



Objektorientierte Programmierung

Kapitel 2 – Klassen und Objekte

Prof. Dr. Kai Höfig

Inhalt

- **Klassen und Objekte**
- Grundprinzipien der Objektorientierten Programmierung erster Teil
(zweiter Teil in nachfolgendem Kapitel)
- Sichtbarkeit
- Klassen und Objekte: Weitere Details
 - Überladen von Methoden, Selbstreferenz `this`, `final`, Immutable Klassen
- Statische Methoden und Attribute
- Vernetzung von Objekten
- Innere Klassen

Zum Nachlesen: <http://openbook.rheinwerk-verlag.de/javainsel/>

Kapitel 3: Klassen und Objekte

C. Ullenboom, Java ist auch eine Insel

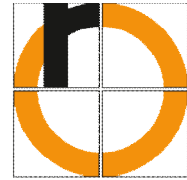
Kapitel 5: Eigene Klassen schreiben (v.a. Kapitel 5.1, 5.2, 5.3, 5.4, 5.6, 5.7)

Wieso Objekte? – Typen in C

```
struct konto {  
    int kontonummer;  
    double saldo;  
};
```

- **Typen in C werden durch `struct` definiert.**
 - Beliebig viele „Objekte“ lassen sich auf diese Weise von einem Typen ableiten
- **Nachteile:**
 - Umliegender Programmcode muss sich um die **Konsistenz** der inneren primitiven Datentypen kümmern. Was passiert wenn z.B. `=` mit `+=` bei der Zuweisung von Saldi eines Kontos verwechselt wird? Wer stellt sicher, dass Methoden auch immer benutzt werden (z.B. `zahleEin(double betrag, konto k)`)?
 - **Redundanter** Programmcode entsteht schnell, wenn ähnliche Typen verwendet werden.

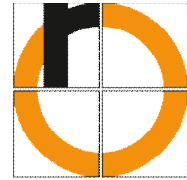
Wieso Objekte? – Klassen in Java, der bessere Typ



```
class Konto {  
    private int kontonummer;  
    private double saldo;  
    public void zahleEin(double betrag) {  
        saldo+=betrag;  
    }  
}
```

- **Typen in Java werden durch `class` definiert.**
 - Zusätzlich zu primitiven Datentypen enthalten Klassen auch Methoden und können vererbt werden.
- Vorteile:
 - **Konsistenz** wird sichergestellt durch ausschließliche Verwendung der Methoden und Einschränkung des Zugriffs auf primitive Typen einer Klasse.
 - **Redundanter** Programmcode kann durch geschickte Vererbung drastisch reduziert werden.

Klassen und Objekte

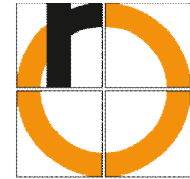


Eine **Klasse** ist eine allgemeingültige Beschreibung von Dingen, die in verschiedenen Ausprägungen vorkommen können, aber alle eine **gemeinsame Struktur** und ein **gemeinsames Verhalten** haben. Sie ist ein **Bauplan** für die Erzeugung von einzelnen konkreten Ausprägungen. Diese Ausprägungen bezeichnet man als **Objekte oder Instanzen** der Klasse.

Quelle: D. Abts, *Grundkurs Java*, 9. Auflage, Springer

Weitere Beispiele aus dem Alltag für Klassen und Objekte?

Wesentliche Elemente eines Objekts



- **Identität**
 - "Gerätenummer": Wie heißt das Objekt? Identifier? Kontonummer?
 - Nötig, da es viele Ausprägungen/Instanzen geben kann.
- **Zustand**
 - Sogenannte Attribute beschreiben den Zustand eines Objekts.
 - Beispiel: Wieviel Geld ist auf dem Konto?
 - Zustand meist **Privatsache** und sollte von außen nicht **direkt** veränderbar sein. (z.B. Kontoslado verändert sich nur durch Überweisung)
 - Jedes Objekt hat eine möglichst kleine Schnittstelle zur Außenwelt. .
- **Verhalten**
 - Jedes Objekt besitzt Funktionen (**Methoden**), die Attributwerte und damit den Zustand verändern.
 - Beispiel Konto: Bei Eröffnung wird eine Kontonummer festgelegt.
- **Vernetzung von Objekten**
 - Objekte müssen mit anderen Objekten interagieren.
 - Objekte rufen Methoden anderer Objekte auf.
 - Z.B. kann von einem Konto Geld an ein anderes Konto überwiesen werden.

