Konstanten, Javadoc, Patterns, Anti-Patterns

Konstanten

In Java werden Konstanten definiert als

```
final <typ> <NAME> = <wert>;
```

Beispielsweise

```
final int ANTWORT = 42;
```

`final` sagt aus, dass der Wert nicht geändert werden kann; das unterscheidet eine Konstante von einer Variablen

Dabei muss man beachten:

- `final` macht nur aus primitiven/skalaren Typen korrekt eine Konstante (`int`, `double`, `...`)
- Bei Objekten hingegen kann nur das Objekt nicht mehr durch ein anderes Objekt ersetzen, der Inhalt des Objekts kann immernoch geändert werden

Konstanten: Verwendung und Vorteile

Bis auf wenige Ausnahmen sollte man keine Zahlen im Programmlauf verwenden, besser: An zentraler Stelle Konstanten definieren
Zahlen im Programmlauf sind in Ordnung, wenn die Zahlen „klein“ sind und es keine große Wahlfreiheit gibt:

- 0 für kleinsten möglichen Arrayindex, z.B. `for(i=0; i<foo; i++)`
- 2 als Teil der Berechnung des Kreisumfangs `2*pi*radius`

Ansonsten sollte man nur Konstanten verwenden

Vorteile von Konstanten:

- Man kann Werte besser einheitlich ändern
- Für den Entwickler: Gleichbleibender Wert garantiert, muss Wertänderung während des Programmlaufs nicht berücksichtigen
- Für den Compiler: Kann besser optimieren

Konstanten und Enums

Per `final` deklarierte Konstanten sind für einzelne Werte gedacht
Hat man mehrere angeordnete Konstanten eignen sich Enums besser; Vorteile von Enums

- Enums erlauben Durchlaufen von Konstanten:
  `for (Kartenfarbe f : Kartenfarbe.values()) {`
- Enums erlauben switch-case-Konstrukte

```
switch(karte) {
  case(KREUZ): ...
    break;
  case(PIK): ...
    break;
  default:
    ...
}
```

Aber: Braucht man nur einzelne Konstanten oder ergeben Aufzählungen keinen Sinn, sind `final`-Konstanten oft einfacher

Dokumentation

Wenn man nicht sagt, was ein Programm machen soll, ist die Wahrscheinlichkeit gering, dass es das macht.

Dokumentation soll helfen,

- Schnittstellen zu verstehen
- Entwurfsideen zu erklären
- implizite Annahmen (z.B. Klasseninvarianten) auszudrücken

Nicht sinnvoll:

```
x++; // erhöhe x um eins
```

Was soll man dokumentieren?

- für jede Methode eine Zusammenfassung, was sie macht
- welche Werte zulässige Eingaben für eine Methode sind
- wie Methoden mit unzulässigen Eingaben umgehen
- wie Fehler behandelt und an den Aufrufer der Methode zurückgegeben werden
- Verträge (Vorbedingungen, Nachbedingungen, Klasseninvarianten)
- Seiteneffekte von Methoden (Zustandsänderungen, Ein- und Ausgabe, usw.)
- wie die Klasse mit Nebenläufigkeit umgeht
- (wie Algorithmen funktionieren, wenn das nicht leicht vom Programmtext ablesbar ist)

Javadoc-Kommentare

Javadoc erlaubt speziell ausgezeichnete Kommentare automatisch aus dem Code herauszuziehen und übersichtlich darzustellen

- Javadoc-Kommentare
  `/** HTML-formatierter Kommentar */`
- stehen vor Packages, Klassen, Methoden, Variablen
- spezielle Tags wie `@see`, `@link`
- HTML-Dokumentation wird erzeugt mit

```
javadoc Package
javadoc Klasse1.java Klasse2.java ...
```

Javadoc-Tags

Allgemein

- `@author Name`
- `@version text`

Vor Methoden

- `@param Name Beschreibung` – beschreibe einen Parameter einer Methode
- `@return Beschreibung` – beschreibe den Rückgabewert einer Methode
- `@throws Exception Beschreibung` – beschreibe eine Exception, die von einer Methode ausgelöst werden kann

Javadoc-Tags

Querverweise

- `{@link Klasse}`
  `{@link Klasse#Methode}`
  ...
  Verweis auf andere Klassen und Methoden im laufenden Text
  (in HTML-Version werden daraus Links, Warnung wenn nicht existent)
- `@see Klasse`
  `@see Klasse#Methode`
  ...
  Fügt der Dokumentation einen „See Also“-Abschnitt mit Querverweisen hinzu.

