10. Der grundlegende Umgang

mit Klassen

Es wird Kapitel 10 von

Ratz / Scheffler / Seese

besprochen.

Wir haben bisher folgenden Programmaufbau:

global sichtbar - wenn nicht verdeckt.

```
public class <Name> {
  static int a=1, b=2, c=3; // Klassen-
                            // variablen
  public static class <KName> { // innere
                                // Klasse
    public int Alter;
    public String Name; } // Komponenten-
                          // variablen, Instanzvariablen

  public static int <MName>(<Par>) {
                          // Methodendeklaration
    int x
    :
  }

  public static void main(String[] args){
    :
  }
```

Auslagern von Klassen zwecks
mehrfacher Benutzung:
Das static Feld.

```
public class <KName> ? // Top-level;
                       // Eluent-Klasse
                    ?
```

in Datei <KName>.java.

Beim Übersetzen

<KName>.class

Bei inneren Klassen

<Name> $ <KName>.class

<Name>: Umgebende Top-level-Klasse

<KName>: Name der inneren Klasse.

Die Idee der Objektorientierung, gebräuchlich in der modernen Softwareentwicklung basiert auf dem Konzept der Klasse. Einführung am Beispiel des Begriffes Student. Eine Top-level Klasse bei seiner Modellierung:

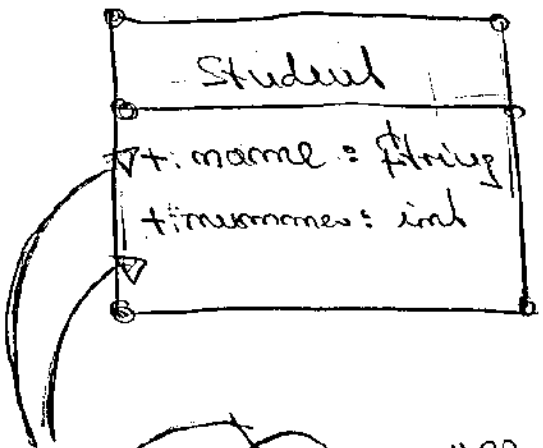
```
public class Student {
    // Stellen Studenten dar.

    public String name;

    public int nummer;
}
```

Klassendiagramm in UML

(unified modelling language)



wegen (public), öffentliche Zugriffsrechte

Instantiierung (d.h. Objekte aus Klasse)

```
Student studi = new Student( );

studi.name = ... ;

studi.matrikelnr = ... ;
```

10.6

Strukturierung durch Kapselung

Bisher Änderung an Klasse Student (etwa zu public String name, zwecks Speicherung) erfordert Änderungen an allen Punkten, wo Student vorkommt!

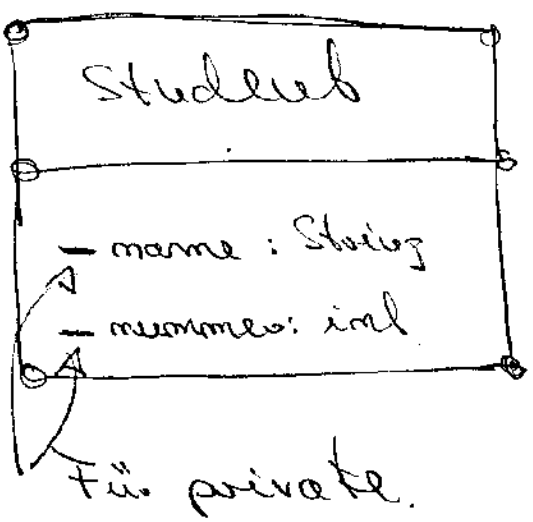
Zugriffsmethoden helfen hier ab.

```
public class Student {
    private String name;
    private int nummer;
}
```

public = jede Klasse kann auf die Variablen zugreifen.

private = keine Klasse kann zugreifen, (außer die Klasse Student selbst).

UML Diagramm:



Zum Zugriff durch andere Klassen:

static fehlt — Instanzmethoden.

public <Rückgabetyp> <MName>(<Pos>) {

// gewünschtes Programm

}

Bisher waren unsere Methodendeklarationen

public static <RTyp> <OMName>(<Pos>) {

// Programm

}

Instanzmethode (ohne static) ist
an spezielles Objekt gebunden, das
existiert ohne ein konkretes Exemplar.
Hat Zugriff auf Instanzvariablen
des zugehörigen Exemplars.