Definition einer Klasse in Java



- Jede Klasse ist eine "Objekt-Schablone", Schlüsselwort "**class**".
 - Objekt muss dann separat erzeugt werden, siehe später!
- Jede Klasse stellt einen **Datentyp** dar
 - Ähnlich wie `int`, `boolean`, etc.

Name der Klasse: Entspricht in der Regel dem Dateinamen

```
public class Konto {  
    private int kontonummer;  
    private double saldo;  
  
    public void zahleEin(double betrag) {  
        saldo += betrag;  
    }  
  
    public void hebeAb(double betrag) {  
        saldo -= betrag;  
    }  
  
    public void printInfo() {  
        System.out.println("Kontonummer: "  
            + kontonummer + " Saldo:  
            " + saldo);  
    }  
}
```

Attribute:

"private" bedeutet, dass nur Methoden dieser Klasse auf die Attribute zugreifen können.

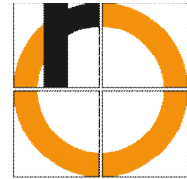
Methoden:

"public" bedeutet, dass diese Methode von außerhalb des Objektes aufgerufen werden kann.

Konstruktor: Die "Hebamme" einer Klasse

- Selbstverständlich müssen in einer Bank-Software **mehrere** Konten (=Objekte) verwaltet werden.
- Konstruktor: Spezielle Methode in der Klassendefinition, die **automatisch** aufgerufen wird, sobald ein Objekt erzeugt wird.
- Konstruktor konfiguriert meist den Anfangszustand eines Objektes.
- Konstruktor hat stets den **gleichen Bezeichner** wie die Klasse.
 - Es können aber Parameter übergeben werden.
- Konstruktor muss nicht zwingend explizit definiert werden.
 - Falls undefiniert: Parameterloser **Default-Konstruktor** ist immer vorhanden.
 - **Wertkonstruktor**: Explizite Angabe durch Programmierer, wobei unterschiedliche Parameterlisten (Signaturen) möglich sind

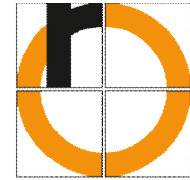
Konstruktor für die Klasse Konto



```
public class Konto {  
    private int kontonummer;           // "-" im UML Diagramm  
    private double saldo;  
  
    public Konto(int kNr, int betrag) { // constructor  
        kontonummer = kNr;  
        saldo = betrag;  
    }  
  
    public void zahleEin(double betrag) {  
        saldo += betrag;  
    }  
  
    public void hebeAb(double betrag) {  
        saldo -= betrag;  
    }  
  
    public void printInfo() {  
        System.out.println("Kontonummer: " + kontonummer + " Saldo: " + saldo);  
    }  
}
```

- Der Konstruktor definiert hier den Anfangszustand
 - Anfangssaldo und Kontonummer

Wie stößt man die Geburt eines Objekts an?



- Vorbereitung: Implementiere ggfs. Konstruktor in Klasse
- Aufruf von `new <Kl assenname>`
 - Häufig im Hauptprogramm: `public static void main(.)`
 - Ggfs. Aufruf mit Parametern.
- Im Hintergrund passiert dann folgendes:
 - Es wird eine Instanz der Klasse angelegt.
 - Gleichbedeutend mit: Bereitstellen des notwendigen Speicherplatzes auf dem Heap.
 - Der **Konstruktor** der Klasse <Klassenname> wird ausgeführt.
- Rückgabe von `new`:
 - Neu erzeugtes Objekt (Referenz)
 - Kann einer Variablen vom korrekten Typ zugewiesen werden.

```
Konto konto1 = new Konto(1234, 5000);
```

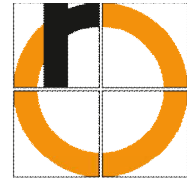
Datentyp
(Klasse ist
Datentyp)

Bezeichner
des Objekts

stößt Aufruf des
Konstruktors der Klasse
Konto an

Parameter für
Konstruktor

Anlegen und Verwenden mehrerer Konten



```
public class KontoTest {  
  
    public static void main(String[] args) {  
  
        // Ein Objekt der Klasse Konto wird erzeugt.  
        Konto konto1 = new Konto(1234, 5000);  
  
        // Erzeuge ein 2. Objekt der Klasse Konto  
        Konto konto2 = new Konto(5678, 1000);  
  
    }  
}
```