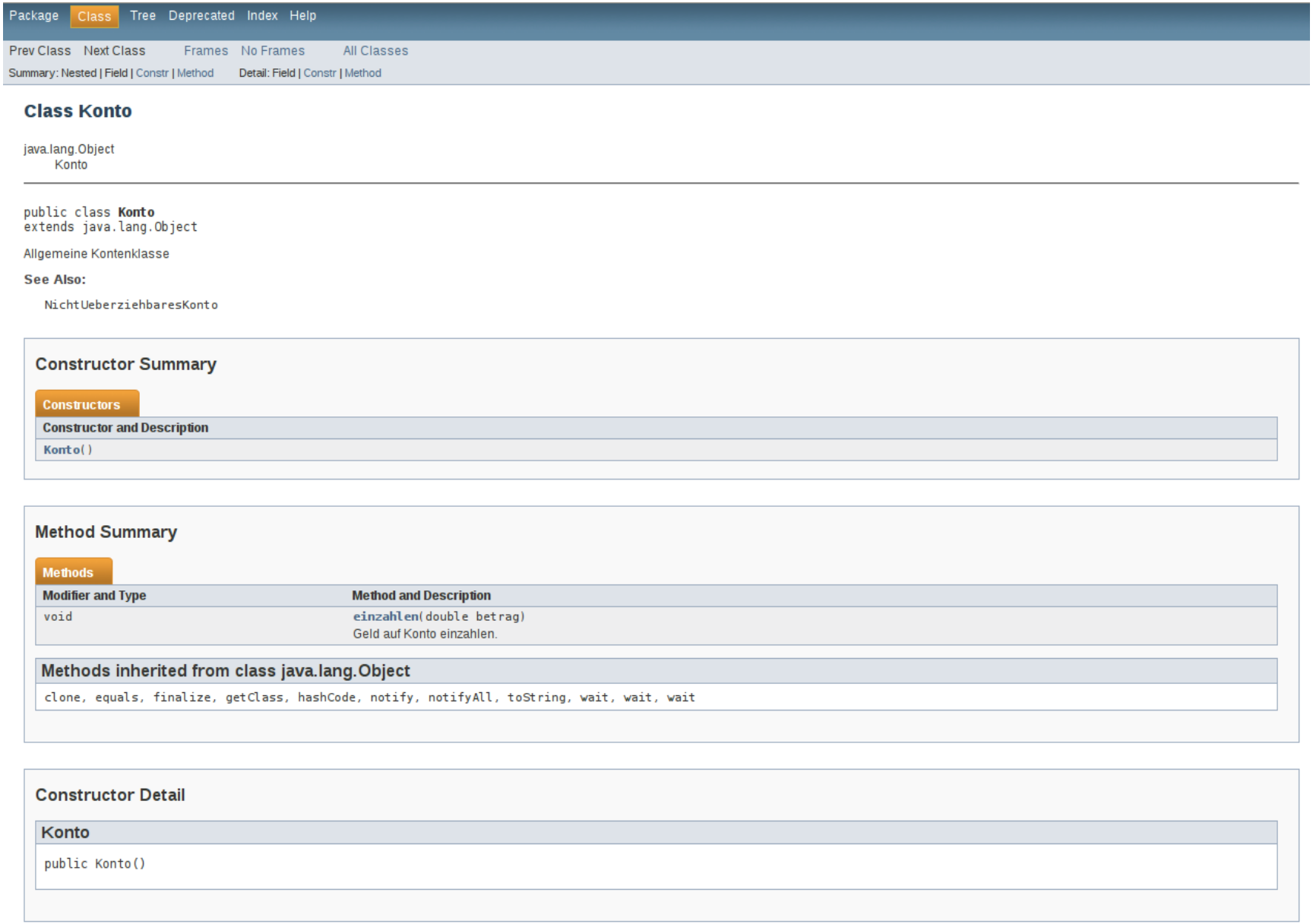
  `@see`-Tags können nicht im laufenden Text stehen

