Grundlagen der Programmierung

Dr. Christian Herzog
Technische Universität München

Wintersemester 2017/2018

Kapitel 6: Objektbasierter und imperativer Programmierstil

Objektbasierter und imperativer Programmierstil - Kapitelüberblick

- Konzepte **objektbasierter** Programmierung
  - Implementierung von Klassen
  - Instanziierung von Objekten (Konstruktor und **new**-Anweisung)
  - Instanzvariable
- Konzepte **imperativer** Programmierung
  - Zustand, Variable, Anweisung, Zuweisung
    - Gültigkeitsbereich von Variablen
    - Instanzvariable vs. lokale Variable
    - Seiteneffekte
  - Anweisungen
    - Zuweisung
    - Anweisungen zur Ablaufkontrolle (Bedingte Anweisung, Methodenaufruf, Schleifen)

Objektbasierte und imperative Programmierung

- Wir haben bisher schon einige Verfahren kennen gelernt, um Algorithmen und Systeme zu beschreiben:
  - Textersetzungssysteme, Markov-Algorithmen, Grammatiken und funktionale Programme.
- Diese Verfahren betrachten eine Berechnung als **Folge von Textersetzungen** oder als **Auswertung eines Ausdrucks**.
- Wir betrachten jetzt Verfahren, in denen Berechnungen als **Folge von Zustandsänderungen** angesehen werden können.
  - **Objektbasierte Programmierung:** Eine Berechnung ist eine Folge von Zustandsänderungen einer Menge von Objekten (als Instanzen einer Klasse).
  - **Imperative Programmierung:** Die Zustandsänderungen in einem Algorithmus werden durch Anweisungen erreicht.
- Wichtige Konzepte objektbasierter Programmierung haben wir bereits in Kapitel 3 kennen gelernt:
  - Klasse, Objekt, Konstruktor, Methodenaufruf

Java ist objektbasiert und imperativ

- Ein Java-Programm besteht aus einer Menge von *Klassendefinitionen*.
- Eine Klassendefinition enthält *Methodendeklarationen (method declaration)*.
- Eine Methodendeklaration besteht aus *Kopf (header)* und *Rumpf (body)*.
- Der Rumpf einer Methode besteht aus einer Menge von *Anweisungen*.
  - In Java gibt es verschiedene Typen von Anweisungen: Deklarationsanweisungen, bedingte Anweisungen, Zuweisungen, Return-Anweisungen, Schleifenanweisungen, Methodenaufrufe.
- *Mehrzeilen-* and *Einzelzeilen-Kommentare* kann man benutzen, um Java-Programme zu dokumentieren.
- Es gibt verschiedene Arten von Java-Programmen:
  - Java-Applikationen (betrachten wir in dieser Vorlesung)
  - Java Applets (werden von einem Browser aus gestartet)

Klassenentwurf

- Fünf wichtige Fragen, die wir beim Klassenentwurf stellen:
  - Was ist die *Aufgabe* der Klasse?
    (Die Aufgabenstellung wird beim Systementwurf beschlossen, und im detaillierten Entwurf verfeinert)
  - Welche *Information* braucht die Klasse, um ihre Aufgabe durchzuführen?
    - **Attribute**, **Variablen** => **Datenstrukturen**
  - Welche *Dienste* muss sie bereitstellen, um diese Information zu verarbeiten?
    - **Operationen, Methoden** => **Algorithmen**
  - Welche Dienste sind für andere Klassen öffentlich (*public*) verfügbar?
  - Welche Dienste sind versteckt und für andere Klassen nicht verfügbar (*private*)?

Beispiel für eine Problemstellung

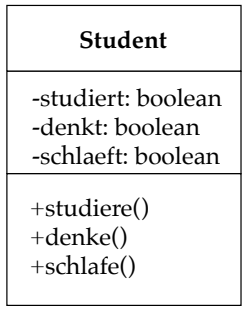
- Entwerfe und implementiere ein Programm, das das typische Verhalten fleißiger Studenten simuliert.
- Meinung von Domänenexperten:
  "Ein fleißiger Student führt drei Tätigkeiten durch:"
  - schlafen
  - denken
  - studieren

Analyse

- Welche *Klassen* brauchen wir, und welche *Dienste* sollen sie verrichten?
- Die Klasse `Student`
  - repräsentiert einen Studenten
  - Dienste: studiere(), denke() und schlafe().
- Die Klasse `Studentenverwaltungssystem`:
  - repräsentiert die Menge aller Studenten an der TUM
  - Dienste: kreiert Studenten und führt Zustandsänderungen an Studenten aus.

Klassenentwurf: `Student` (in UML)

- Zustand: 3 Attribute vom Typ `boolean`: `studiert`, `denkt` und `schlaeft`
- Drei Operationen:
  - Eine Operation `studiere()`, um Studenten in den Zustand `studiert` zu bringen.
  - Eine Operation `denke()`, um Studenten in den Zustand `denkt` zu bringen.
  - Eine Operation `schlafe()`, um Studenten in den Zustand `schlaeft` zu bringen.



Klassenspezifikation `Student`

Student
-studiert: boolean -denkt: boolean -schlaeft: boolean
+Student() +studiere() +denke() +schlafe()

- **Klassenname: `Student`**
  - Aufgabe: Repräsentation typischer Aktivitäten von Studenten

Attribute:

- **`studiert`**: Wird auf **`true`** gesetzt, wenn Student studiert (private)
- **`denkt`**: Wird auf **`true`** gesetzt, wenn Student denkt (private)
- **`schlaeft`**: Wird auf **`true`** gesetzt, wenn Student schläft (private)

Operationen:

- **`studiere()`**: Bringt ein Objekt vom Typ **`Student`** zum Studieren.
- **`denke()`**: Bringt ein Objekt vom Typ **`Student`** zum Denken.
- **`schlafe()`**: Bringt ein Objekt vom Typ **`Student`** zum Schlafen.
- **`Student()`**: Instanziiert ein Objekt vom Typ **`Student`**. (Kriegen wir "umsonst" in Java: Konstruktor)