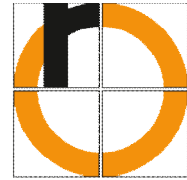
Hier startet der
Programmaufruf

Anlegen eines 2.
Objekts der
Klasse

Konstruktoraufruf

```
public class Konto {  
    private int kontonummer;  
    private double saldo;  
  
    public Konto(int kNr, int betrag) { // constructor  
        kontonummer = kNr;  
        saldo = betrag;  
    }  
  
    public void printInfo() {  
        System.out.println("Kontonummer: " + kontonummer + " Saldo: " + saldo);  
    }  
}
```

Zugriff auf Objektattribute und -methoden

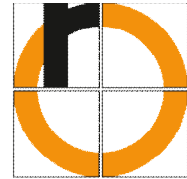


- **Syntax:**
 - Attribut: <Objektbezeichner>. <Attributname>
 - Methode: <Objektbezeichner>. <Methodenname>
- **Info vorab:**
 - Man kann von außen nur auf Attribute und Methoden zugreifen, die als "public" deklariert wurden.

```
public class KontoTest {  
    public static void main(String[] args) {  
  
        // Ein Objekt der Klasse Konto wird erzeugt.  
        Konto konto1 = new Konto(1234, 5000);  
  
        // Erzeuge ein 2. Objekt der Klasse Konto  
        Konto konto2 = new Konto(5678, 1000);  
  
        // Ausgabe des Kontostandes  
        konto1.printInfo();  
        konto2.printInfo();  
    }  
}
```

**Aufruf einer Methode von
außerhalb der Klasse**

Live Programming: Die Klasse Student



- Attribute
- Methoden
- Konstruktoren

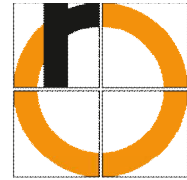
Grundprinzipien der Objektorientierung

Teil 1

- **Abstraktion**
 - Ausschnitt aus der realen Welt
 - Relevante Objekte
 - Relevante, charakteristische Eigenschaften von Objekten.
- **Modularität**
 - Partitionieren in kleinere, weniger komplexe Einheiten
 - Strukturierung durch Objekte, Klassen und Pakete
- **Datenkapselung ("Information Hiding")**
 - Zusammenfassen von Daten und Verhalten.
 - Verbergen der Implementierung hinter einer **Schnittstelle**.
 - Zugriff nur über die Schnittstelle, damit interne Daten konsistent bleiben.

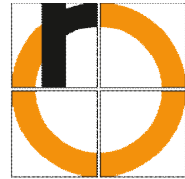
Polymorphie & Vererbung, siehe Teil 2

Sichtbarkeiten von Attributen, Methoden und Klassen



- Die *Sichtbarkeit* kann durch **Modifizierer** eingeschränkt werden
 - **Public** (UML: +)
 - Zugriff von außerhalb der Klasse möglich mit "."-Operator möglich.
 - **private** (UML: -)
 - Kein Zugriff von außerhalb der Klasse.
 - Nur innerhalb von Methoden der gleichen Klasse ist Zugriff möglich.
 - **keine Angabe / paketsichtbar** (UML: ~)
 - Liegt vor, wenn man keine Sichtbarkeit spezifiziert.
 - Allen Klassen des gleichen **Package** (→ später) haben Zugriff
 - **protected** (UML: #)
 - Sichtbar in der eigenen Klasse sowie in allen abgeleiteten Klassen und allen Klassen des Pakets.
 - Siehe Kapitel zur Vererbung.

Prinzip der Datenkapselung

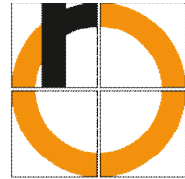


- Zugriff von außen soweit als möglich einschränken!

```
public class Konto {  
  
    private int kontonummer;  
    private double saldo;  
    . . .  
    public void zahleEin(double betrag) {  
        saldo += betrag;  
    }  
}
```

- Wieso?

Zugriff auf private Attribute



```
public class Konto {  
  
    private int kontonummer;  
    private double saldo;  
    . . .  
    public void zahleEin(double betrag) {  
        saldo += betrag;  
    }  
}
```

```
public class Test {  
    public static void main(String[] args) {  
        Konto konto1 = new Konto(1234, 5000);  
        konto1.kontonummer = 0.0;  
    }  
}
```

- Zugriff nicht möglich, da Kontonummer als "privat" deklariert wurde.
- Kontonummer kann nur innerhalb der Klasse "Konto" verändert werden.