10.3

Erweitern wir die Klasse Student um Zugriffsmethoden, so sieht die Klassendeklaration so aus:

```
public class Student {

    private String name;
    private int nummer;

    public String getName( ) {
        return this.name; }

            // "this" ist Referenz auf
            // Objekt selbst, aus dem
            // aufgerufen wird. "this"
            // ist implizite Komponenten-
            // variable.
```

10.10

```
public void setName(String name) {
    this.name = name; }
    // aus Objekt.

public int getMatrNr() {
    return matrNr; }
    // aus Klasse.matrNr

public void setMatrNr(int m) {
    matrNr = m; }
```

UML-Diagramm in Abbildung 10.3.

Aufruf dann so:

```
...
matrNr = 123
Student stud = new Student();
stud.setName("Hülle");
stud.setMatrNr(78710);
int Mat.Nr = stud.getMatrNr();
```

10.11

Beste

int Matr Nr = studi. nummer

bzgl. Fehlermeldung (sofern nicht in der Klasse Student selbst).

Also: Verlassen von static in der Methodendeklaration

⇔ Binden an ein Objekt.

Die Testausmethode

```
public boolean valNummer {

    return

        nummer >= 10000

            && nummer <= 99999

            && nummer %2 != 0; }
```

überprüft die Gültigkeit einer Matrikelnummer. Darauf muss

setNummer:

```
public void setNummer(int n) {

    int alteNummer = nummer;

    nummer = n;          // war ist neue
                         // nummer noch nicht
    if (!valNummer())    // initialisiert &
        nummer = alteNummer;
}
```

Bild nach außen.

Validiert: gleiche Schnittstelle wie

Klasse Student, auch bei neuer (!)

Regeln für die Matrikelnummer.

Erleichterung von Modifikationen.

16.1b

Weitere Methode:

```
public String toString() {
    return name + "(" + nummer
        + ")"; }
```

Aufruf dann so:

```
Student studi = new Student();
studi.setName("k.k");
studi.setNummer(12345);

System.out.println(studi.toString());
```

Ergibt k.k.(12345) als Ausgabe.

Auch möglich wäre:

System.out.println(studi);

für studi.toString

da toString bei jeder Instanz

als Methode dabei ist. Wir haben

das alte toString überschrieben.

Unsere früheren Methoden gewiß

public static <Rückgabetyp> <Name>(<Par>){

// Programm }

sind Klassenmethoden oder

statische Methoden. Auch ohne

Instanz existiert. Das

Programm 12ITest.java

zeigt noch einmal, daß unsere

bisher geschriebenen Programme

einfach Top-level Klassen sind.

Es können Instanzen generiert

und ausgegeben werden. Auch dem Objekt!

Zum Nutzen der als statisch

definierten Methoden.

Wollen alle existierenden Studenten

zählen. Variable zählen einmal

für alle Instanzen der Klasse;

gemäß:

Kein Zugriff von außen möglich!

Für alle Instanzen.

```
private static int zaehler = 5;
```

Hatten früher bereits:

```
public static int ...
static int ...
```

am Beginn unserer Klasse.

Zugriff von außen durch:

Klassenmethode, ohne Objekt aufrufbar.

```
public static int getZaehler() {
    return zaehler }
```

Aufruf

Die Klasse! Kein Objekt.

```
System.out.println(Student.getZaehler())
```

Um Zaehler aktuell zu halten:

Rückgabetyp.

```
public static Student createSt() {
    Zaehler++;      // sind in der Klasse
                    // Student.
    return new Student(); }
```

Aufruf

```
Student
  studi = Student.createSt();
System.out.println
      (Student.getZaehler)
```

Haben Student in Student. Vergleiche

```
public class Knoten {
  public int Wert;
  public Knoten ref; }
```

Problem: Was ist wenn auch irgendwo noch new Student() nicht innerhalb createSt() ?

Abbildung 10.5 ist jetzt die aktuelle Klasse Student.

Weitere statische Komponenten einer Klasse sind Konstanten.

Als Beispiel das Studienfach: gemäß Tabelle 10.1.

Erweiterung der Klasse Student:

```
private int fach;

public int getFach() {
    return fach; }
```

Zugriffsmethode.

```
public void setFach(int fach) {
    this.fach = fach }
```

(10.19)

2 Möglichkeiten: Klassenmethode

```
Student $studi = Student.createSt();
studi.setFach(3)
```

Zustandsmethode

- Wenig schreibgeschützt.
- Fehleranfällig.

