

*Einführung in die Informatik I*  
*Imperative Programmierung*

Prof. Bernd Brügge, Ph.D  
Technische Universität München

Wintersemester 2000/2001  
11. Dezember 2000

# *Überblick über den nächsten Vorlesungsblock*

## ❖ **11./12. Dezember:**

- Der Begriff der Variable
- Strukturierte Programmierung: 4 Kontrollstrukturen
- Methodenaufruf
- Kontrollstrukturen für Schleifen
  - Zählschleife, konditionale Schleifen

## ❖ **Nächste Woche:**

- Einfache Sortier- und Suchalgorithmen
- Rekursion vs Iteration

## ❖ **Wichtigstes Ziel dieses Vorlesungsblockes:**

Vertrautheit mit Java-Applikationen, Begriff der Variable, grundlegende Java-Anweisungen, Prinzipien der Schleifenprogrammierung, Such- und Sortieralgorithmen

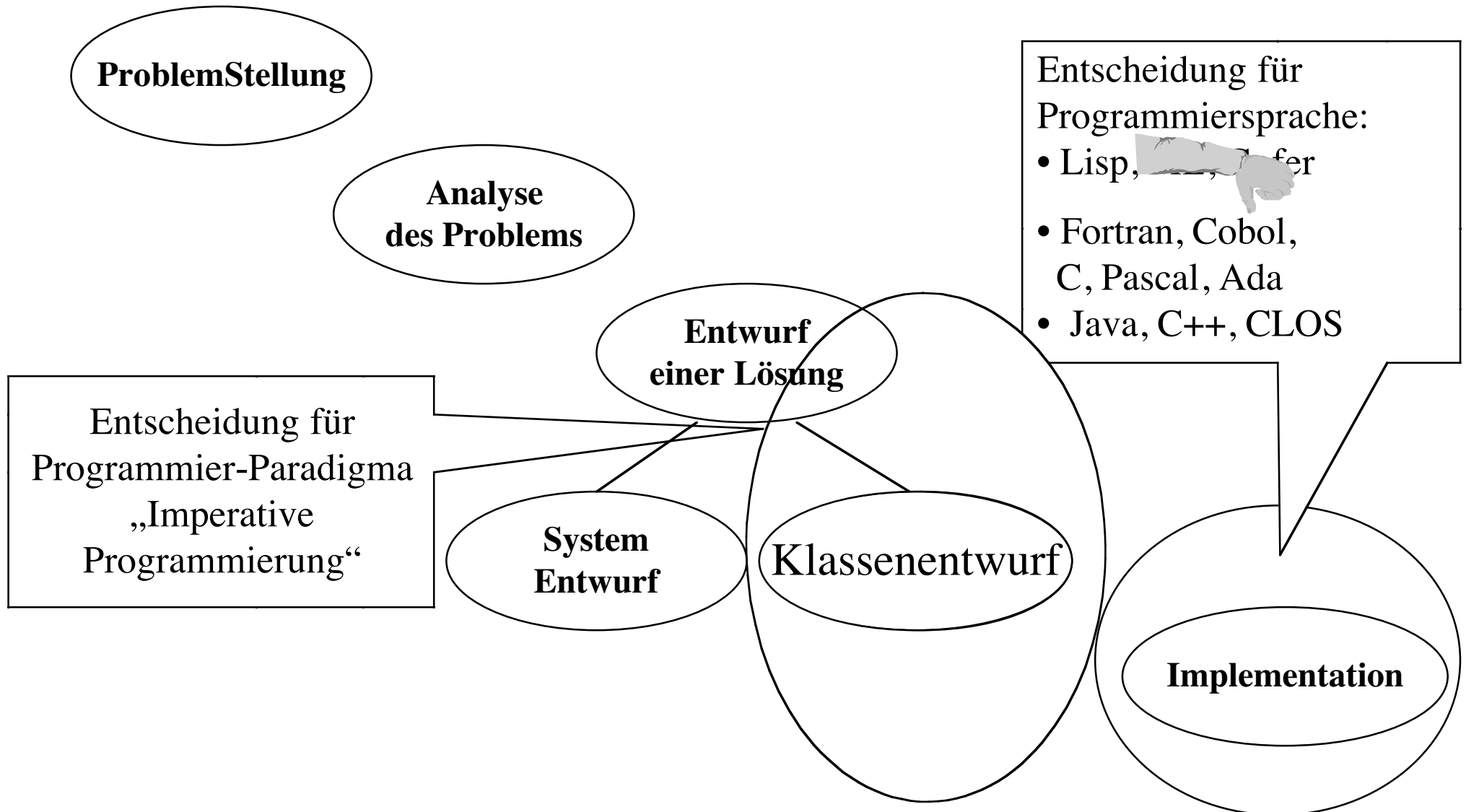
## *Wo stehen wir?*

- ❖ **Oktober/November:** Modellierung, Informatik Systeme, Textersetzungssystem, Markov-Algorithmen, Algebra, Boolesche Algebra, Aussagenlogik, Termersetzungssysteme
- ❖ **Rest des Semesters:** Programmierparadigmen
  - Funktionale Programmierung (Ende November)
  - **jetzt:** Imperative Programmierung (Dezember)
  - Objekt-Orientierte Programmierung (Januar)
  - Ereignis-basierte Programmierung (Ende Januar)
- ❖ **Ergänzungen im Sommersemester:**
  - Ereignis-basierte Programmierung
  - Regel-basierte Programmierung

## *Was kommt nun?*

- ❖ Wir haben bisher schon zahlreiche Verfahren kennengelernt, um Algorithmen und Systeme zu beschreiben:
  - Markov-Algorithmen, Semi-Thue-Systeme, Termersetzungssysteme, Formeln der Aussagenlogik und funktionale Programme.
- ❖ Einige Verfahren beschreiben Beziehungen zwischen verschiedenen Größen. Andere Verfahren überprüfen, ob eine Menge von Größen eine bestimmte Beziehung erfüllt. Andere spezifizieren Abbildungen oder deren schrittweise Berechnung.
- ❖ Wir betrachten jetzt Verfahren, in denen Berechnungen als Zustandsfolgen angesehen werden können.
  - **Imperative Programmierung:** Die Zustandsänderungen in einem Algorithmus werden durch Anweisungen erreicht.

# Aktivitäten bei der Entwicklung eines Informatik-Systems



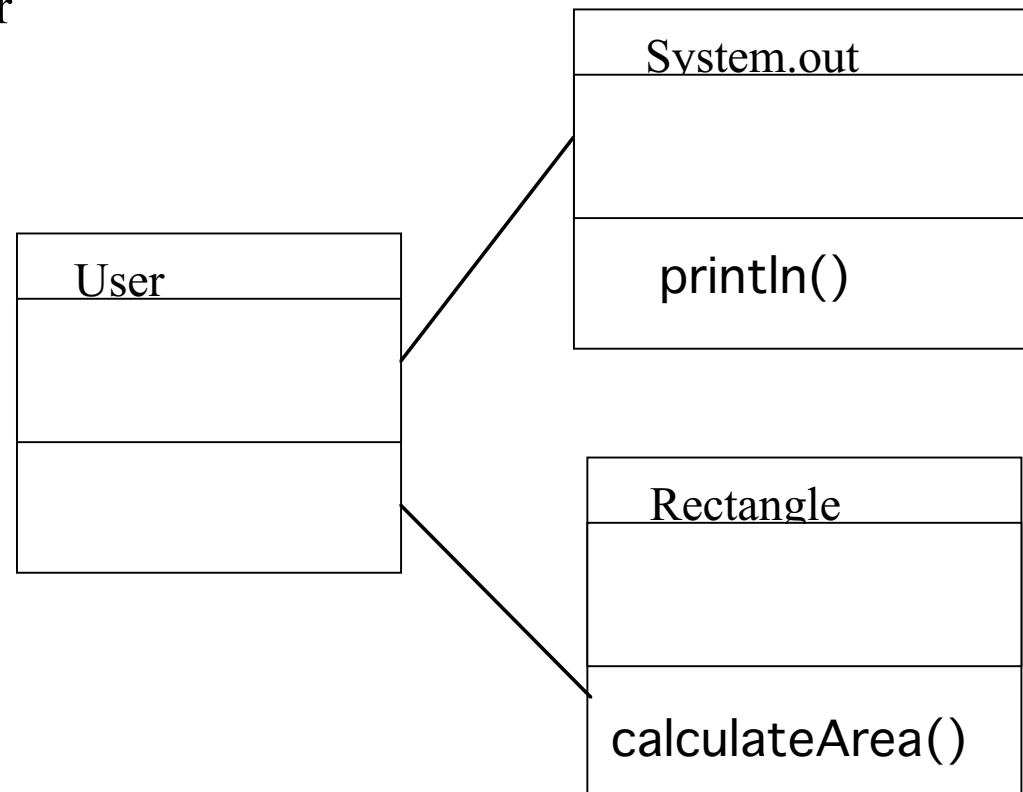
# *Detaillierter Entwurf = Klassenentwurf*

- ❖ 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?
    - **Zustand (Variablen, Datenstrukturen)**
  - Welche *Dienste* muss sie bereitstellen, um diese Information zu verarbeiten?
    - **Methoden (Algorithmen)**
  - Welche von den Informationen ist für andere Klassen öffentlich (*public*) verfügbar?
  - Welche von den Informationen ist versteckt (*information hiding*) und für andere Klassen nicht verfügbar (*private*)?