Privatsphäre zwischen Objekten der gleichen Klasse

- Funktioniert der Zugriff im folgenden Code?
-
- Ein Objekt der Klasse Konto greift auf Attribut kontoNummer eines anderen Objekts der Klasse Konto zu.

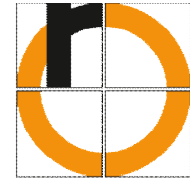
```
public class Konto {  
    private int kontoNummer;  
  
    public Konto( int k ) {  
        kontoNummer = k;  
    }  
  
    public boolean compare( Konto other ) {  
        if (kontoNummer == other.kontoNummer)  
            return true;  
        else  
            return false;  
    }  
}
```

- Zugriff auf other.kontoNummer ist zulässig, da es ein Objekt der **gleichen** Klasse ist!
- Die Klasse greift auf sich selbst zu (nicht aber die Objekte)

Sichtbarkeiten: Sonstiges

- **Privater Konstruktor** ist nur in Ausnahmefällen sinnvoll.
- **Sichtbarkeiten bei Klassen**
 - Werden Klassen als `private` deklariert, sind automatisch alle Attribute und Methoden der Klasse `private`.
 - Eigentlich nur sinnvoll im Zusammenhang mit ***inneren Klassen*** (siehe später)

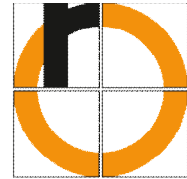
Ändern des Objektzustands: Getter/Setter



- **Problem:**
 - Zustand des Objekts soll soweit wie möglich geschützt sein.
 - Falls dennoch Änderungen am Zustand nötig sind, muss man unbedingt sicherstellen, dass Objektzustand **konsistent** verändert wird.
- **Standardansatz**
 - Attribute soweit als möglich einschränken → `private`!
 - Ändern des Zustandes niemals direkt, sondern mit **Setter-Methode**
 - Sicherstellen der Konsistenz!
 - Auslesen des Zustands mit spezieller **Getter-Methode**.
 - Ggfs. könnte man Ausgabe filtern oder noch bearbeiten.
- **Beispiel für Getter**
- **Beispiel für Setter**

```
public int getKontonummer() {  
    return kontonummer;  
}  
  
public void setKontonummer(int k) {  
    if (k > 0) {  
        kontonummer = k;    //Konsistenz  
    }  
}
```

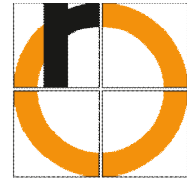
Überladen von Methoden



- **Überladen von Methoden**
 - Klasse hat mehrere Methoden mit **derselben Bezeichnung**.
 - Methoden unterscheiden sich aber in **Signatur**.
 - *Signatur* = Methodenname + Parameterliste
 - Unterscheidung nur im Rückgabewert genügt nicht.
- **Beispiel:** Eine Klasse hat mehrere Konstruktoren oder gleichnamige Methoden.

```
public class Konto {  
  
    private int kontoNummer;  
    private double saldo;  
  
    . . .  
  
    // Default-Konstruktor  
    public Konto() {  
    }  
  
    // Wertkonstruktor 1  
    public Konto(int kNr) {  
        kontoNummer = kNr;  
    }  
  
    // Wertkonstruktor 2  
    public Konto(int kNr, int betrag) {  
        kontoNummer = kNr;  
        saldo = betrag;  
    }  
  
    . . .  
}
```

Was gibt die Methode `voodoo()` aus?



```
public class BlackMagic {  
    int num = 1;  
  
    public void voodoo() {  
        int num = 2;  
        System.out.println(num);  
    }  
}
```

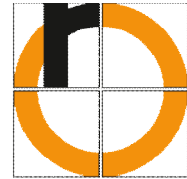
```
public class BlackMagic {  
    int num = 1;  
  
    public void voodoo() {  
        int num = 2;  
        System.out.println(this.num);  
    }  
}
```

- **Ausgabe:** Zahl 2
 - Variable, die innerhalb der Methode definiert wird, überschreibt die Sichtbarkeit des Attributs.
 - Kein Namenskonflikt zwischen Attribut und Variable `num`!
 - Compiler beschwert sich nicht!
- Wie könnte man innerhalb von `voodoo()` auf Attribut `num` zugreifen?
 - Lösung: Selbstreferenz `this`