Abhilfe mit Konstanten in der Klassendeklaration gemäß:

```
private final static int
        MAXID = 1;
```

Haben auch innerhalb einer Methode die Möglichkeit der Konstantendeklaration:

```
final int MAX = 5;
```

(siehe S. 67)

Beispiel gemäß S. 265. Aufruf

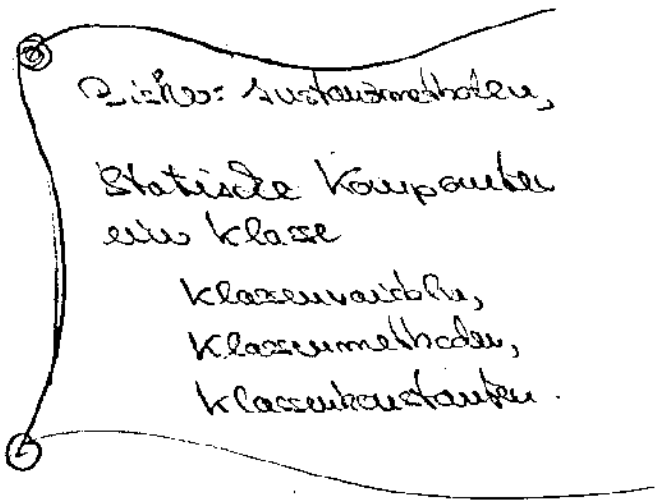
also

Student.Informatikstudium;

Dagegen gibt

Student.Zulstudium = 2;

Fehlermeldung:



Wir kommen zur Instantiierung.

= Erzeugung von Objekten. Siehe

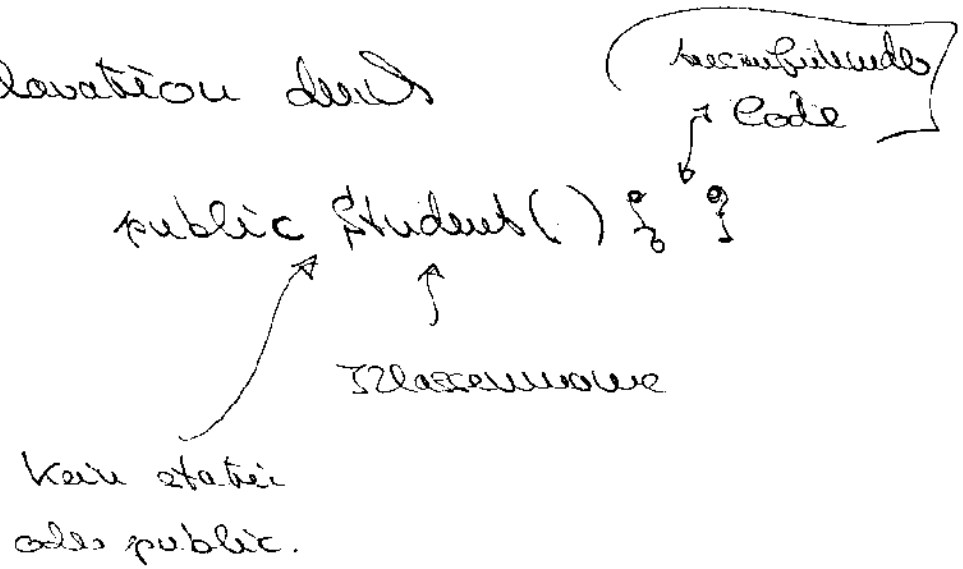
- new Student(), new PKW()
- createPkt().

Klassenmethode.

Konstruktoren erlauben es, den Erzeugungsprozess eines Objekts zu steuern.

Bisher nur `Student()`.

Deklaration durch



Standardkonstruktor, Defaultkonstruktor.

Aufruf wie gehabt

```
Student studi = new Student();
```

Ein anderer Konstruktor:

```
public Student() {
    zaehler++
}
```

Dann

```
public static Student create Student() {
    return new Student ();
}
```

↑

Erhöhung von
Zähler implizit
dabei.

Überladen von Konstruktoren,

Seite S. 269, 270, 271.

10.23

Zur Objekterzeugung.

Zu der Klassenelevation

private String name = "Dirk".

Aber im Konstruktor

this.name = "Namenlos".

Wenn ich das Ergebnis zwei

Student stud = Student.createStudent();

10.24

Was geschieht genau beim Aufruf eines Konstruktors ?

① Speicherplatz für Instanzvariablen.

② Standardwerte auf Instanzvariablen. (Tabelle 10.2, Seite)

③ Konstruktor wird mit seinen Parametern ausgeführt.

ⓐ 1. Zeile this ( ), dann letzten ausführen. Dann Rest des Konstruktors.

ⓑ 1. Zeile nicht this, dann erst alle Initialisierungen der Klassendeklaration, dann Rest des Konstruktors.

wird nach 2. ausgeführt wenn nicht 1. Zeile this (Überlader)