Javadoc-Beispiel

```
/**
 * Allgemeine Kontenklasse
 * @author Marcus Licinius Crassus
 * @see NichtUeberziehbaresKonto
 */
public class Konto {

   /**
    *  Geld auf Konto einzahlen.
    *  <p>
    *  Wenn vorher {@code getKontoStand() = x}
    *  und {@code betrag >=0},
    *  dann danach {@code getKontoStand() = x + betrag}
    *  @param betrag positive Zahl, der einzuzahlende Betrag
    *  @throws ArgumentNegativ wenn betrag negativ
    */
   public void einzahlen(double betrag);
}
```

Javadoc-Beispiel: erzeugte html-Dokumentation



Package Class Tree Deprecated Index Help

Prev Class Next Class Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

Class Konto

java.lang.Object
    Konto

---

public class **Konto**
extends java.lang.Object

Allgemeine Kontenklasse

See Also:

NichtUeberziehbaresKonto

Constructor Summary

Constructors

Constructor and Description
Konto()

Method Summary

Methods

Modifier and Type	Method and Description
void	einzahlen(double betrag) Geld auf Konto einzahlen.

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Constructor Detail

Konto

public Konto()

Javadoc-Beispiel: erzeugte html-Dokumentation, Fortsetzung

Method Summary

Methods

Modifier and Type	Method and Description
`void`	**`einzahlen`**`(double betrag)` Geld auf Konto einzahlen.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Konto

```
public Konto()
```

Method Detail

einzahlen

```
public void einzahlen(double betrag)
```

Geld auf Konto einzahlen.

Wenn vorher `getKontoStand() = x` und `betrag >=0` , dann danach `getKontoStand() = x + betrag`

Parameters:

`betrag` - positive Zahl, der einzuzahlende Betrag

Throws:

`ArgumentNegativ` - wenn betrag negativ

Lesbarkeit, Testbarkeit, Änderbarkeit

Ob ein Programm korrekt funktioniert, ist nicht das einzige wichtige Kriterium. Ferner zu beachten sind auch

- **Lesbarkeit**: Wie gut kann man nachvollziehen, was der Code macht?
- **Testbarkeit**: Wie leicht kann man testen, ob der Code das gewünscht macht?
- **Änderbarkeit**: Wie gut kann man den Code an geänderte Anforderungen anpassen?
- **Wiederverwertbartkeit**: Wie gut kann man den geschriebenen Code an verschiedenen Stellen verwenden?

Denn ohne diese Eigenschaften

- können sehr leicht Fehler versteckt sein
- macht man leichter Fehler und sucht oft länger pro Fehler
- schreibt man oft sehr ähnlichen Code immer wieder

Patterns, Anti-Patterns

Patterns (Design Patterns, Entwurfsmuster, . . .) und Anti-Patterns beschreiben Erfahrungswerte

- Anti-Patterns (auch genannt: Code smells, übelriechender Code, . . .): sind keine Fehler im eigentlichen Sinn, aber sollte man genauso vermeiden, da sie oft Fehler und andere Probleme verursachen
- Pattern: Codemuster, die dabei unterstützen korrekte Programme zu schreiben

Patterns betreffen oft die Lesbarkeit, Testbarkeit, Änderbarkeit und Wiederverwertbartkeit.