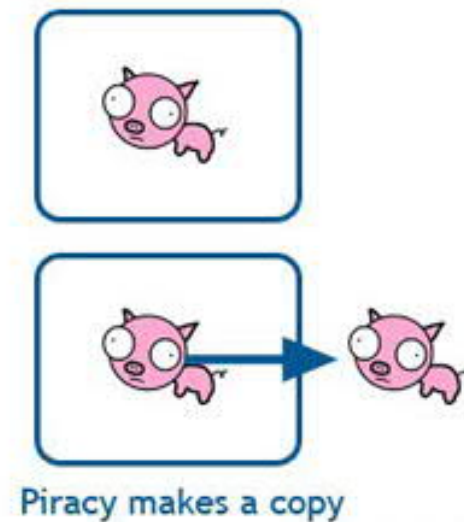
Selbstreferenz `this` und Nullreferenz `null`

- **Referenz:** Ähnlich zu einem Zeiger in C
 - Adresse eines Objekts, die jedoch nicht wie in C manipuliert werden kann.
- **Selbstreferenz:** `this`
 - Zeigt auf das eigene Objekt.
 - Steht in jeder Objektmethode und Konstruktor zur Verfügung.
 - Anwendungen
 - Notwendig wenn Parameter bzw. lokale Variablen Attribute überdecken.
 - Man übergibt einer anderen Methode (eines anderen Objekts) eine Referenz auf sich selbst.
 - Liefert eine Methode als Rückgabe `this`, lassen sich Methoden der Klasse hintereinander setzen.
 - Beispiel: `voodoo.mehrfachVeränderung(a)`
- **Nullreferenz:** `null`
 - Zeigt auf leeres Objekt ("Nullzeiger").
 - Standardwert falls eine Objektvariable nicht initialisiert wird.

Call-by-reference, Call-by-value

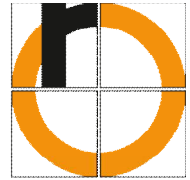


Piracy is not theft. It's piracy. (a handy guide)



Call-by-reference, Call-by-value

Beispiel



```
Class BlackMagic{
    void voodoo3(Konto k){
        k.zahleEin(1);
    }
    void voodoo4(Konto k){
        k = new Konto(200);
    }
    void voodoo5(int i){
        i++;
    }
}
...
Konto k3 = new Konto(0);
BlackMagic.voodoo3(k3);
System.out.println(k3.saldo);

BlackMagic.voodoo4(k3);
System.out.println(k3.saldo);

int zahl=0;
BlackMagic.voodoo5(zahl);
System.out.println(zahl);
```



1

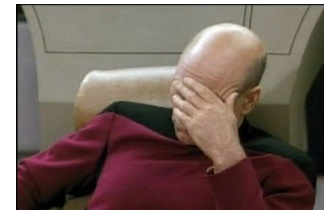


1



0

- *Call-by-reference* bedeutet es wird ein Objektverweis übergeben. (*Theft*)
- *Call-by-value* bedeutet es wird nur der Wert übergeben (*Piracy*)
- In Java immer call-by-value (**where value is a reference**)



Finale Variablen

- **Lokale Variablen, Parameter oder Attribute** können mit dem Schlüsselwort `final` versehen werden.
- **Bedeutung**
 - `final` verbietet spätere Zuweisung bzw. Änderung.
 - Der Wert muss jedoch nicht zum Zeitpunkt der Variablendeklaration zugewiesen werden.
 - Im Programmcode jedoch maximale 1 Zuweisung erlaubt.
 - Bei Attributen muss Zuweisung jedoch spätestens im Konstruktor erfolgen.
- **Beispiel:**

```
final int i = 5;
i++;

final int j;
System.out.println("Ich lasse mir Zeit");
j = 5;
```

Immutable: Unveränderliche Klassen

- **Definition:**
 - Eine Klasse ist **immutable**, falls sich der Zustand eines Objekts nach der Instanziierung nicht mehr ändert.
- **Klassen sollten soweit als möglich immutable sein!!!**
 - Der Zustand ist bereits "bei Geburt" festgelegt. Vereinfacht Anwendung in Datenstrukturen
 - Thread-safe
 - Keine Implementierung der Methode clone notwendig, siehe später.
- **Wie macht man in Java eine Klasse immutable?**
 - Deklariere die Klasse als final
 - Verhindert, dass man von der Klasse ableiten darf.
 - Siehe Kapitel zur Vererbung.
 - Deklariere alle Attribute als private und final
 - Keine Methoden, die Attribute verändern
 - Ausnahme: Konstruktor