# *Was ist Klassenentwurf? Wo fängt er an?*

## *Ein sehr einfaches Beispiel:*

- ❖ **Problemstellung:** Ein Architekt möchte ein System haben, welches für beliebige Rechtecke die Fläche berechnet.
- ❖ **Analyse:** Wir definieren eine Klasse *rectangle*, die eine Operation *calculateArea()* bereitstellt.
- ❖ **Systementwurf:** Das CAD-System (Computer Aided Design) besteht aus 3 Klassen:
  - *User*
  - *Rectangle* mit der Schnittstelle *calculateArea()*
  - *System.out*: Schnittstelle *println()*



# *Klassentwurf der Rectangle Klasse*

- ❖ *Aufgabe* der Klasse (aus der Analyse und Systementwurf):
  - Berechnung der Fläche eines Rechteckes für einen Benutzer
- ❖ *Information* (Variablen)
  - Höhe und Breite des Rechteckes
- ❖ *Methoden*
  - Algorithmus für calculateArea():
    - Fläche = Höhe mal Breite
- ❖ Öffentliche Information (*public*):
  - Funktionalität calculateArea(): Berechnung der Fläche
- ❖ *Private Information* (*private*):
  - Höhe und Breite des Rechteckes

Zur Spezifikation des Algorithmus verwenden wir "Pseudocode", den wir während der Implementation in Java-Code umwandeln.

# *Klassenspezifikation: Rectangle Klasse*

## ❖ Variablen:

- Länge (length) und Breite (width) des Rechteckes

## ❖ *Methoden*: calculateArea()

- $area = (length * width)$

## ❖ *Öffentlich*:

- Berechnung der Fläche (calculateArea)

## ❖ *Privat*:

- Höhe und Breite des Rechteckes (length, width)

<b>rectangle</b>
private int length private int width
public int calculateArea()

# *Implementation des Klassenentwurfs: Definition, Instantiierung, Aufruf*

## ❖ **Klassendefinition:**

Definition aller Klassen aus dem Systementwurf  
(Rectangle, User)

## ❖ **Objektinstantiierung:**

Kreation von Objekten als Instanzen von Klassen  
(rectangle1, rectangle2)

## ❖ **Objektaufruf:**

Aufruf von Objektmethoden, die den Dienst des Objektes realisieren  
(rectangle1.calculateArea())

# *Klassendefinition von Rectangle in Java*

Rectangle ist public, ist also für andere Klassen zugreifbar.

```
public class Rectangle
{
    private int length; // Instance variables
    private int width;

    public Rectangle(int l, int w) // Konstruktor Methode (Constructor method)
    {
        length = l;
        width = w;
    } // Rectangle constructor

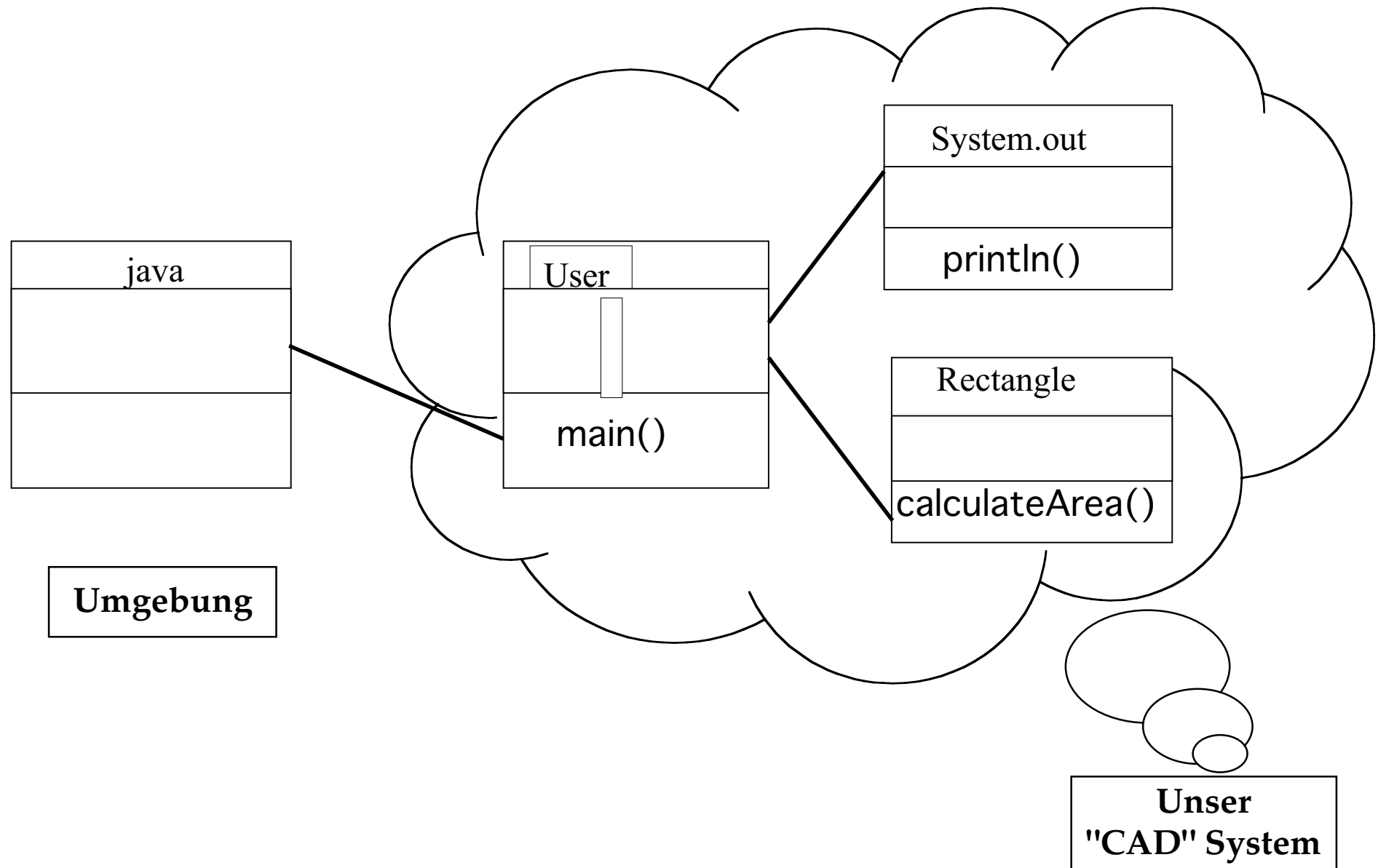
    public double calculateArea() // Zugriff Methode (access method)
    {
        return length * width;
    } // calculateArea

} // Rectangle class
```

Attribute sind immer private (in Info I)

Die Schnittstelle (interface) einer Klasse ist die Menge aller ihrer öffentlichen (public) Methoden

# Der Systementwurf für unser "CAD" System



## *Definition Java-Programm*

- ❖ Ein Java-Programm besteht aus einer Menge von *Klassendefinitionen*.
- ❖ Eine Klassendefinition besteht aus *Kopf (header)* und *Rumpf (body)*.
- ❖ Der Klassenrumpf besteht aus einer Menge von Deklarationsanweisungen.
  - In Java gibt es verschiedene Typen von *Anweisungen*:  
Deklarationsanweisungen, Bedingte Anweisungen, Zuweisungen, Return-Anweisungen, Schleifenanweisungen.
- ❖ Eine *Methodendeklaration (method declaration)* ist ein Bereich in einer Klassendefinition. Die Methode kann anhand ihres Namens aufgerufen werden.
- ❖ *Mehrzeilen-* and *Einzelzeilen-Kommentare* kann man benutzen, um Java Programme zu dokumentieren.
- ❖ Es gibt verschiedene Arten von Java-Programmen:  
In Info I/II machen wir Java-Applikationen, Java Applets usw.