Bereits vorgestelltes Pattern: Model-View-Controller

$\Rightarrow$ Patterns merken und verwenden, Anti-Patterns merken und vermeiden

Anti-Pattern: Code-Duplikation

Code-Duplizierung (Klone, Kopien, . . .): Code wird an verschiedene Stellen kopiert und stellenweise für eine andere Aufgabe angepasst.
Probleme:

- Code wird sehr schnell unübersichtlich
- Erhöhter Wartungsaufwand: Der gleiche Code muss oft gelesen werden
- Inkonstistente Änderungen: Wenn der Code an einer Stelle geändert wird, müsste er an den kopierten Stellen auch geändert werden, wird oft aber übersehen. Darum muss oft der gleiche Fehler mehrfach gesucht werden
- Unterschiede zwischen ähnlichen Stellen sehr schwer erkennbar; Funktionalität wird also verschleiert, Lesbarkeit leidet stark

Stattdessen: Code anders strukturieren; Funktionen schreiben und diese aufrufen

Anti-Pattern: Lange Methoden

Zu lange Methoden sind oft Symptome für andere Probleme

- Zu viel Funktionalität in einer Methode implementiert
  $\Rightarrow$ schlechtere Wiederverwertbarkeit von Codeteilen
- Methode leidet unter Code-Duplikation
  Code-Duplikation beheben!
- Problem ist umständlich gelöst: Oft ist der Code dadurch schwerer zu lesen und hat mehr Fehler, einfachere Lösungen sind meistens besser

Abgesehen von zugrundeliegenden Problemen ist eine zu lange Methode auch ein Problem an sich

- Methode ist schwerer zu verstehen
- Eine Methode für sich ist leichter zu testen als ein Teil. Fehler kann man darum oft auf eine Methode eingrenzen. Ist diese länger, ist der genaue Fehler schwerer zu finden

Anti-Pattern: Zu große Klasse/Gott-Objekt

Eine zu große Klasse hat ähnliche Probleme wie eine zu große Methode

Zudem: Eine zu große Klasse kann den Programmfluss des ganzen Programms verschleiern und nicht nur einen Teil des Programms unleserlich machen

Eine zu große Klasse ist auch oft ein Symptom von anderen Problemen

- Schlecht definierte Rollenverteilung von verschiedenen Klassen
- Eine Klasse kümmert sich zu viel um Probleme, die in anderen Klassen gelöst werden sollten

Pattern: Wenige Variablen, kleiner Gültigkeitsbereich

Hat man nur wenige Variablen, kann man den Code deutlich besser nachvollziehen, aber auch andere Probleme werden vermieden

- Man kann alle gültigen Variablen gleichzeitig im Kopf haben
- Kurze Einlesezeit
- Reduziert versehentliche Variablenverwechslungen

Dazu gibt es eine Reihe von Möglichkeiten

- Immer minimal möglichen Gültigkeitsbereich wählen
  - Schleifenvariablen in der Schleife definieren
  - Variablen für Zwischenergebnisse einer Methode in der Methode deklarieren
  - Variablen für Zwischenergebnisse erst bei erster Verwendung deklarieren

Anti-Pattern: Viele Variablen

Sind viele Variablen aktiv, ist das oft auch ein Symptom für andere Probleme

- Code-Duplikation, vielleicht sogar Mehrfachberechnung mit unterschiedlichen Variablennamen
- Unsauber definierter Berechnungsablauf
- Zu große Methoden
- Zu große Klassen
- Es werden Berechnungen durchgeführt, deren Ergebnisse nicht (mehr) verwendet werden

Pattern: Nameskonvention, Groß-/Kleinschreibung

Die in Java gebräuchliche Namenskonvention gibt an

- Variablen und Methoden fangen mit einem kleinen Buchstaben an, Wortteile werden durch Großbuchstaben getrennent `beispielName`; nennt man dromedaryCase
- Konstanten werden in Großbuchstaben geschrieben, Wortteile werden durch Unterstrich abgetrennt `BEISPIEL_NAME`; nennt man SNAKE_CASE
- Klassen und Interfaces fangen mit einem Großbuchstaben an, Wortteile werden durch Großbuchstaben getrennent `BeispielName`; nennt man CamelCase

So kann man leichter erkennen, um was es sich handelt, wenn man etwas hinschreibt

Hinweis: Nicht in allen Programmiersprachen gibt es so einheitliche Namenskonventionen, in dem Fall sollte man sich zumindestens innerhalb einer Projektgruppe auf eine Konvention für dieses Projekt einigen