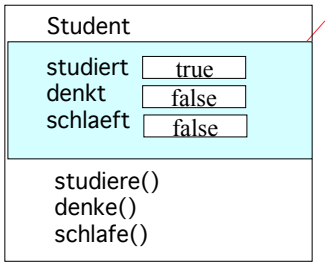
Implementierung der Klasse Student (in Java)

```
public class Student {
                                                  // Attribute:
    private boolean studiert;                     // Zustand des Studenten
    private boolean denkt;
    private boolean schlaeft;

    public Student() {                            // Konstruktor:
       studiere();                                // Setzt Zustand auf Studieren
    }                                             // Ende des Konstruktors

                                                  // Methoden:
    public void studiere()  {                     // Start von studiere()
         studiert = true;                         // Ändere  den Zustand
         denkt = false; schlaeft = false;
    }                                             // Ende von studiere()

    public void denke() {                         // Start von denke()
        denkt = true;                             // Andere den Zustand
        studiert = false; schlaeft = false;
  }                                               // Ende von denke()

    public void schlafe() {                       // Start von schlafe()
        schlaeft = true;                          // Andere den Zustand
        studiert = false; denkt = false;
  }                                               // Ende von schlafe()
}                                                 // Ende der Klasse Student
```

`void` bedeutet hier, dass die Methode kein Ergebnis liefert.

Die Klasse Student

- Eine Klasse ist eine Schablone für Objekte (Instanzen). In unserem Fall ist jeder Student im initialen Zustand **`studiert`**.
- Jede Instanz einer Klasse besitzt für jedes Attribut der Klasse eine instanz-lokale **Variable**, die in Java auch als Instanzvariable bezeichnet wird.



Den Instanzvariablen **`studiert`**, **`denkt`** und **`schlaeft`** werden vom Konstruktor Anfangswerte zugewiesen.

Konstruktoren

- Jede Java-Klasse hat mindestens eine Methode, die denselben Namen hat wie die Klasse. Diese Methode heißt **Konstruktor**.
- Der Zweck eines Konstruktors ist es, die notwendigen Initialisierungen bei der Instanziierung eines Objekts dieser Klasse durchzuführen.
  - Wenn man keinen Konstruktor angibt, dann gibt uns Java den so genannten Default-Konstruktor, der keine Argumente hat und keine spezielle Initialisierung durchführt.
- Der Konstruktor kann auch explizit deklariert werden.
  - Programmierregel für Grundlagen der Programmierung: Der Konstruktor wird explizit angegeben.

Wie funktioniert ein Konstruktor?

- Das Schlüsselwort **new** kreiert eine neue Instanz einer Klasse, d.h. es kreiert ein neues Objekt.

  zum Beispiel: `stud1 = new Student();`

  und ruft automatisch den Konstruktor `Student()` auf.

- Woher weiß die Methode `Student()`, auf welchen Daten sie operieren soll? Sie hat doch keine Parameter!
  - Wichtig: Wenn eine Methode auf die Attribute ihrer eigenen Klasse zugreift, dann kann sie auf diese direkt zugreifen (ohne einen vorangestellten Objektbezeichner).

  zum Beispiel:

```
public class Student {
    private boolean studiert;  // Attribut
    Student() {                // Konstruktor:
       studiert = true;        // Direkter Zugriff aufs Attribut
    }                          // Ende des Konstruktors
}                              // Ende der Klasse Student
```


Aufbau des Klassenkopfes (Class Header)

- Beispiel:

```
public class Student  // Klassenkopf
{                     // Start des Klassenrumpfes


}                     // Ende des Klassenrumpfes
```

- Allgemeiner Aufbau eines Klassenkopfes:

  KlassenModifikator$_{opt}$ class *KlassenName Superklasse*$_{opt}$

public class Student

Objekt-orientierte Programmierung (späteres Kapitel)

Imperative Programmierung

- Zustände werden durch die Ausführung von Anweisungen geändert.
- Die meisten heutigen objektbasierten Sprachen sind zugleich imperative Sprachen (d.h. sie enthalten Anweisungen)
- Beispiele für Programmiersprachen, die einen imperativen Programmierstil erlauben:
  - 1950-1960: Fortran, Algol, Cobol, Simula
  - 1970-1980: Pascal, C
  - 1980-1990: C++
  - 1990- : Java
- In diesem Vorlesungsblock beschreiben wir Java's imperative Eigenschaften (*Anweisungen, Ablaufstrukturen*).
  Java's objektorientierte Eigenschaften (*Vererbung, Polymorphismus*) werden wir in einem späteren Kapitel behandeln.

Der Begriff „Variable“ in der imperativen Programmierung

- **Definition (Programm-)Variable**: Eine (Programm-)Variable ist ein Paar (Bezeichner, Wert).
  - Zwei verschiedene Variablen können denselben Wert haben, aber ihre Bezeichner müssen unterschiedlich sein, sonst sind die Variablen nicht verschieden.
- Auf Variable kann man lesend und schreibend zugreifen.
  Dazu benötigt man eine Zugriffsfunktion.
- Beispiel:

```
int zaehler;
int k;
zaehler = 1;
zaehler = 5;
if (zaehler == 5) k = 5 else k = 7;
```

Funktionale vs. Imperative Programmierung

- In der **imperativen Programmierung** ist eine **Variable** eine Größe, die ihre Identität behält, aber ihren **Wert ändern kann**.
  - In der imperativen Programmierung ordnet die Zuweisung v = a der Variablen v den Wert a zu. Dieser Wert kann durch eine andere Zuweisung v = b überschrieben werden.
- In der **funktionalen Programmierung** ist eine Variable eine Größe (z.B. x), die durch Substitution (z.B. false für x) einen bestimmten, ab diesem Zeitpunkt **unveränderlichen** Wert erhält.
  - In der funktionalen Programmierung ordnet die Definition v = a der Variablen v den Wert a endgültig zu. Der Wert ist nach der Zuweisung unveränderlich.
    - (Im letzten Kapitel haben wir Variable nur als formale Parameter kennengelernt.)
  - In der imperativen Programmierung geht das übrigens auch: v heißt dann eine **Konstante**.

Instanzvariablen vs. Lokale Variablen

- Innerhalb einer Klasse unterscheidet Java
  - **Instanzvariablen** (instance variables), die auf Klassenebene als Attribute deklariert werden und in Objekten instanziiert werden.
  - **Lokale Variablen** (local variables), die innerhalb von Methoden oder allgemeiner innerhalb einer Verbundanweisung (eines Blocks, siehe später) deklariert werden können.
  - Auch die **formalen Parameter** gehören zu den lokalen Variablen einer Methode.
- Eine Instanzvariable kann einen Modifizierer haben (private, public), eine lokale Variable nicht.
- In nicht objektbasierten Programmiersprachen (z.B. C oder Pascal) nennt man Variablen, die außerhalb von Methoden (dort Funktionen und Prozeduren genannt) deklariert werden, auch **globale Variable**.