## *Ein weiteres Beispiel für Klassenentwurf*

- ❖ Problemstellung
- ❖ Analyse und Systementwurf
- ❖ Klassenentwurf: Student
  - Wahl von Datenstrukturen und Datentypen für die Variablen und Algorithmen für die Methoden
- ❖ Klassenspezifikation
  - Dient als Grundlage für die Implementation
- ❖ Die Klassenspezifikation wird dann an die Programmierer geschickt
- ⇒ Implementation in Java

# *Problemstellung*

- ❖ Entwerfe und implementiere ein Programm, das das typische Verhalten von Info I Studenten simuliert.
  
- ❖ Meinung von Domänenexperten:  
*“Ein Student führt zwei Tätigkeiten durch:”*
  - schlafen
  - studieren

# Analyse

- ❖ Welche *Klassen* brauchen wir, und welche *Dienste* sollen sie verrichten?
- ❖ Die Klasse Student
  - Repräsentiert einen Studenten
  - Dienste: `studiere()` and `schlafe()`.
- ❖ Die Klasse Studentenverwaltung 
  - Repräsentiert die Menge aller Studenten in Info I
  - Dienste: Kreiert Studenten und führt Zustandsänderungen an Studenten aus.

# *Klassentwurf: Student*

- ❖ Zustand: 2 Variablen vom Typ boolean:  
studiert und schlaeft
  
- ❖ 3 Methoden:
  - Eine Methode zur Initialisierung von Studenten
  - Eine Method studiere(), um Studenten in den Zustand *studiert* zu bringen
  - Eine Methode schlafe(), um Studenten in den Zustand *schlaeft* zu bringen

# *Klassenspezifikation Student*

## ❖ **Klassenname: Student**

- Aufgabe: Repräsentation typischer Aktivitäten von Studenten

## ❖ **Variablen:**

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

## ❖ **Methoden:**

- Student(): Eine Methode, um ein Objekt vom Typ Student zu initialisieren. (Kriegen wir “umsonst” in Java: Konstruktor)
- schlafe(): Eine Methode, um ein Objekt vom Typ Student in den Schlafzustand zu bringen.
- studiere(): Eine Methode, um ein Objekt vom Typ Student in den Schlafzustand zu bringen.

# *Java Implementation der Klasse Student*

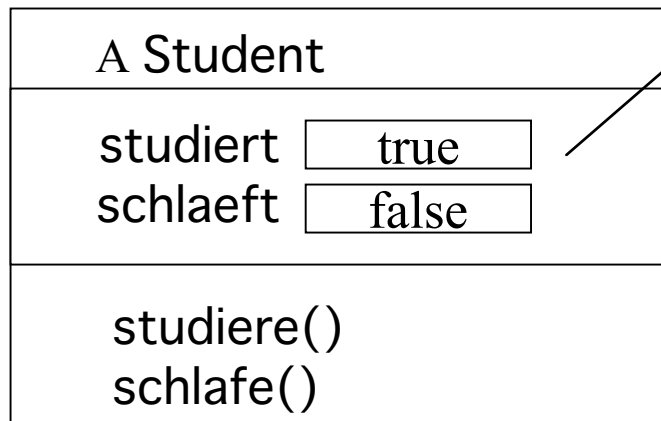
```
public class Student
{
    // Daten:
    private boolean studiert = true; // Zustand des Studenten
    private boolean schlaeft = false;

    Student() {} // Konstruktor
    // Methodendeklarationen:
    public void studiere() // Start von studiere()
    {
        studiert = true; // Ändere den Zustand
        schlaeft = false;
        System.out.println("Student studiert");
        return;
    } // Ende von studiere()

    public void schlafe() // Start von schlafe()
    {
        schlaeft = true; // Ändere den Zustand
        studiert = false;
        System.out.println("Student schläft");
        return;
    } // Ende von schlafe()
} // Ende der Klasse Student
```

# Die Klasse Student

- ❖ Eine Klasse ist eine Schablone für Objekte. In unserem Fall ist jeder Student im initialen Zustand **studiert**.
- ❖ Jedes Objekt einer Klasse besitzt für jedes Attribut der Klasse eine objektlokale Variable, die in Java auch als Instanzvariable bezeichnet wird.



Die Instanzvariablen studiert and schlaeft haben Anfangswerte.

# Aufbau des Klassenkopfes (Class Header)

❖ Beispiel:

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

❖ Allgemeiner Aufbau eines Klassenkopfes:

*KlassenModifikator<sub>opt</sub>* **class** *KlassenName* *Superklasse<sub>opt</sub>*

public

class

Student

ex

Objekt-orientierte  
Programmierung  
(Januar)

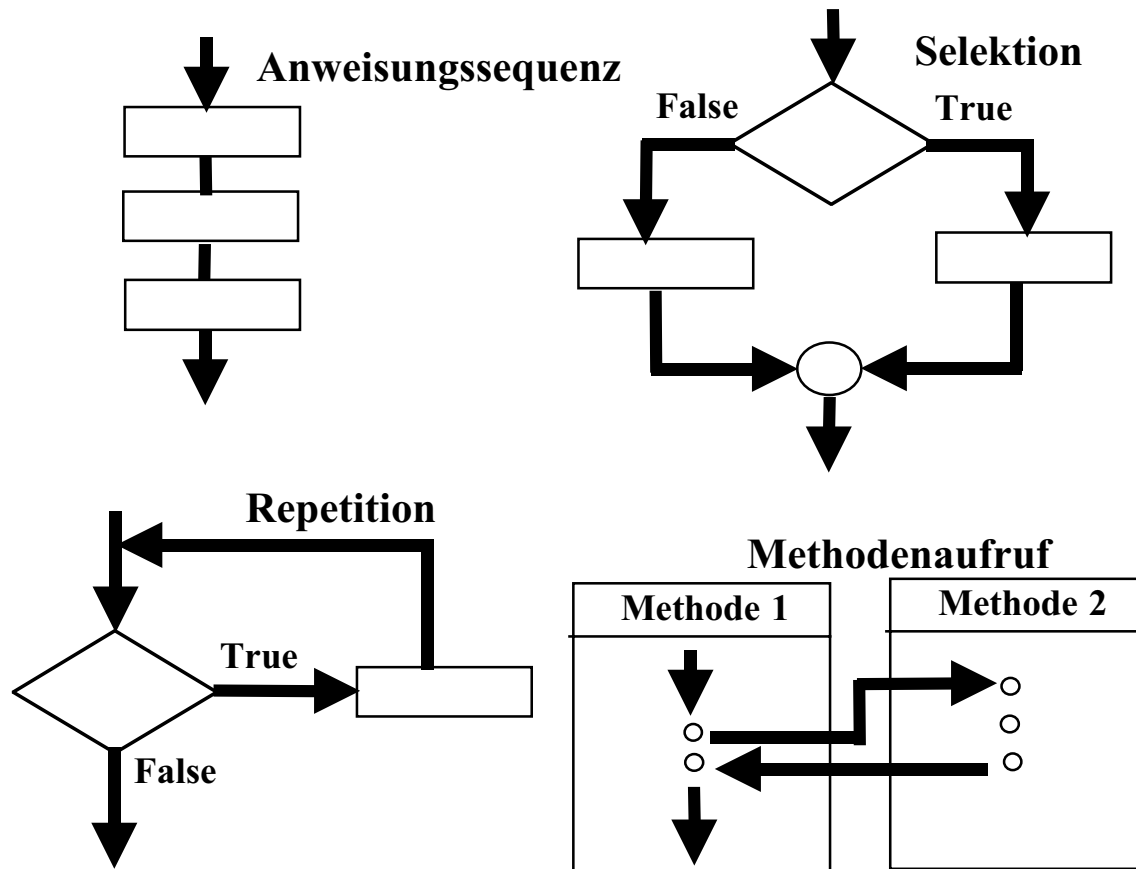
# *Imperative Programmierung*

- ❖ Zustände werden durch die Ausführung von Anweisungen geändert.
- ❖ Die meisten heutigen objektorientierten 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++
  - 1990-2000: Java
- ❖ In diesem Vorlesungsblock beschreiben wir Java's imperative Eigenschaften (*Anweisungen, Kontrollstrukturen*).  
Java's objektorientierte Eigenschaften (*Vererbung, Polymorphismus*) werden wir ab Januar behandeln.