Pattern: Gute Namen

Ein Name sollte dem Zweck angemessen und übersichtlich sein

- Der Name sollte die Funktion beschreiben und nicht den Inhalt (`final BITS = 8;` statt `final EIGHT = 8;`)
  Damit wird die Funktion klarer; vermeidet zudem verwirrende Namen nach Änderungen `final EIGHT = 16;`
- Die meisten Namen sollten von mittlerer Länge sein:
  - zu kurze Namen geben nicht genug Auskunft über ihren Zweck
  - zu lange Namen behindern den Lesefluß
- Lediglich Schleifenvariablen in einer Schleife mit kurzem Inhalt können kurze Namen haben:

```
for(int i=0; i<maxVal; i++) {
  squares.add(i*i);
  }
```

Pattern: Design by Contract

Teile das zu entwickelnde Programm in kleine Einheiten (Klassen, Methoden), die unabhängig voneinander entwickelt und überprüft werden können.

Einheiten mit klar definierten Aufgaben und wohldefiniertem Verhalten, z.B.

- Klasse `List`
- Methode `List.add`

Setze größere Komponenten (letztendlich das Gesamtprogramm) aus kleineren zusammen.

Wenn sich alle Teilkomponenten korrekt verhalten, dann auch die zusammengesetzte Komponente.

Entwicklung von Komponenten

Jede Komponente

- hat eine klar definierte Aufgabe.
- stellt ihre Dienste durch eine öffentliche Schnittstelle bereit.
- kann selbst andere Komponenten benutzen.

Komponenten sollen voneinander *unabhängig* sein:

- unabhängige Entwicklung und Überprüfung (z.B. von verschiedenen Teammitgliedern)
- Austauschbarkeit
- interne Änderungen in einer Komponente beeinflussen andere Komponenten nicht negativ

Wie kann man solche Komponenten entwickeln und sicherstellen, dass sich zusammengesetzte Komponenten wunschgemäß verhalten?

Entwicklung von Komponenten

In einer objektorientierten Sprache spielen Klassen die Rolle von Komponenten.

Beim Implementieren einer Klasse sollte man sich genau überlegen

- welche *Leistungen* die einzelnen Funktionen der Klasse erbringen sollen.
- welche *Annahmen* die Funktionen über ihre Argumente und den Zustand der Klasse machen.

Dokumentiere Leistungen und Annahmen der Klasse.

Benutze nur die dokumentierten Leistungen einer Klasse und stelle immer sicher, dass die dokumentierten Annahmen erfüllt sind.

Java-Typsystem oft zu schwach, um Annahmen zu garantieren (z.B. Wert von x immer positiv).

Beispiel

```
public class Konto {

   public double getKontostand()

   /**
    *  Wenn vorher getKontoStand() = x
    *  und betrag >= 0,
    *  dann danach getKontoStand() = x + betrag
    */
   public void einzahlen(double betrag)

   /**
    *  Wenn vorher getKontoStand() = x
    *  und x > betrag >= 0,
    *  dann danach getKontoStand() = x - betrag */
   public void abheben(double betrag)
}
```

Entwicklungsprinzip

Benutze immer nur die dokumentierten Leistungen einer Klasse und stelle immer sicher, dass die dokumentierten Annahmen erfüllt sind.

Austauschbarkeit Jede Klasse kann durch eine andere mit der gleichen öffentlichen Schnittstelle ersetzt werden, solange diese mindestens die gleichen Leistungen erbringt und nicht mehr Annahmen macht und

unabhängige Entwicklung Teammitglieder einigen sich über Leistungen und Annahmen und können diese dann unabhängig voneinander implementieren.

unabhängige Überprüfung systematisches Testen der dokumentierten Leistungen

Design by Contract

Dieses Entwicklungsprinzip ist unter dem Namen Design by Contract (etwa: Entwurf gemäß Vertrag) bekannt.

Design by Contract enthält eine Reihe von methodologischen Prinzipien zur komponentenbasierten Softwareentwicklung.