Deklaration von Instanzvariablen

- Beispiel:

```
// Instanzvariablen

private boolean studiert = true;
private boolean denkt = false;
private boolean schlaeft = false;
```

- Allgemeiner:

  Modifizierer$_{opt}$ *TypId Bezeichner Initialisierer*$_{opt}$

- **Gültigkeitsbereich (scope) von Instanzvariablen**: Instanzvariablen haben *Klassengültigkeitsbereich* (class scope), d.h ihre Namen können beliebig innerhalb der Klasse verwendet werden, in der sie definiert sind.

Deklaration von lokalen Variablen

- Beispiel:

```
public void nächsterZustand () {

    boolean b = studiert;    // lokale Variable

    studiert = schlaeft;
    schlaeft = denkt;
    denkt = b;
}
```

- Allgemeiner:

  TypId Bezeichner Initialisierer$_{opt}$

- **Gültigkeitsbereich von lokalen Variablen:** ab ihrer Deklaration bis zum Ende des Blocks, der sie umgibt.
- **Verschattung**: Wenn lokale Variable dieselbe Bezeichnung haben wie weiter außen deklarierte Variable, so unterbrechen sie deren Gültigkeit (nur die innerste ist gültig).

Zugriffskontrolle: public vs. private

- Instanzvariablen sind bei uns (Vorlesung und Übung) immer als private deklariert.
  - Dadurch sind sie für Objekte anderer Klassen nicht direkt zugreifbar.
- Methoden, die Zugriff auf geschützte Variablen erlauben sollen, werden als public deklariert.
- Die Menge der öffentlichen (public) Methoden definiert die *Schnittstelle* der Klasse, nämlich die Methoden, auf die von Objekten anderer Klassen zugegriffen werden kann.

Warum sollen Instanzvariablen "private" sein?

- Öffentliche Instanzvariablen können zu einem inkonsistenten Zustand führen.
- Beispiel: Nehmen wir an, wir definieren **`studiert`**, **`denkt`** und **`schlaeft`** als öffentliche Variablen:

```
public boolean studiert;
public boolean denkt;
public boolean schlaeft;
```

- Dann sind folgende Zugriffe erlaubt:

```
georg.studiert = false;
georg.denkt = true;
georg.schlaeft = true;      // Inkonsistenter Zustand!
```

- Die einzige richtige Art, Georg denken (und nicht schlafen) zu lassen, ist, die zugehörige Zugriffsmethode aufzurufen:

```
georg.denke();              // denke() ist public
```


Seiteneffekte

- Wenn eine Methode Instanzvariablen verändert, so nennt man das auch einen **Seiteneffekt** der Methode.
- Seiteneffekte sind erwünscht, wenn sie - wie auf den vorangegangenen Folien - den konsistenten Zugriff auf Instanzvariable erlauben.
- In Methoden, die ein Ergebnis liefern, durchbrechen Seiteneffekte das funktionale Konzept.
  - diese Methoden sehen wie Funktionen aus, sind aber keine Funktionen im Sinne von Kapitel 5.
  - Beispiel:

```
public boolean studentSchlaeft() {
   boolean erSchlaeft = schlaeft;
   studiere();
   return erSchlaeft;
}
```

- Solche Seiteneffekte sind mit Vorsicht zu behandeln.
  - In der Regel versagen die Beweismethoden aus Kapitel 5!

Anweisungen

- Anweisungen in der imperativen Programmierung:
  - Sie verändern den Zustand.
  - Sie regeln den Programmablauf (Kontrollfluss).
- Arten von Anweisungen:
  - Deklarationsanweisung
  - Zuweisung
  - Anweisungen zur Regelung des Kontrollflusses (Anweisungssequenzen, bedingte Anweisungen, Schleifen, Methodenaufruf, **`return`**-Anweisung)
  - **`new`** -Anweisung
  - ...

Deklarationen: Java-Bezeichner

- Ein Java-Bezeichner (*identifier*) ist ein Name für eine Variable, Methode, oder Klasse.
- Ein Java-Bezeichner muss mit einem Buchstaben beginnen, gefolgt von einer beliebigen Folge von Buchstaben, Zahlen und Unterstrich-Zeichen ('_').
- **Erlaubt:**
  - Student
  - studiere, schlafe,
  - Student1, Student_2
- **Nicht erlaubt:**
  - Cyber Student
  - 30Tage
  - Student$
  - n!

Deklaration von Methoden

- Allgemeiner Aufbau des Methodenkopfes

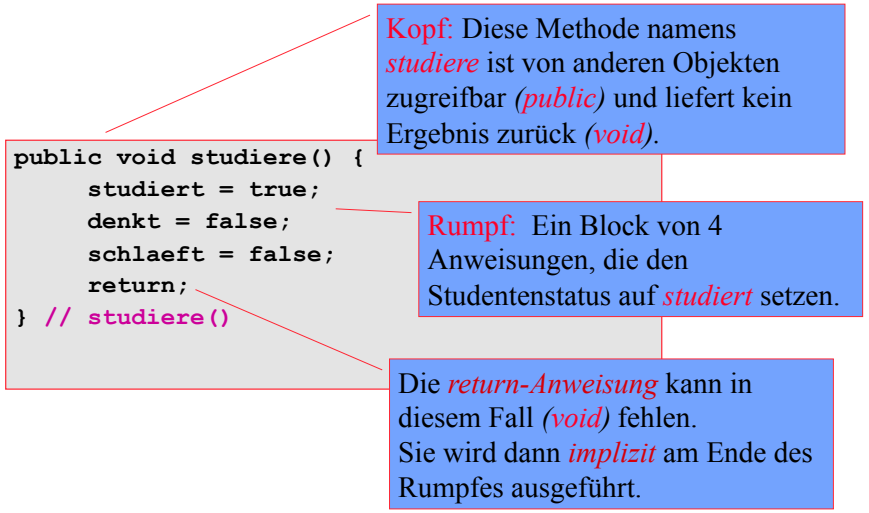
Modifizierer$_{opt}$ *ResultatTyp MethodenName* (*FormaleParameterListe*)

```
public static   void    main      (String[] argv)
public          void    paint     (Graphics g)
protected       int     zaehle    (Studentenverzeichnis s)
public          void    studiere  ()
```