# *Strukturierte Programmierung*

- ❖ **Definition Strukturierte Programmierung:** Imperative Programme, die nur mit bestimmten Kontrollstrukturen (Typen von Anweisungen) geschrieben werden. Diese sind:
  - **Anweisungssequenz** --- Eine Folge von Anweisungen, die eine nach der anderen sequentiell exekutiert werden.
  - **Selektion** --- Eine Anweisung, die eine Wahl zwischen zwei oder mehr Alternativen von Anweisungssequenzen erlaubt (Bedingte Anweisung (**if, if-else**), Fallunterscheidung (**switch**)).
  - **Repetition** --- 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 transferiert. Wenn diese Methode ausgeführt worden ist, wird die Ausführung an der Stelle unmittelbar nach dem Methodenaufruf fortgesetzt.

# 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).

## *Was machen wir jetzt?*

- ❖ Zunächst definieren wir jetzt den Begriff der Variablen genauer.
- ❖ Dann definieren wir den Begriff des Zustandes eines Imperativen Programms (als die Menge der Werte aller Variablen des Programms).
- ❖ Dann schauen wir uns die verschiedenen Anweisungstypen in Java an.
- ❖ Dann schauen wir uns die vier Kontrollstrukturen der strukturierten Programmierung detaillierter an:
  - Beim Konstruieren von Suchalgorithmen
  - Beim Konstruieren von Sortieralgorithmen

# *Der Begriff „Variable“ in der imperativen Programmierung*

- ❖ **Definition Variable:** Eine Variable ist ein Tupel (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;
```

# *Zuweisungsanweisung*

- ❖ Die grundlegende Operation auf einer Variablen in der imperativen Programmierung ist die Zuweisung.
- ❖ Beispiel einer Zuweisung:
  - $x := 5+7$
  - Die Zuweisung berechnet den Wert auf der rechten Seite (12) und ersetzt den Wert der Variablen  $x$  auf der linken Seite durch diesen Wert (12).

# *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.  $\text{false}/x$ ) 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.
  - In der imperativen Programmierung geht das übrigens auch:  $v$  heißt dann eine **Konstante**.

# *Anweisungssequenz: Anweisungen und Verbundanweisungen*

- ❖ Eine Anweisungssequenz besteht aus einer Menge von Anweisungen (statements). Die Reihenfolge, in der diese Anweisungen exekutiert werden, bestimmt den Kontrollfluss im Programm.
- ❖ In einer Anweisungssequenz werden Anweisungen von anderen Anweisungen durch Separatoren getrennt.  
Java kennt folgende Separatoren: ";", "{", und "}".
  - Ein **Block**, auch **Verbundanweisung** (compound statement) genannt, ist eine Folge von Anweisungen, die durch Klammern "{" und "}" eingegrenzt sind.
- ❖ Beispiele von Anweisungssequenzen:
  - `i = i + 1; i = 1; j = 3;`
  - `rectangle.calculateArea();`
  - `{2 = 1 + 1; 5 = 3 + 2; k = 2 + 5}`

# *Anweisungstypen in Java*

- ❖ In Java gibt es verschiedenen Typen von Anweisungen:
  - **Deklarationsanweisung (declaration statement)**
  - **Zuweisung (assignment)**
  - **Bedingte Anweisung (conditional statement)**
  - **return-Anweisung (return statement)**
  - **Schleifenanweisungen (iteration statements)**
- ❖ **Deklarationsanweisung:** Deklariert eine Variable, eine Methode oder eine Klasse.

- `int i;`
- `Rectangle rec;`
- `Student stud1; Student stud2;`
- `Student stud1, stud2;`
- `public void methodenName() {};`
- `public class Student {};`

Bezeichner



## ***Deklarationen: Java-Bezeichner***

- ❖ Ein Java-Bezeichner (*identifizier*) 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!

# *Qualifizierte Namen*

- ❖ Ein *Qualifizierter Name* ist von der Form  
*Referenz.elementName*

wobei *Referenz* sich auf eine Klasse bezieht und *elementName* der Name eines der Merkmale der Klasse ist.

Merkmal = Attribut oder Operation (siehe Vorlesung 2)

- ❖ Wofür brauchen wir Qualifizierung von Namen?
- ❖ Variablen und Methoden in verschiedenen Klassen können denselben Bezeichner haben. Um diese unterschiedlichen Variablen zu identifizieren, benutzen wir qualifizierte Namen.

```
stud1.studiere(); // Aufruf der Methode studiere() des Objekts stud1
```

# *Instanzvariablen vs Lokale Variablen*

- ❖ Innerhalb einer Klasse unterscheidet Java
  - **Instanzvariablen** (instance variables), die auf Klassenebene als Attribute deklariert werden und in Objekten instantiiert werden.
  - **Lokale Variablen** (local variables), die innerhalb von Methoden oder allgemeiner innerhalb einer Verbundanweisung deklariert werden können.
- ❖ Eine Instanzvariable kann einen Modifizierer haben (private, public), eine lokale Variable nicht.

# Deklaration von Instanzvariablen

## ❖ Beispiel:

```
// Instanzvariablen
private boolean studiert = true;
private boolean schlaeft = false;
```

## ❖ Allgemeiner:

*TypId Bezeichner Initialisierer<sub>opt</sub>*

## ❖ 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.

## ❖ Gültigkeitsbereich von lokalen Variablen:

ab ihrer Deklaration bis zum Ende des Blocks, der sie umgibt

## *Zugriffskontrolle: public vs private*

- ❖ Instanzvariablen sind bei uns (InfoI) immer als *private* deklariert.
  - Dadurch sind sie für andere Objekte nicht direkt zugreifbar.
- ❖ Methoden, die Zugriff auf *private* Variablen erlauben sollen, werden als *public* deklariert.
- ❖ Die Menge der öffentlichen (*public*) Methoden definieren die *Schnittstelle (interface)* der Klasse, nämlich die Methoden, auf die von anderen Objekten/Klassen zugegriffen werden kann.

## *Warum sollen Instanzvariablen “private” sein?*

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

```
public boolean studiert = true;  
public boolean schlaeft = false;
```

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

- ❖ Die richtige Art, Georg studieren zu lassen, ist, die zugehörige Zugriffsmethode aufzurufen:

```
georg.studiere(); // studiert() ist public
```

# *Deklaration von Methoden*

## ❖ Allgemeiner Aufbau des Methodenkopfes

*Modifizierer<sub>opt</sub> ResultatTyp MethodenName ( FormaleParameterListe )*

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

## ❖ Definition einer öffentlichen Methode ohne Resultatergebnis

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

# Methodendeklaration

```
public void studiere()  
{  
    studiert = true;  
    schlaeft = false;  
    System.out.println("Student studiert");  
    return;  
} // studiere()
```

Kopf: Diese Methode namens *studiere* ist von anderen Objekten zugreifbar (*public*) und liefert kein Ergebnis zurück (*void*).

Rumpf: Ein Block von 4 Anweisungen, die den Studentenstatus auf *studiert* setzen.

# *Anweisungstypen in Java*

❖ In Java gibt es verschiedenen Typen von Anweisungen:

✓ **Deklarationsanweisung (declaration statement)**

⇒ **Zuweisung (assignment)**

– **Bedingte Anweisung (conditional statement)**

– **return-Anweisung (return statement)**

– **Schleifenanweisungen (iteration statements)**

# *Initialisierung von Instanzvariablen ist eine Zuweisung*

- ❖ Allgemeine Form: *Variable = Ausdruck*
- ❖ Der *Ausdruck* auf der rechten Seite des Zuweisungsoperators (*assignment operator*) wird evaluiert und anschließend der Variablen auf der linken Seite zugewiesen.
- ❖ Beispiel:

```
private boolean studiert = true;  
private int semester = 10;  
private int alter = true;    // Typfehler!
```

- ❖ Typfehler (Type error):  
Man kann einer Variable von Typ **int** nicht einen Wert von Typ **boolean** zuweisen.

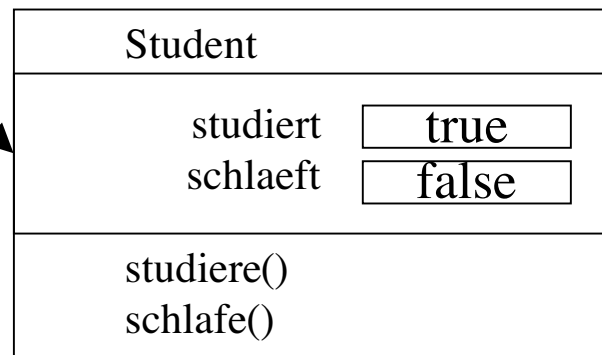
## *Auch eine Zuweisung: Erzeugung einer Student-Instanz*

- ❖ Eine *Variable* kann nicht nur Werte wie **false** oder **100** speichern, sondern auch einen Verweis (“Referenz”, “Adresse”) auf ein Objekt.

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

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

**stud1**



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

**stud2**

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


# Auch eine Zuweisung: Erzeugung einer Student-Instanz

- ❖ Eine *Variable* kann nicht  
sondern auch einen Verw

**Student stud1, stud2**

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

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

stud1 

**new Student()** ist ein Aufruf  
des Konstruktors **Student()**  
der Klasse **Student**

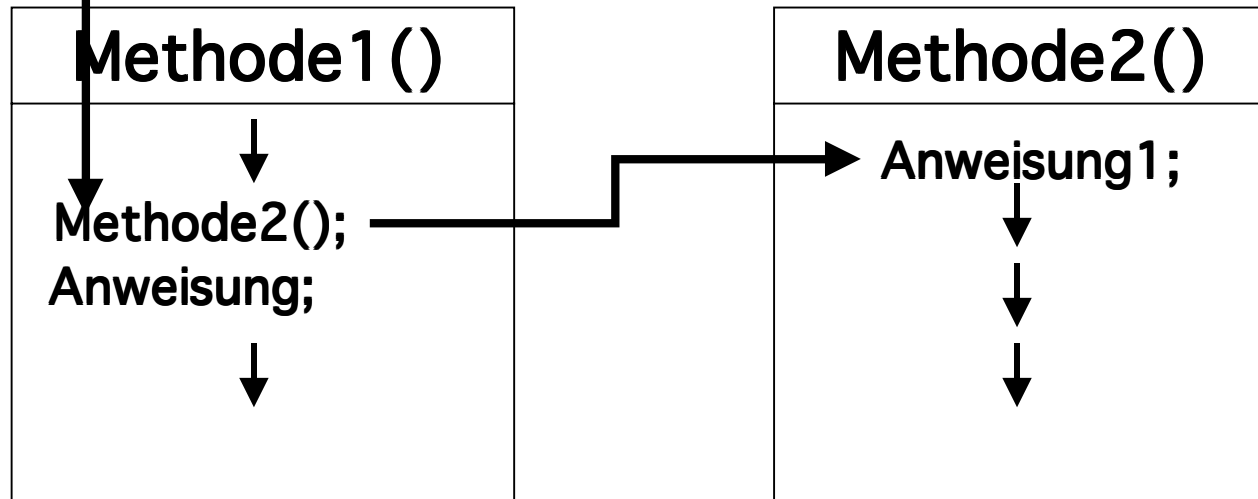
studiere()  
schlafe()

stud2 

Die andere Variable **stud2** kann irgendwann  
mal auf ein **student**-Objekt verweisen, aber  
ihre Referenz ist zur Zeit nicht definiert (**null**).

# Methodenaufruf

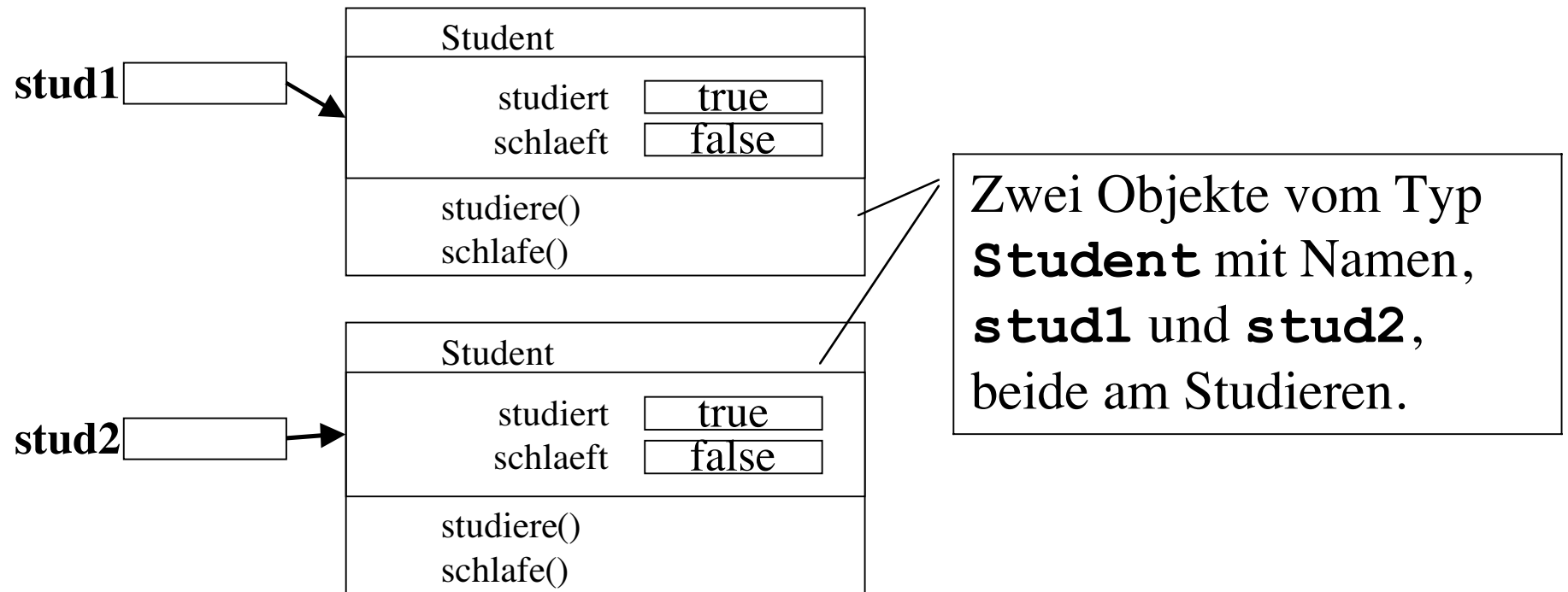
- ❖ Ein Methodenaufruf (*method call*) ist auch eine Anweisung. Der Aufruf verursacht einen Transfer der Kontrolle innerhalb eines Programms zur ersten Anweisung in der aufgerufenen Methode.



## Erzeugung einer *Student-Instanz* (cont'd)

- ❖ Jetzt rufen wir auch für stud2 die Konstruktormethode auf:

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



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

## *Zuweisungsoperator und Gleichheitsoperator in Java*

❖ In Java wird für den Zuweisungsoperator ':=' das Gleichheitszeichen '=' verwendet. Die Gleichheitsüberprüfung wird durch '==' ausgedrückt.

❖ Beispiele:

- `string status; /* definiert eine Variable status`
- `status = "bachelor"; /* status wird der Wert  
"bachelor" zugewiesen.`
- `if (status == "bachelor") {return true};`

❖ **Wichtig:** Ein oft gemachter Fehler ist die Benutzung des Java-Zuweisungsoperators "=" anstelle des Java-Gleichheitsoperators "==".

- Beispiel: Die Anweisungen

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

sind syntaktisch korrekt, aber semantisch falsch.

Semantische Fehler wie diese können von Compilern nicht gefunden werden!

# *Anweisungstypen in Java*

❖ In Java gibt es verschiedenen Typen von Anweisungen:

√ **Deklarationsanweisung (declaration statement)**

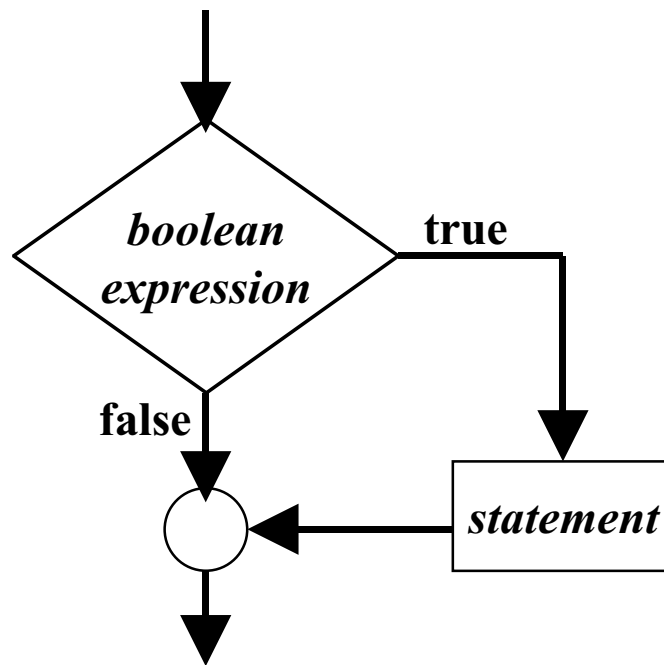
√ **Zuweisung (assignment)**

⇒ **Bedingte Anweisung (conditional statement)**

– **return-Anweisung (return statement)**

– **Schleifenanweisungen (iteration statements)**

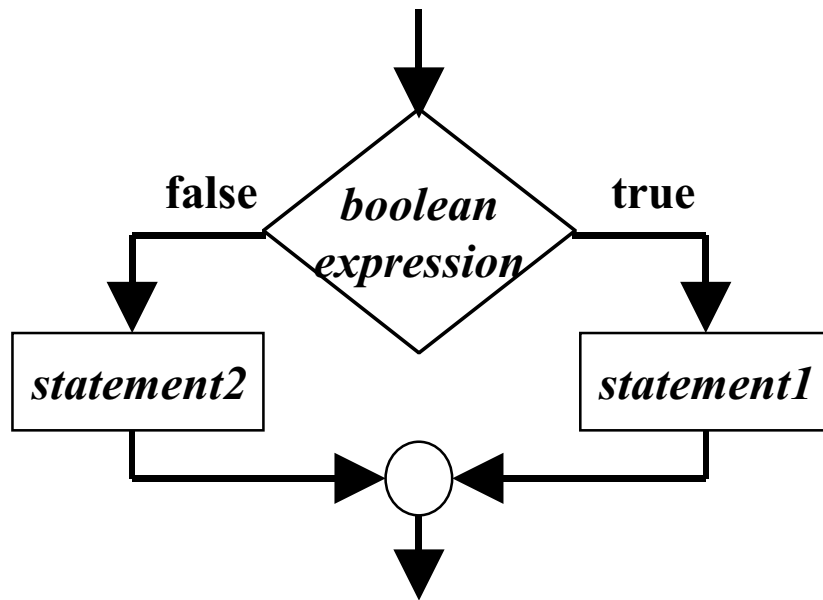
## *Bedingte Anweisungen: Die if-Anweisung*



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

- ❖ Wenn der boolesche Ausdruck “*boolean expression*“ wahr ist, dann wird “*statement*“ exekutiert. 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, exekutiere “*statement1*“, sonst exekutiere “*statement2*“.

# *Bedingte Ausdrücke und bedingte Anweisungen*

- ❖ Bedingte Ausdrücke hatten wir schon in der funktionalen Programmierung kennengelernt:

- return boolean\_expression ? Ausdruck1: Ausdruck2

- Beispiel:

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

- ❖ Bedingte Ausdrücke können wir mit einer bedingten Anweisung schreiben:

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

# *Beispiele von bedingten Anweisungen*

Einfaches if

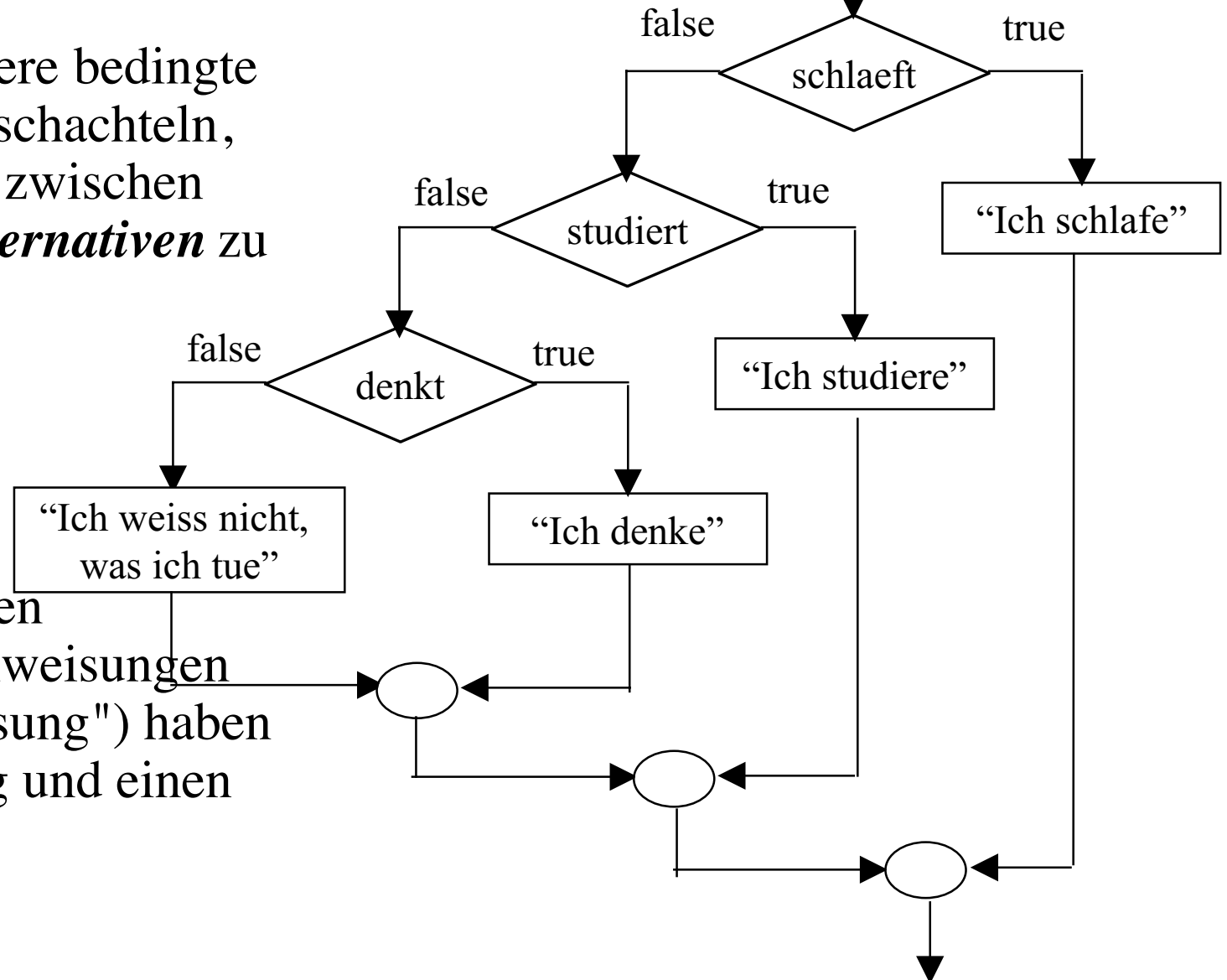
```
if (studiert)  
    return "studiert";
```

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");
```