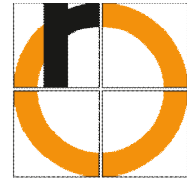
Statische Methoden / Klassenmethoden

- **Statische Methoden**
 - Existieren ***unabhängig*** von einer bestimmten Instanz
 - Können aufgerufen werden, ohne dass zuvor mit new eine Instanz einer Klasse angelegt wird.
- **Deklaration** mit static-Modifizierer
- **Beispiele**
 - Klasse Random: **static double** random() liefert Zufallszahl aus [0..1]
 - Klasse Math: **static double** sqrt(**double** a) berechnet Quadratwurzel.
 - **public static void** main(String[] args)
- Es wäre "Overkill", wenn man zum Berechnen der Quadratwurzel erst ein Objekt der Klasse Math erzeugen müsste.

Statische Variablen / Klassenattribute

- **Statische Variablen**
 - Existieren **unabhängig** von einem Objekt
 - Werden **nur einmal angelegt** in der "Objektschablone" bzw. im "Bauplan"
 - Können von allen Methoden der Klasse verwendet werden, d.h. Methoden **teilen** sich die Variable
 - Lebensdauer erstreckt sich auf das gesamte Programm.

Anwendung für statische Attribute: Instanzenzähler


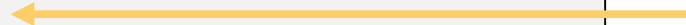


```
public class Konto {  
  
    // class attribute  
    private static int instanceCounter = 0;  
  
    // object attribute  
    private int kontoNummer;  
    private double saldo;  
  
    public Konto(int kNr) {  
        instanceCounter++;  
        kontoNummer = kNr;  
    }  
  
    public Konto(int kNr, int betrag) {  
        instanceCounter++;  
        kontoNummer = kNr;  
        saldo = betrag;  
    }  
  
    // static method  
    public static int getNumInstances(){  
        return instanceCounter;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Konto konto1 = new Konto(1234, 5000);  
  
    Konto konto2 = new Konto(5678, 1000);  
  
    int n = Konto.getNumInstances();  
}
```

Aufruf der statischen Methode:
Was wird zurückgegeben?

Klassen- vs. Objektvariablen

```
public class Example {  
    private static int var1 = 0;    // Klassenattribut  
    private int var2;              // Objektattribut  
    public static int getVar1() {    // Klassenmethode  
        return var2;                  
    }  
    public int getVar2() {          // Objektmethode  
        ...  
    }  
}
```

Innerhalb einer
Klassenmethode
kann man nicht auf
ein Objektattribut
zugreifen!

- Aufruf von

- **Klassenmethoden**

- Example.getVar1();

- **Objektmethoden**

- Example e = new Example(...);
- e.getVar2();

- Klassenmethoden (static) haben Zugriff auf Klassenvariablen (static) und –methoden

- Objektmethoden (non static) haben Zugriff auf Klassen-(non static) und Objektmethoden (static) bzw. Objektattribute (static)!

static → static
static ~~X~~ non static
non static → non static
non static → static

Innere Klassen

- Klassen können innerhalb von anderen Klassen deklariert werden.
 - Details → Programmieren 3!

```
class Out {  
    ...  
    class In {  
        ...  
    }  
}
```

- **Anwendung**
 - Verstecke lokale Typdefinitionen (Klassen), der nur von **einer (äußeren)** Klasse benötigt werden, in einer **inneren** Klasse.
 - Prinzip: Information Hiding bzw. Blackbox-Sicht.

Live Programming: Die Klasse Student

- Attribute
 - Methoden
 - Konstruktoren
-
- Sichtbarkeit
 - Getter/Setter
 - Überladen von Methoden/Konstruktoren
 - Statische Methoden/Attribute
 - Innere Klassen

Grobe Phasen typischer objektorientierter Programme

- **Erzeugen von Objekten in main-Methode**
 - new-Operator
 - Ggfs. initialisieren von Attributen der Objekte innerhalb des Konstruktors.
- **Vernetzung der Objekte**
 - Entweder über Parameter des Konstruktors
 - Oder durch Aufrufen einer Methode
- **Arbeitsphase**
 - Objekte arbeiten zusammen.
- **Abbauphase**
 - Abbau des Objektnetzes
 - Java erkennt automatisch welche Objekte nicht mehr verlinkt/verwendet werden und löscht diese (**Garbage Collection**)