- Definition einer öffentlichen Methode ohne Ergebnis

```
public void methodenName(...) // Methodenkopf
{                             // Beginn des Methodenrumpfs
   ...
}                             // Ende des Methodenrumpfs
```


Methodendeklaration



Die Zuweisungsanweisung

- Die grundlegende Operation auf einer Variablen in der imperativen Programmierung ist die Zuweisung.
- Allgemeine Form: *VariablenName = Ausdruck*
- Der *Ausdruck* auf der rechten Seite des Zuweisungsoperators wird ausgewertet, und der Wert wird in *VariablenName* auf der linken Seite gespeichert.
- Beispiel:

```
studiert = true;
denkt = false;
schlaeft = 100;      // Typ-Fehler
```

- Typ-Fehler (Type error): Der Typ des Wertes, der zuzuweisen ist, muss derselbe sein wie der Typ der Variablen (strong typing).

Zuweisungsoperator und Gleichheitsoperator in Java

- In Java wird für den Zuweisungsoperator das Gleichheitszeichen '=' verwendet. Die Gleichheitsüberprüfung wird durch '==' ausgedrückt.
- **Wichtig:** Ein oft gemachter Fehler ist die Benutzung des Java-Zuweisungsoperators "=" anstelle des Java-Gleichheitsoperators "==".
  - Beispiel: Die Anweisungen

```
boolean studiert = false;
if (studiert = true) {
   System.out.println("Student studiert.");
}
```

  sind syntaktisch korrekt, aber semantisch falsch.

Eine Zuweisung ist in Java immer auch ein Ausdruck. Ihr Typ und ihr Wert sind identisch dem Typ und dem Wert des Ausdrucks auf der rechten Seite.

- Semantische Fehler wie diese können von Compilern nicht gefunden werden!

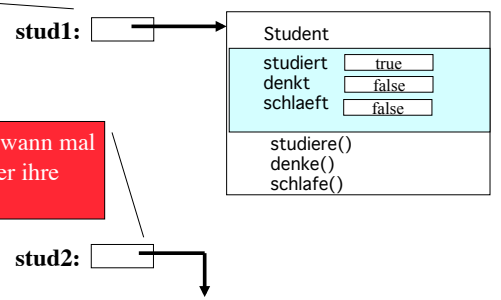
Zuweisung von Referenzen auf Objekte

- Eine *Variable* kann nicht nur Werte wie **`false`** oder **`100`** speichern, sondern auch einen Verweis ("Referenz", "Addresse") auf ein Objekt.

```
Student stud1, stud2;     // Deklaration von 2 Variablen

stud1 = new Student();    // Erzeugt Objekt vom Typ Student
```

Nach der Instanziierung verweist **`stud1`** auf ein Objekt vom Typ **`Student`**.

Die andere Variable **`stud2`** kann irgendwann mal auf ein **`Student`**-Objekt verweisen, aber ihre Referenz ist zur Zeit nicht definiert.



Zuweisung von Referenzen auf Objekte

- Eine *Variable* kann nicht
  sondern auch einen Verwe

```
Student stud1, stud
```

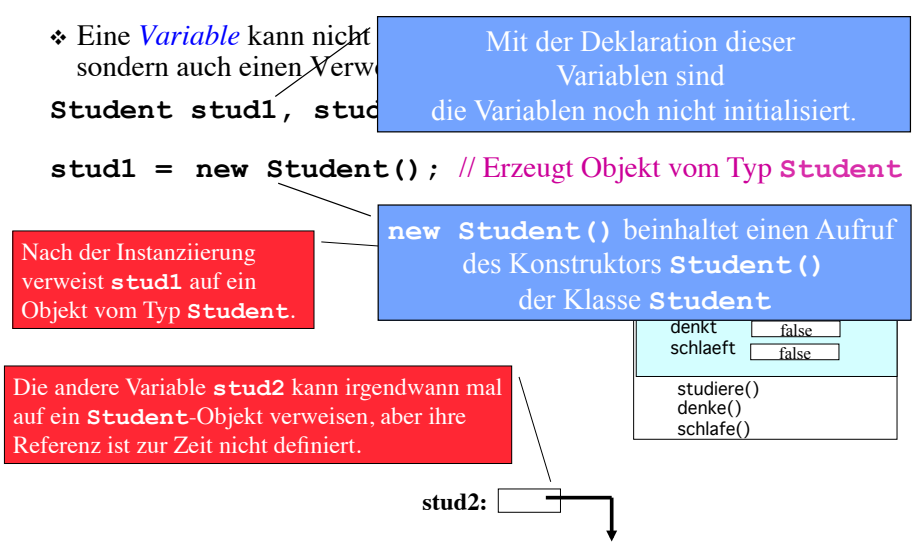
Mit der Deklaration dieser Variablen sind die Variablen noch nicht initialisiert.

```
stud1 = new Student();    // Erzeugt Objekt vom Typ Student
```

`new Student()` beinhaltet einen Aufruf des Konstruktors **`Student()`** der Klasse **`Student`**

Nach der Instanziierung verweist **`stud1`** auf ein Objekt vom Typ **`Student`**.

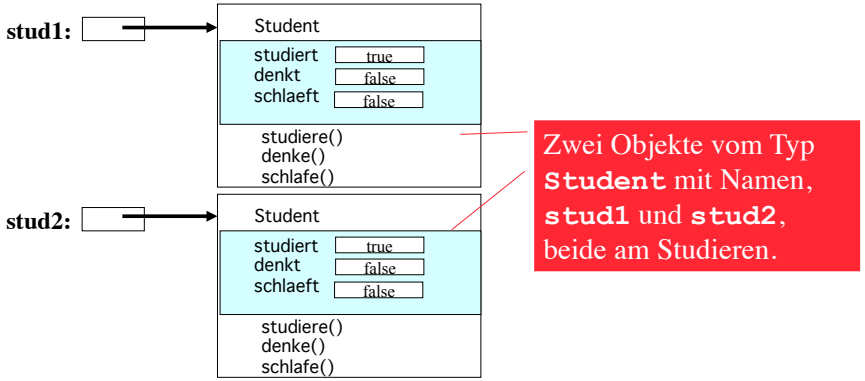
Die andere Variable **`stud2`** kann irgendwann mal auf ein **`Student`**-Objekt verweisen, aber ihre Referenz ist zur Zeit nicht definiert.