Analogie mit Verträgen im Geschäftsleben:

- Die Komponenten schließen untereinander Verträge ab.
- Ein Vertrag beinhaltet das Versprechen einer Programmkomponente, eine bestimmte Leistung zu erbringen, wenn bestimmte Voraussetzungen erfüllt sind.
- Setze Programm so aus Komponenten zusammen, dass es richtig funktioniert, wenn nur alle ihre Verträge einhalten.

Verträge

Verträge werden zwischen Klassen abgeschlossen und haben folgende Form:

Wenn vor dem Aufruf einer Methode in einer Klasse eine bestimmte Voraussetzung erfüllt ist, dann wird die Klasse durch die Ausführung der Methode in einen bestimmten Zustand versetzt.

Beispiele

Für jedes Objekt `List l` gilt:

- Nach der Ausführung von `l.add(x)` gilt die Eigenschaft `l.isEmpty() = false`.
- Ist die Voraussetzung `l.size() > 0` erfüllt, so wird die Funktion `l.remove(0)` keine Exception werfen.

Verträge

Ein Vertrag für eine Methode besteht aus:

Vorbedingung: Welche Annahmen werden an die Argumente der Methode und den Zustand der Klasse gemacht.

Nachbedingung: Welche Leistung erbringt die Methode, d.h. welche Eigenschaft gilt nach ihrer Ausführung.

Effektbeschreibung: Welche Nebeneffekte hat die Ausführung der Methode (Exceptions, Ein- und Ausgabe, usw.)

Für uns genügt es, Verträge informell zu beschreiben.

Beispiel – Klasseninvarianten

Klasseninvariante Zu jedem Zeitpunkt erfüllt der Zustand der Klasse eine bestimmte Eigenschaft (die *Invariante*).

(Achtung: “zu jedem Zeitpunkt” fragwürdig, warum?)

Beispiel

- Klasse `Universitaet` mit privatem Feld

  ```
  private Map<Student, Set<Seminar>> seminareProStudent;
  ```

- Um herauszufinden, welche Studenten ein bestimmtes Seminar besuchen wird, muss man alle Studenten durchgehen.
- effizienter: Daten auch noch nach Seminaren indizieren.

  ```
  private Map<Seminar, Set<Student>> studentenInSeminar;
  ```

- Invariante: Die beiden Felder `seminareProStudent` und `studentenInSeminar` enthalten die gleichen Daten.

Beispiel – Klasseninvarianten

Implementierung von Klasseninvarianten

- Konstruktoren haben Invariante als Nachbedingung.

  Im Beispiel: `seminareProStudent` und `studentenInSeminar` beide anfangs leer

- Alle Methoden haben Invariante sowohl als Vor- als auch als Nachbedingung: Wenn die Invariante vor der Ausführung gilt, dann auch danach.

  Im Beispiel: Alle Methoden können annehmen, dass `seminareProStudent` und `studentenInSeminar` die gleichen Daten enthalten, müssen aber auch sicherstellen, dass das nach ihrer Ausführung immer noch gilt.

Überprüfung von Bedingungen mit `assert`

Bedingungen und Invarianten können mit der Java-Instruktion `assert` getestet werden.

Die Instruktion

```
assert test : errorValue;
```

prüft, ob der boolesche Ausdruck `test` zu `true` auswertet, und wirft eine `AssertionError`, wenn das nicht der Fall ist.

äquivalent:

```
if (!test) {
  throw new AssertionError(errorValue)
}
```

Warum benutzt man nicht die `if`-Abfrage?

Überprüfung von Bedingungen mit `assert`

`assert`-Instruktionen werden nur zum Testen benutzt.

- Standardmäßig werden Assertions ignoriert, d.h. sie haben keinen Einfluss auf die Progammgeschwindingkeit.
- Assertions müssen beim Programmstart ausdrücklich eingeschaltet werden.

  ```
  java -ea Main
  ```

  (`-ea` für „enable assertions“)

$\Rightarrow$ Das Programm sollte sich mit und ohne Assertions gleich verhalten.

