Getter können „Objektvariable vorgaukeln“, die in Wahrheit nicht existiert:

```
double getReal() {  
    return (double)numer/denom;  
}  
void setReal(double real) {  
    // Version 0.1-alpha-prerelease :-)  
    numer = (int)(real*1000);  
    denom = 1000;  
    reduce();  
}
```

- Aufrufe greifen nicht auf Objektvariable zu, sondern berechnen Werte
- Für den Anwender nicht erkennbar

- Mit Gettern können Maßnahmen bis zum tatsächlichen Zugriff verzögert werden
- Beispiel **Rational**: Mit ungekürztem Bruch rechnen, erst beim Aufruf eines Getters wirklich kürzen

## Nicht privat = öffentlich?

- Java hat noch eine Zwischenstufe
- bisher haben wir vor den Member (Objektvariablen und Methoden) nichts angegeben
  - damit sind die Member **package-private**
  - da wir im Moment alles in das gleiche (namenlose) Package schreiben, können wir aus unseren anderen Klassen auf alle Member die package-private sind, zugreifen
- um die Member aber auch außerhalb des Packages sichtbar zu machen, gibt es das Schlüsselwort **public**, z.B.

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

- ab dem Blatt 3 verwenden wir **public** und brauchen dann (für Checkstyle) Javadoc



- **Schnittstelle** (engl. interface) einer Klasse:  
  
    **Signaturen** der öffentlichen (= nicht-privaten) Methoden
  - öffentliche (= nicht-private) Objektvariablen
- **Implementierung**: alles andere
- Plakativ:
  - Das Interface beschreibt, was eine Klasse bietet
  - Die Implementierung legt fest, wie sie das bewerkstelligt
- Anwender muss nur das Interface einer Klasse kennen, um sie zu verwenden

- Beziehung zwischen Klassendefinition und Anwendung ähnelt Geschäftsbeziehung

---

Java	Geschäftsleben
Klassendefinition	Anbieter, Lieferant Kunde Vertrag,
Anwendung einer Klasse	vereinbarte Leistungen
Interface	Betriebsmittel, interne Maßnahmen,
Implementierung	Geschäftsgeheimnisse

---

**Anbieter und Kunde vereinbaren Vertrag** Java: Interface festlegen, noch vor der Implementierung

**Anbieter ändert oder streicht zugesicherte Leistungen** Alle Kunden sind betroffen (voraussichtlich wenig Begeisterung)

Java: Modifikationen an der Schnittstelle einer Klasse problematisch

**Anbieter ändert interne Maßnahmen** Den Kunden gleichgültig, solange Verträge eingehalten werden

Java: Implementierung der Klasse kann ohne Rücksprache und Auswirkungen manipuliert werden

**Über den Vertrag hinausgehende Leistungen** Werden aus wirtschaftlichen Gründen kaum freiwillig erbracht

Java: Alle Objektvariablen und Methoden **private** definieren, die nicht gemäß Interface verlangt sind

### Fazit

- Schnittstelle einer Klasse frühzeitig festlegen und minimal halten = maximaler Freiraum für die Implementierung

# Organisation von Klassendefinitionen

- Anordnung der Bestandteile innerhalb einer Klassendefinition formal unerheblich (abgesehen von statischen Elementen)
- Dennoch: Einheitliche Organisation sinnvoll  $\Rightarrow$  Lesbarkeit, Dokumentationswert, Codequalität
- Vorschlag für konkrete Reihenfolge von Abschnitten:
  1. Konstanten
  2. Objektvariablen
  3. Default-Konstruktor, Copy-Konstruktor, Custom-Konstruktoren
  4. Getter und weitere Auskunftsmethoden
  5. Setter und weitere ändernde Methoden
  6. Standard-Methoden (`equals`, ...)
  7. Hilfsmethoden
- Optische Trennung (Zeilenkommentare) der Abschnitte erleichtert

# UML

- **Unified Modeling Language (UML)** = neutrale Darstellung von Programmstrukturen
- Halbformale Diagramme mit Beschriftungen
- Mehrere Versionen, siehe [OMG](#)
- Zahlreiche Diagrammformen für unterschiedliche Aspekte
- Für diese Zwecke nützlich: statische Klassendiagramme
- Aktivitätsdiagramme (1. Semester) sind auch UML
- mehr UML in Software Engineering

- Diagrammform der UML unter vielen anderen
- Ziele: Darstellung der ...
  - ... Beziehungen von Klassen untereinander
  - ... wichtigsten Bestandteile von Klassen
- „Landkarte“ eines komplexen Programms, schneller Überblick, leichtere Orientierung



- Klasse = Kasten mit drei „Fächern“: Name der Klasse, Objektvariablen, Methoden

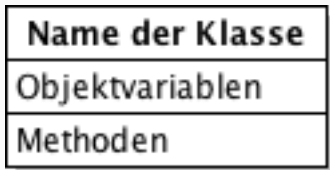


Abbildung: Klassendiagramm

- Objektvariablen mit Typ und Name
- Methoden mit Signatur, keine Rümpfe
- Modifier „-“ für `private`-Elemente

- Statisches UML-Klassendiagramm

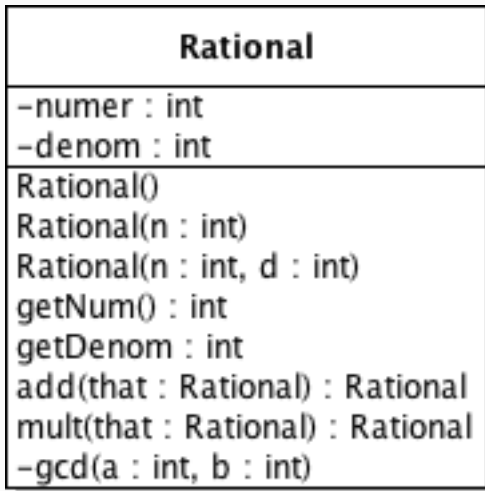


Abbildung: Rational UML