Erzeugung einer weiteren `Student`-Instanz

- Jetzt rufen wir auch für stud2 die Konstruktormethode auf:

```
Student stud1, stud2;
stud1 = new Student();
stud2 = new Student();
```

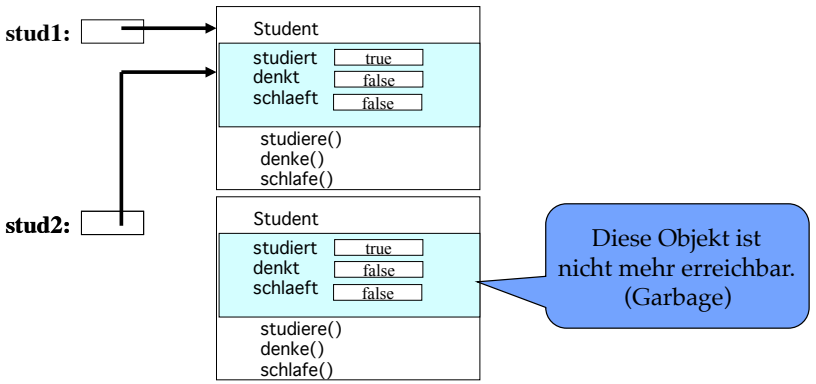


Zwei Objekte vom Typ **`Student`** mit Namen, **`stud1`** und **`stud2`**, beide am Studieren.

Zuweisung an Referenzvariablen

- Nun führen wir eine Zuweisung an die Variable stud2 aus:

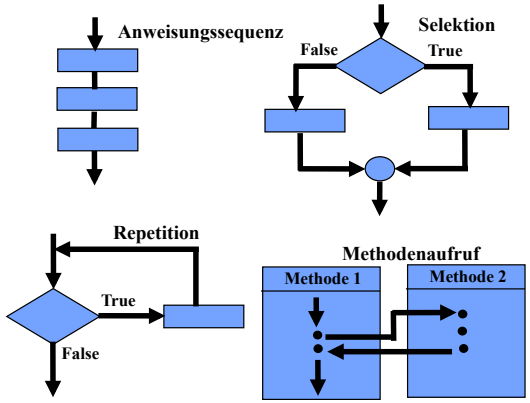
```
Student stud1, stud2;
stud1 = new Student();
stud2 = new Student();
stud2 = stud1;
```



Strukturierte Programmierung

- ***Definition Strukturierte Programmierung:*** Imperative Programme, die nur mit bestimmten Ablaufstrukturen (Typen von Anweisungen) geschrieben werden. Diese sind:
  - **Anweisungssequenz** --- Eine Folge von Anweisungen, die eine nach der anderen sequentiell ausgeführt werden.
  - **Selektion** --- Eine Anweisung, die eine Wahl zwischen zwei oder mehr Alternativen von Anweisungssequenzen erlaubt (Bedingte Anweisung (**if, if-else**), Fallunterscheidung (**switch**)).
  - **Repetition** (Schleife) --- Eine Anweisung, die es erlaubt, eine Anweisungssequenz zu wiederholen (**for, while,** und **do-while** Struktur).
  - **Methodenaufruf** --- Eine Anweisung, die die Kontrolle im Programm zur benannten Methode überträgt. Wenn diese Methode ausgeführt worden ist, wird die Ausführung an der Stelle unmittelbar nach dem Methodenaufruf fortgesetzt.
- **Streng verboten** (auch wenn in den meisten Sprachen möglich) **ist**:
  - **goto**

Die 4 Konstrukte der Strukturierten Programmierung



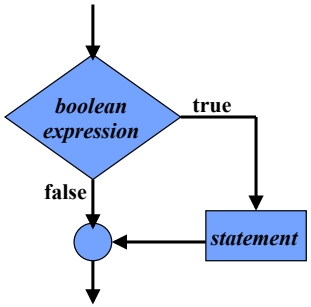
- Egal, wie groß oder klein ein Programm ist, es kann nur durch eine beliebige Kombination dieser 4 Konstrukte beschrieben werden.
- Jede dieser Konstrukte hat genau einen Eingang (entry) und genau einen Ausgang (exit).

Anweisungssequenz: Anweisungen und Verbundanweisungen

- Eine Anweisungssequenz besteht aus einer Menge von Anweisungen (statements). Die Reihenfolge, in der diese Anweisungen ausgeführt werden, bestimmt den Kontrollfluss im Programm.
- In einer Anweisungssequenz werden Anweisungen von anderen Anweisungen durch Strichpunkt „;“ getrennt.
- Ein **Block**, auch **Verbundanweisung** (compound statement) genannt, ist eine Folge von Anweisungen, die durch Klammern "{" und "}" eingegrenzt sind.
  - Ein Block selbst ist dann wieder eine Anweisung.
- Beispiele von Anweisungssequenzen:
  - `i = i + 1; i = 1; j = 3;`
  - `stud1.denke();`
  - `{int i = 1; int j = i + 1; k = i + j;}`

Ein Block mit lokalen Variablen `i` und `j`.

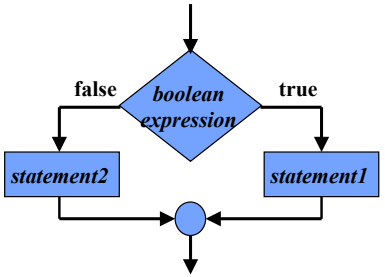
Bedingte Anweisungen: Die `if`-Anweisung



In Java:
if (*boolean expression*)
statement ;

- Wenn der boolesche Ausdruck "*boolean expression*" wahr ist, dann wird "*statement*" ausgeführt. Sonst wird es übergangen.

Bedingte Anweisungen: Die `if-then-else`-Anweisung



In Java:
if (*boolean expression*)
statement1;
else
statement2 ;

- Wenn der boolesche Ausdruck "*boolean expression* " wahr ist, führe "*statement1*" aus, sonst "*statement2*".