## *print- und println-Aufrufe*

- ❖ Die Methode **System.out.print** druckt eine Zeile Text.  
Der Text kann aus mehreren Zeichenketten zusammengesetzt sein.
- ❖ Von Zeichenketten in Gänsefüßchen `""` (double quotes) wird alles zwischen den `""` gedruckt:
- ❖ Beispiel:
  - `System.out.print("Eine Zeichenkette");`
- ❖ Ausgabe:
  - **Eine Zeichenkette**
- ❖ Der Konkatenationsoperator für Zeichenketten ist `+`.
- ❖ Beispiel:
  - `System.out.print("Diesist" + " eine Zeichenkette");`
- ❖ Ausgabe:
  - **Diesist eine Zeichenkette**

## *print- und println-Aufrufe (cont'd)*

- ❖ Werte von Variablen und Ausdrücken werden in Zeichenketten konvertiert, bevor sie ausgedruckt werden.

- ❖ Beispiel:

```
int i; int k;  
i = 56; k = 1;  
System.out.print("Der Wert von i ist " + i);
```

- ❖ Ausgabe:

```
Der Wert von i ist 56
```

- ❖ Beispiel:

```
System.out.print("Die Variable i ist auf " + i +  
"      initialisiert worden");
```

- ❖ Ausgabe:

```
Die Variable i ist auf 56      initialisiert worden
```

## *Unterschied zwischen print und println*

- ❖ `System.out.println` druckt am Ende noch einen zusätzlichen Zeilenvorschub (return).

```
System.out.print("i = " + i);
```

```
System.out.print(", k =" + k);
```

```
i = 56, k =1
```

```
System.out.println("i=" + i + k);
```

```
i=561
```

```
System.out.println("i=" + (i + k));
```

```
i=57
```

- ❖ Das Zeichen `"\n"` bewirkt auch einen Zeilenvorschub.

```
System.out.print("i =" + i + "\n");
```

```
System.out.println(",k =" + k);
```

```
i =56  
,k =1
```

## Das “hängendes-else”-Problem

Was wird hier gedruckt, wenn `condition1 == false` ist?