`assert`-Instruktionen dienen auch der Dokumentation von Verträgen.

Assertions – Beispiele

Nachbedingung prüfen

```
class NichtUeberziehbaresKonto {
  private double kontoStand;
  ...
  public void abheben(double betrag) {
     ...
     assert (kontoStand >= 0) : "Interner Fehler " +
                                "Argument nicht richtig geprüft," +
                                "Konto überzogen";
  }
}
```

Unerreichbare Programmteile

```
  ...
  for (...) {
    if (...) return;
  }
  assert false; // Diese Instuktion sollte unerreichbar sein
```

Assertions – Beispiele

Argumente von öffentlichen Methoden nicht durch Assertions überprüfen.

öffentliche Methoden müssen Argumente immer selbst validieren

```
public void setX(int x) {
  if ((0 > x) || (x >= xmax)) {
    throw new IndexOutOfRangeException();
  }
  ...
}
```

Programm soll sich gleich verhalten, egal ob Assertions eingeschaltet sind oder nicht.

Assertions – Beispiele

Argumente von privaten Methoden

Private Methoden müssen falsche Argumente nicht immer mit Exceptions behandeln.

Da sie nur von der Klasse selbst benutzt werden können, kann man intern sicherstellen, dass sie nur mit guten Argumenten aufgerufen werden.

Dies kann man mit `assert` testen.

```
private void setX(int x) {
  assert (0 <= x) && (x < xmax);
  ...
}
```

Beispielprogramm: Schlechter Code

```
public enum Kartenfarbe { KREUZ, PIK, HERZ, KARO; };

int kartenKreuz;
int kartenPik;
int kartenHerz;
int kartenKaro;
int kartenAnzahl;

void nimm(Kartenfarbe farbe) {
  if(Kartenfarbe.KREUZ != farbe) {
    if(Kartenfarbe.KARO != farbe) {
      if(Kartenfarbe.PIK != farbe) {
        if(Kartenfarbe.HERZ != farbe) {
          kartenAnzahl++;
        } else {
          kartenAnzahl++;
          kartenHerz++;
        }
      } else {
        kartenAnzahl++;
        kartenPik++;
      }
    } else {
      kartenAnzahl++;
      kartenKaro++;
    }
  } else {
    kartenAnzahl++;
    kartenKreuz++;
  }
}
```

Beispielprogramm: Weniger kopierter Code

```
public enum Kartenfarbe { KREUZ, PIK, HERZ, KARO; };

int kartenKreuz;
int kartenPik;
int kartenHerz;
int kartenKaro;
int kartenAnzahl;

void nimm(Kartenfarbe farbe) {
  kartenAnzahl++;
  if(Kartenfarbe.KREUZ != farbe) {
    if(Kartenfarbe.KARO != farbe) {
      if(Kartenfarbe.PIK != farbe) {
        if(Kartenfarbe.HERZ != farbe) {
        } else {
          kartenHerz++;
        }
      } else {
        kartenPik++;
      }
    } else {
      kartenKaro++;
    }
  } else {
    kartenKreuz++;
  }
}
```

Beispielprogramm: Switch statt tiefe Verschachtelung

```
public enum Kartenfarbe { KREUZ, PIK, HERZ, KARO; };

int kartenKreuz;
int kartenPik;
int kartenHerz;
int kartenKaro;
int kartenAnzahl;

void nimm(Kartenfarbe farbe) {
  kartenAnzahl++;
  switch(farbe) {
    case KREUZ:
      kartenKreuz++;
      break;
    case PIK:
      kartenPik++;
      break;
    case HERZ:
      kartenHerz++;
      break;
    case KARO:
      kartenKaro++;
      break;
  }
}
```

Beispielprogramm: Map statt Fallunterscheidung

```
public enum Kartenfarbe { KREUZ, PIK, HERZ, KARO; };

Map<Kartenfarbe, Integer> karten;
int kartenAnzahl;

void nimm(Kartenfarbe farbe) {
  kartenAnzahl++;
  karten.put(farbe, karten.get(farbe) + 1);
}
```