Bedingte Ausdrücke und bedingte Anweisungen

- Bedingte Ausdrücke hatten wir schon in der funktionalen Programmierung kennengelernt:
  - `return boolean_expression ? A1 : A2;`
  - Beispiel:

```
int abs(int x) {
   return x < 0 ? -x : x;
}
```

- Solche bedingte Ausdrücke werden wir nun immer als bedingte Anweisungen (und damit besser lesbar) formulieren:

```
int abs(int x) {
  if (x < 0)
      return -x ;
  else
      return x;
}
```


Beispiele von bedingten Anweisungen

Einfaches if

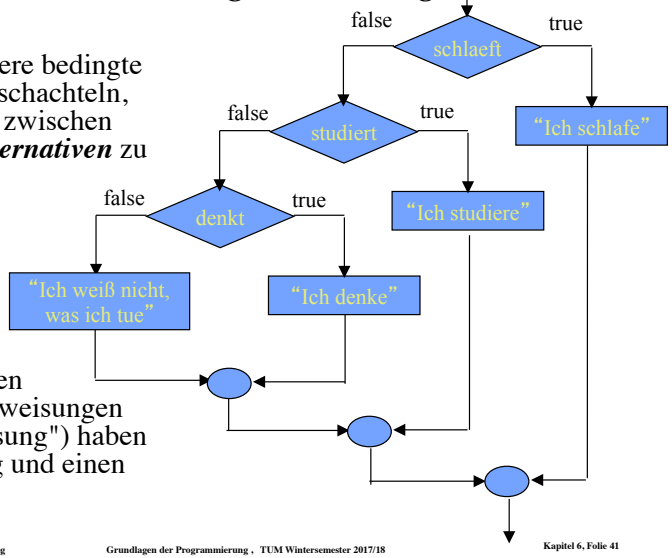
```
if (studiert)
    System.out.println("Ist am Studieren");
```

if-then-else

```
if (studiert)
    System.out.println("Ist am Studieren");
else
    System.out.println("Ist nicht am Studieren");
```

Mehrfach geschachtelte bedingte Anweisungen

- Wir können mehrere bedingte Anweisungen verschachteln, um eine Auswahl zwischen ***mehr als zwei Alternativen*** zu realisieren.



- Auch komplizierten verschachtelte Anweisungen ("Mehrweganweisung") haben nur einen Eingang und einen Ausgang.

Beispiel einer Mehrweganweisung …

Mehrweg-Auswahl

```
if (schlaeft)
        System.out.println("Ich schlafe.");
else if (studiert)
        System.out.println("Ich studiere.");
else if (denkt)
        System.out.println("Ich denke.");
else
        System.out.println(
                  "Ich weiß nicht, was ich tue.");
```


Das "hängendes-`else`"-Problem

Was wird hier gedruckt, wenn **`condition1 == false`** gilt?

```
if (condition1)
  if (condition2)
    System.out.println("Eins");
else
  System.out.println("Zwei");
```

"Falsche" Einrückung

```
if (condition1)
  if (condition2)
    System.out.println("Eins");
  else
    System.out.println("Zwei");
```

Korrekte Einrückung

- Als Programmierer müssen Sie sorgfältig darauf achten, dass jedes **`else`** zu seinem korrespondierenden **`if`** gehört.
- Regel: Eine **`else`**-Klausel gehört immer zur innersten **`if`**-Klausel.
- Einrückungen sollen die Logik Ihres Programms reflektieren, es lesbarer machen, aber….
  - Einrückungen werden vom Compiler ignoriert.

Vermeiden Sie hängendes `else`

- Verwenden Sie immer Verbundanweisungen, d.h. geschweifte Klammern, um die **`then`**- und **`else`**-Zweige zu kennzeichnen, auch wenn diese nur aus einer Anweisung bestehen:

`Nicht so gut:`

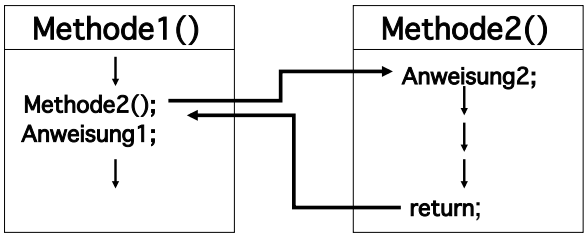
```
if (condition1)
  if (condition2) {
    System.out.println("Eins");
    System.out.println("wahr");
  }
  else
    System.out.println("Zwei");
```

`Empfehlenswert:`

```
if (condition1) {
  if (condition2) {
    System.out.println("Eins");
    System.out.println("wahr");
  }
  else {
    System.out.println("Zwei");
  }
}
```

Methodenaufruf und `return`

- Ein Methodenaufruf verursacht einen Transfer der Kontrolle innerhalb eines Programms zur ersten Anweisung in der aufgerufenen Methode.
- Eine **`return`**-Anweisung (*return statement*) bringt die Kontrolle wieder zurück zur Anweisung, die den Aufruf verursacht hat.



- Eine **`return`**-Anweisung kann auch benutzt werden, um eine Methode vorzeitig zu verlassen.
  - Z.B. zur Vermeidung geschachtelter **`if-else`**-Konstrukte

`return`-Anweisung zur Vermeidung geschachtelter `if-else`-Konstrukte

- Eine Methode mit geschachteltem **`if-else`**-Konstrukt:

```
void meineMethode () {
 if (Bed1) {
   if (Bed2) {
     if (Bed3) {
       <<eigentlicher Rumpf>>
     }
     else {
       System.out.print("Bed3!");
     }
   }
   else {
     System.out.print("Bed2!");
   }
 }
 else {
   System.out.print("Bed1!");
 }
 return;
}
```

- Dieselbe Methode mit **`return`**-Anweisungen:

```
void meineMethode () {
 if (! Bed1) {
   System.out.print("Bed1!");
   return;
 }
 if (! Bed2) {
   System.out.print("Bed2!");
   return;
 }
 if (! Bed3) {
   System.out.print("Bed3!");
   return;
 }
 <<eigentlicher Rumpf>>
 retur
}
```

Diese Stelle wird nur erreicht, wenn alle 3 Bedingungen wahr sind.