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

“Falsche” Indentation

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

Korrekte Indentation

- ❖ 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.
- ❖ Indentierungen sollen die Logik Ihres Programms reflektieren, aber....
  - Indentierungen 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");
```

Besser:

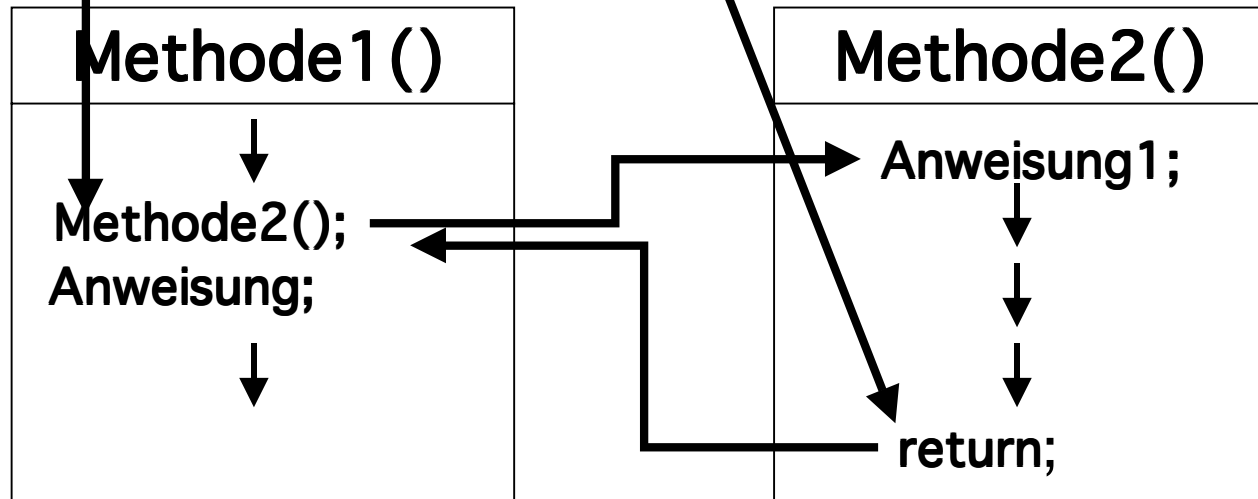
```
if (condition1) {
  if (condition2) {
    System.out.println("Eins");
  } else {
    System.out.println("Zwei");
  }
}
```

# *Anweisungstypen in Java*

- ❖ In Java gibt es verschiedenen Typen von Anweisungen:
  - ✓ **Deklarationsanweisung (declaration statement)**
  - ✓ **Zuweisung (assignment)**
  - ✓ **Bedingte Anweisung (conditional statement)**
  - ⇒ **return-Anweisung (return statement)**
  - **Schleifenanweisungen (iteration statements)**

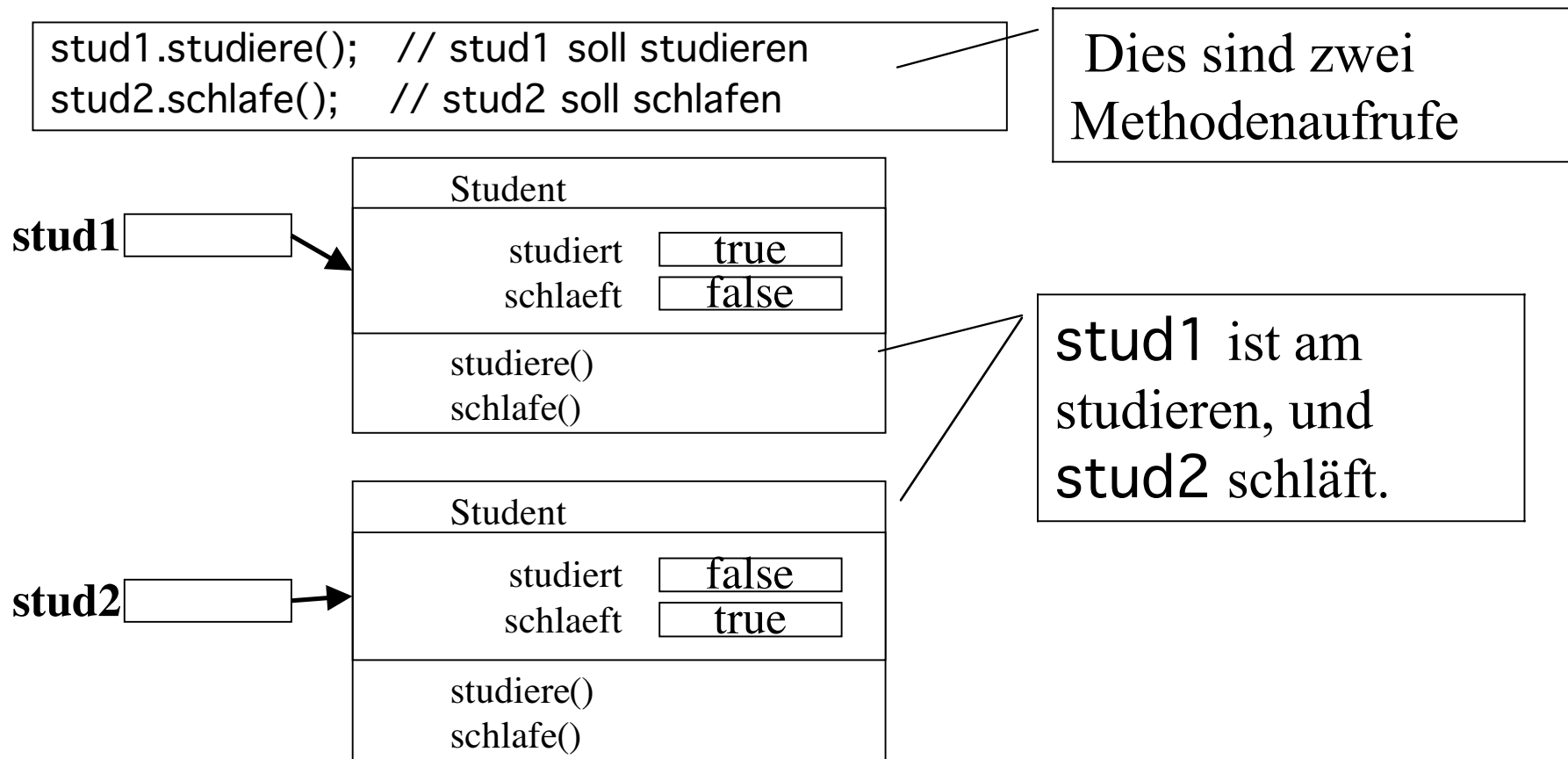
# Methodenaufruf und *return*

- ❖ 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.



# Manipulation von Studenten

- ❖ Der Zustand eines Objektes wird verändert, indem man seine öffentlichen Methoden aufruft:



# *Implementation der Klasse Studentenverwaltung: Deklarationen, Zuweisungen, Methodenaufruf und return-Anweisungen*