Schleifenstrukturen (Wiederholungsanweisungen)

- **Definition Schleifenrumpf** (*loop body*)**:** Eine Anzahl von Anweisungen, die innerhalb der Schleifenanweisung durchlaufen wird.
- **Definition Schleifeneintrittsbedingung** (*loop entry condition*)**:** Muss wahr sein, damit der Schleifenrumpf (erneut) ausgeführt wird.
- In der strukturierten Programmierung gibt es 3 Typen von Schleifenstrukturen:
  - **Zählschleife:** Eine Wiederholungsanweisung, in der bereits vor Beginn der Schleifenausführung klar ist, wie oft der Schleifenrumpf insgesamt durchlaufen werden muss.
  - **`while`-Schleife:** Eine Wiederholungsanweisung, in der die Schleifenbedingung ***vor jedem Eintritt in den Schleifenrumpf*** abgefragt wird.
  - **`do-while`-Schleife:** Eine Wiederholungsanweisung, in der die Schleifenbedingung ***nach jeder Ausführung des Schleifenrumpfes*** abgefragt wird.

Die Zählschleife

- Allgemeine Struktur:

```
for (Zählerinitialisierung;
     Schleifeneintrittsbedingung;
     Weiterschaltung)
  Schleifenrumpf;
```

- Beispiel:

```
for ( int k = 0; k < 100; k = k+1 )
  System.out.print("Hello");
```

k ist eine lokale Variable der Schleife.

Der Schleifenrumpf kann eine einzelne Anweisung oder eine Verbundanweisung (Block) sein.

Gültigkeitsbereich für Schleifenvariablen

- Eine Schleifenvariable ist eine lokale Variable.
- Wenn **k** innerhalb der **for**-Anweisung deklariert ist, kann es nicht außerhalb des Schleifenrumpfes verwendet werden:

k++ ist Abkürzung für **k = k+1**

```
for (int k = 0; k < 100; k++)
     System.out.println("Hello");
System.out.println(k);    // Syntaxfehler: k nicht deklariert
```

- Wenn **k** außerhalb der **for**-Anweisung deklariert ist, kann es auch außerhalb benutzt werden:

```
int k;                          // Deklaration der Schleifenvariable
for (k = 0; k < 100; k++)
    System.out.println("Hello");
System.out.println(k);          // Jetzt kann k außerhalb des
                                // Schleifenrumpfes benutzt werden
```

ausgedruckt wird **100**

Zählschleifen können nach oben oder unten zählen

- Eine Zählschleife initialisiert den **Zähler** mit einem Anfangswert und zählt 0 oder mehr Iterationen, bis die Grenze der Schleife (*loop bound*) erreicht ist.
- Die **Schleifeneintrittsbedingung** testet, ob die Grenze der Schleife erreicht worden ist.

k-- ist Abkürzung für **k = k-1**

```
public void countdown() {
  for (int k = 10; k > 0; k--)
     System.out.println(k);
  System.out.println("Start");
} // countdown()
```

- Bei der **Weiterschaltung** muss ein Fortschritt in Richtung Schleifengrenze erzielt werden, sonst gibt es eine unendliche Schleife.

Unendliche Zählschleifen

- Beispiele

k += 2 ist Abkürzung von **k = k+2**

```
for (int k = 0; k < 100 ; k--)      // Werte von k: 0,-1,-2, …
    System.out.println("Hello");

for (int k = 1; k != 100 ; k += 2) // Werte von k: 1,3,…,99,101,…
    System.out.println("Hello");
```

- In beiden Fällen macht die *Weiterschaltung* überhaupt keinen Fortschritt in Richtung Schleifengrenze und die Schleifeneintrittsbedingung wird deshalb nie **false**.
- Vergleiche: Terminierung rekursiver Funktionen (Kapitel 5)

Geschachtelte Zählschleifen

- Nehmen wir an, Sie sollen folgende Tabelle mit 4 Zeilen und 9 Spalten drucken:

```
1  2  3  4  5  6  7  8  9
2  4  6  8  10 12 14 16 18
3  6  9  12 15 18 21 24 27
4  8  12 16 20 24 28 32 36
```

- Hier können Sie eine **geschachtelte** Zählschleife verwenden:
  Die **äußere Schleife** (*outer loop*) druckt die 4 Zeilen.
  Die **innere Schleife** (*inner loop*) druckt die 9 Spalten.

Konkatenation auf Strings

```
for (int row = 1; row <= 4; row++) {     // Für jede der 4 Zeilen
  for (int col = 1; col <= 9; col++) {   // Für jede der 9 Spalten
    System.out.print(col * row + "\t");
  }              // innere Schleife: Druckt insgesamt 36 Zahlen
  System.out.println();         // Starte eine neue Zeile
}                // äußere Schleife
```

Beispiel: iterative Fassung der Funktion summe

- Rekursive Fassung der Funktion **`summe`** aus Kapitel 5:

```
int summe (int n) {
  return n==0 ? 0
              : summe(n-1) + n;
}
```

- Iterative Fassung der Funktion **`summe`** mit **`for`**-Schleife:

```
int summe (int n) {
  int result = 0;
  for (int i=1; i<=n; i++) {
    result += i;
  }
  return result;
}
```


Analog: iterative Fassung der Funktion fakultaet

- Rekursive Fassung der Funktion **`fakultaet`** aus Kapitel 5:

```
int fakultaet (int n) {
  return n==0 ? 1
              : fakultaet(n-1) * n;
}
```

- Iterative Fassung der Funktion **`fakultaet`** mit **`for`**-Schleife:

```
int fakultaet (int n) {
  int result = 1;
  for (int i=1; i<=n; i++) {
    result *= i;
  }
  return result;
}
```


Beispiel: iterative Berechnung der Fibonacci-Zahlen

- Rekursive Berechnung der Fibonacci-Zahlen aus Kapitel 5:

```
int fib (int n) {
  return n==0 ? 0
              : n==1 ? 1
                     : fib(n-1) + fib(n-2);
}
```

- Iterative Berechnung der Fibonacci-Zahlen mit einer Zählschleife:

```
int fib (int n) {
  if (n <= 1)
    return n;
  int fibNminus1 = 1, fibNminus2 = 0;
  for (int i=2; i<n; i++) {
    int fibNminus1Alt = fibNminus1;
    fibNminus1 = fibNminus1 + fibNminus2;
    fibNminus2 = fibNminus1Alt ;
  }
  return fibNminus1 + fibNminus2;
}
```

Mehrere Deklarationen vom selben Typ können durch Komma getrennt angegeben werden

Lokale Variable im Schleifen-rumpf

Kann man abkürzen mit `fibNminus1 += fibNminus2;`

Die while-Schleife

- Allgemeine Struktur:

```
while (Schleifeneintrittsbedingung)
     Schleifenrumpf;
```

- Beispiel (Zählschleife als **`while`**-Schleife):

```
int k = 0;
while (k < 100) {
  System.out.println("Hello");
  k++;
}
```

Beispiel: iterative Fassung des ggT

- Rekursive Fassung des ggT aus Kapitel 5:

```
int ggT (int a, int b) {
 return a==b ? a
             : a>b ? ggT(a-b, b)
                   : ggT(a, b-a);
}
```

- Umwandlung der repetitiven Rekursion in eine **`while`**-Schleife:

```
int ggT (int x, int y) {
  int a=x; int b=y;
  while (a != b) {
    if (a > b)
      a = a-b;
    else
      b = b-a;
  }
  return a;
}
```


Beispiel: iterative Berechnung der Länge einer Sequenz

- Rekursive Berechnung der Länge einer Integer-Sequenz aus Kapitel 5:

```
int laenge (IntSequenz s) {
   return isEmpty(s) ? 0 : laenge(rest(s)) + 1;
}
```

- Iterative Berechnung der Länge mit einer **`while`**-Schleife:

```
int laenge (IntSequenz s) {
  int result = 0;
  IntSequenz r = s;
  while (!isEmpty(r)) {
    result++;
    r = rest(r);
  }
  return result;
}
```


Zählschleife zur Fortschaltung eines Pegels

- Betrachten wir noch einmal die gerade angegebene Methode:

```
int laenge (IntSequenz s) {
  int result = 0;
  IntSequenz r = s;
  while (!isEmpty(r)) {
     result++;
     r = rest(r);
  }
  return result;
}
```

- Die Variable **`r`** übernimmt hier die Rolle eines Pegels, der durch eine lineare, rekursive Datenstruktur geschoben wird.
- In diesem Fall wird oft eine **`for`**-Schleife verwendet. Auch wir wollen diesen Ausnahmefall bei der strukturierten Programmierung zulassen:

```
int laenge (IntSequenz s) {
  int result = 0;
  for (IntSequenz r=s; !isEmpty(r); r=rest(r))
     result++;
  return result;
}
```


Allgemein: Zählschleife als while-Schleife

- In Java kann die Zählschleife als abgekürzte Schreibweise einer **`while`**-Schleife aufgefasst werden:

```
for (Zählerinitialisierung;
     Schleifeneintrittsbedingung;
     Weiterschaltung)
  Schleifenrumpf;
```

ist Abkürzung für:

```
{
  Zählerinitialisierung;
  while (Schleifeneintrittsbedingung) {
    Schleifenrumpf;
    Weiterschaltung;
  }
}
```

Der äußere Block wird benötig, um Schleifen-lokale Variablen deklarieren zu können.

In der strukturierten Programmierung verwenden wir die for-Schleife nur als Zählschleife oder zum Fortschalten eines Pegels!

- Damit kann vieles (fast alles), was als **`while`**-Schleife formuliert wird, auch als **`for`**-Schleife formuliert werden!

Die do-while-Schleife

- Allgemeine Struktur:

```
do
 Schleifenrumpf;
while (Schleifeneintrittsbedingung)
```

- Der Schleifenrumpf wird mindestens einmal ausgeführt.
  - Er wird bereits ausgeführt, ehe die Schleifeneintrittsbedingung zum ersten Mal getestet wird.
- Der Schleifenrumpf wird solange ausgeführt, bis die Schleifeneintrittsbedingung nicht mehr wahr ist.

Das Standardbeispiel für eine `do-while`-Schleife: Validierung von Eingabedaten

- **Problem:** Mache es unmöglich, fehlerhafte Klausurergebnisse einzugeben. Gültige Ergebnisse: 0 Punkte bis 40 Punkte (weder -10 Punkte noch 55 Punkte sind akzeptierbare Ergebnisse)
- **Algorithmus:** Benutze eine **`do-while`**-Schleife für diese Aufgabe, denn der Benutzer benötigt eventuell mehr als einen Versuch, ein gültiges Ergebnis einzugeben.
- Der Algorithmus in „Pseudocode“:

```
do {
    <<get the next grade from user>>;     // Initialisierung
    if (grade < 0 || grade > 40)          // Fehlerfall
        <<print an error message>>;
} while (grade < 0 || grade > 40);        // Wächtertest
```


Zusammenfassung Schleifenstrukturen

- Eine **Zählschleife** (**`for`-Schleife**) sollte man benutzen, wenn man von vornherein weiss, wieviele Iterationen benötigt werden, oder wenn eine lineare Datenstruktur „durchlaufen“ wird.
- Eine **`while`-Schleife** sollte man verwenden, wenn der Schleifenrumpf eventuell überhaupt nicht ausgeführt werden soll.
- Eine **`do-while`-Schleife** sollte man verwenden, wenn eine oder mehr Iterationen durchgeführt werden.
- Eine **unendliche Schleife** ist das Resultat einer fehlerhaft spezifizierten Initialisierung, Weiterschaltung oder einer schlecht gewählten Schleifeneintrittsbedingung
  - oder sie ist beabsichtigt.

Einschub: Implementierungssheuristiken ("Gute Ratschläge")

- **Modularität:**
  - Code, der wiederholt an mehreren Stellen im Programm auftritt, sollte in einer Methode zusammengefasst werden.
    Das reduziert die Redundanz und erleichtert das Testen/Debugging und spätere Modifikationen des Codes.
- **Für jede Aufgabe eine Methode:**
  - Jede Methode sollte nur eine Aufgabe implementieren.
  - Sehr lange Methoden (mehr als eine Seite), sind oft ein Hinweis, dass man zu viele Aufgaben in eine Methode gesteckt hat.
- **Benutzerschnittstelle:**
  - Benutze eine **Eingabeaufforderung** (*prompt*), um den Benutzer zu informieren, sobald das Programm eine Eingabe erwartet.
  - Informiere den Benutzer, welche Art von Eingabe erwartet wird.
  - Zeige dem Benutzer positiv an, dass seine Eingabe akzeptiert ist.