```
public class Studentenverwaltung
{
    public static void main (String[] argv)
    {
        // Exekution startet hier
        System.out.println("main() is starting");
        Student stud1;           // Deklaration von 2 Variablen
        Student stud2;
        stud1 = new Student();   // Der Variable stud1 wird ein Wert zugewiesen
        stud2 = new Student();   // Der Variable stud2 wird ein Wert zugewiesen
        stud1.schlafe();         // stud1 soll schlafen.
        stud1.studiere();        // stud1 soll studieren.
        stud2.schlafe();         // stud2 soll schlafen.
        System.out.println(" Ich habe fertig");
        return;                  // Zurück zur Umgebung (Java-Interpreter)
    } // main()
} //Studentenverwaltungssystem
```

# *Implementation der Klasse Student*

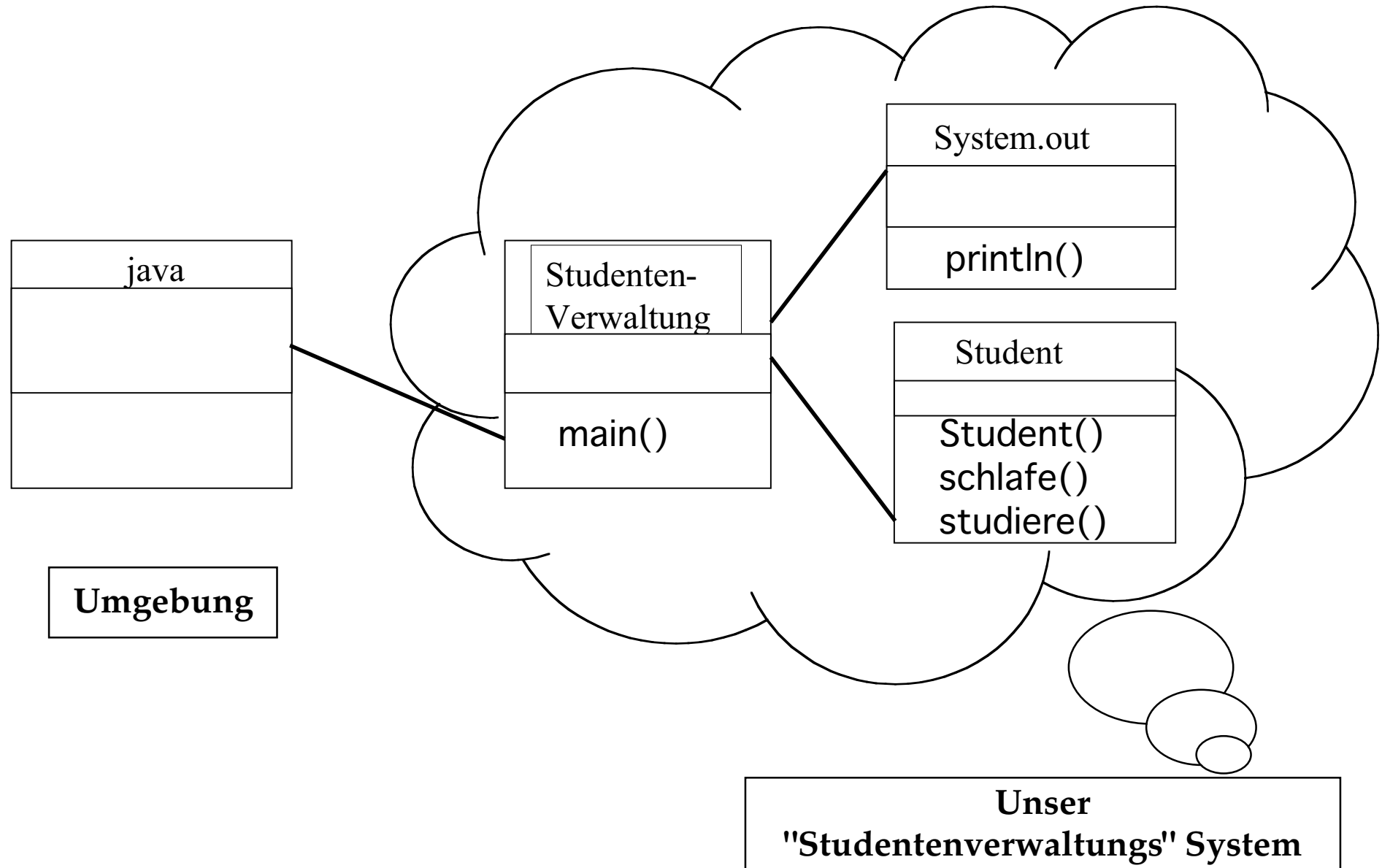
```
public class Student
{
    private boolean studiert = true;    // Zustand der Klasse
    private boolean schlaeft = false;

    // Methoden

    public void studiere()             // Start von studiere
    {
        studiert = true;                // Ändere den Zustand
        schlaeft = false;
        System.out.println("Student studiert");
        return;
    } // studiere()

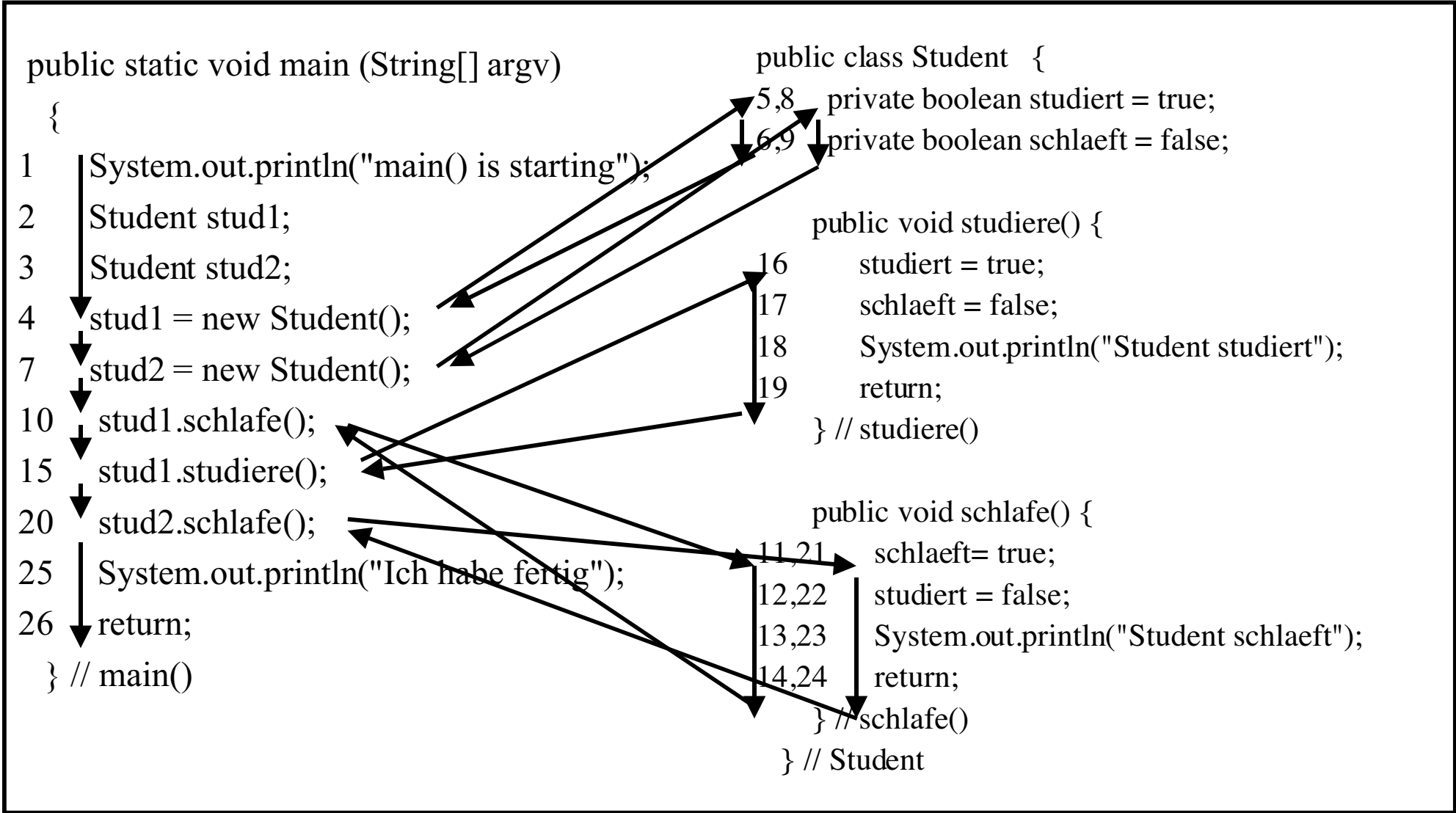
    public void schlafe()              // Start schlafe
    {
        schlaeft = true;                // Ändere den Zustand
        studiert = false;
        System.out.println("Student schläft");
        return;
    } // schlafe()
} // Student Klasse
```

# *Klassendiagramm für das Studentenverwaltungssystem*



# Trace vom Studentenverwaltungssystem

SVS 0.1



# *Konstruktoren*

- ❖ Jede Java-Klasse hat mindestens eine Operation, die denselben Namen hat wie die Klasse. Diese Operation heißt **Konstruktor**.
- ❖ Der Zweck eines Konstruktors ist es, die notwendigen Initialisierungen bei der Instantiierung einer Klasse durchzuführen.
  - Wenn man keinen Konstruktor angibt, dann gibt uns Java den sogenannten Default-Konstruktor, der keine Argumente hat und keine spezielle Initialisierung durchführt.
- ❖ Der Konstruktor kann auch explizit deklariert werden.
  - Info I Programmierregel:  
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 Objekt-/Klassenbezeichner).

# *Verbesserung der Implementation von Student: Explizit definierter Konstruktor*

```
public class Student
{
    private boolean studiert;
    private boolean schlaeft;

    public Student() { // Zustand der Klasse wird im Konstruktor gesetzt
        studiert = true;
        schlaeft = false;
    }

    public void studiere() { // Start von studiere
        studiert = true; // Ändere den Zustand
        schlaeft = false;
        System.out.println("Student studiert");
        return;
    } // studiere()

    public void schlafe() { // Ändere den Zustand
        schlaeft = true;
        studiert = false;
        System.out.println("Student schläft");
        return;
    } // schlafe()
} // Student Klasse
```

Beispiel