Propädeutikum
Programmierung in der Bioinformatik

Java – Klassen und Objekte

Thomas Mauermeier

27.11.2018

Ludwig-Maximilians-Universität München

Rückblick

Imperative Programmierung

- Variablen
- Fallunterscheidungen
- Schleifen
- statische Methoden

Objektorientierte Programmierung

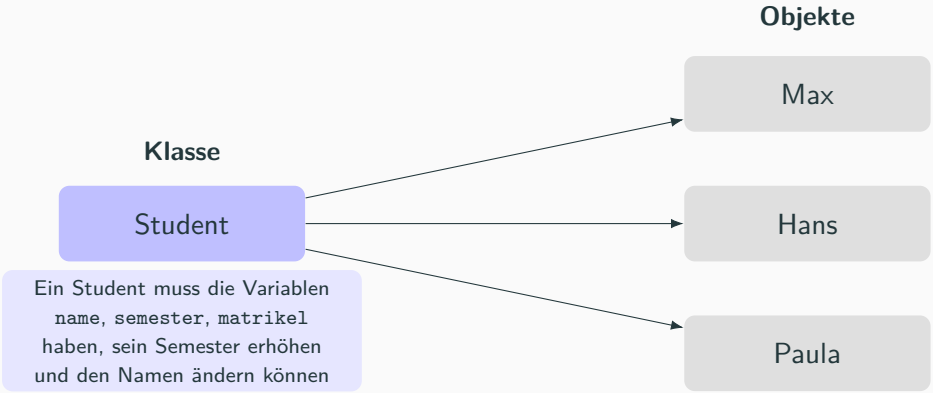
- Klassen
- Objekte
- Vererbung
- Schnittstellen

Bisheriger Programmierstil: imperative Programmierung

```java
public class Student {
  // Beispiel Imperativ
  String s1_name = "Max";
  int s1_semester = 1;
  int s1_matrikel = 12345;
  String s2_name = "Hans";
  int s2_semester = 8;
  int s2_matrikel = 87654;
  String s3_name = "Paula";
  int s3_semester = 4;
  int s3_matrikel = 87651;
  public static void
      ↪ main(String[] args) {
    s1_semester++;
    s3_name = "Pauline";
  }
}
```

- Problem: Schnell unübersichtlich, wenig wiederverwendbare Strukturen
- Wie macht man das **objektorientiert**?

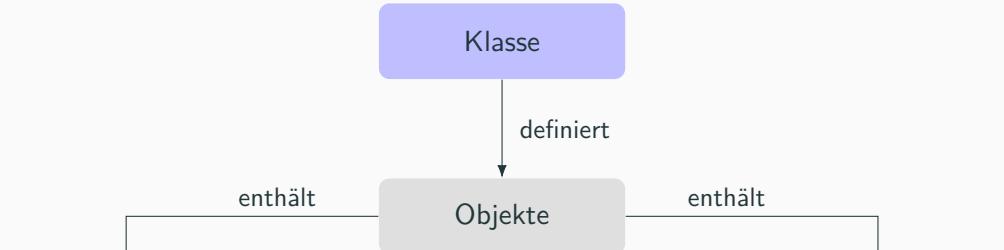
Lösungsansatz: Objektorientierung



abstrakter “Bauplan” für einen Studenten;
Student als Datentyp

konkrete Objekte nach “Bauplan”;
Instanzen der Klasse Student

OOP: Objektorientierte Programmierung



Was **hat** das Objekt?
→ Variablen

Was **kann** das Objekt?
→ Methoden

Klassen: Attribute

- Was **hat** das Objekt? → Variablen aus denen Objekt bestehen soll
- Man deklariert in der Klasse **einmal** die Variablen die **jedes** Objekt haben soll
- Jedes Objekt "besitzt" dann **eigene** Menge von Variablen die es nicht mit anderen Objekten geteilt werden

```
public class Student {
  private String name;
  private int matrikel;
  private int semester;
  (...)
}
```

Klassen: Prozeduren

- Was **kann** das Objekt? → Methoden die ein Objekt haben soll
- Man deklariert in der Klasse **einmal** die Methoden die **jedes** Objekt haben soll
- Methoden arbeiten dann mit den Daten des Objekts (“`this`”), das sie aufruft

```
public class Student {
  (...)
  public void increaseSemester() {
    this.semester++;
  }
  (...)
}
```

Klassen: Konstruktoren

```
public class Student {
  (...)
  public Student(String n,
                 int s,
                 int m) {
    this.name = n;
    this.semester = s;
    this.matrikel = m;
  }
  (...)
}
```

Konstruktor

- besondere Methode
- Definiert wie die Attribute des Objekts "befüllt" werden; also wie Objekt "konstruiert" wird.
- Methode zum **erzeugen** von Objekten mit `new`: `Student s = new Student("Max", 1, 12345)`
- Unterschied zu "normalen" Methoden:
  - Name der Konstruktormethode = Klassenname
  - **kein** Rückgabewert (nicht mal `void`) in Signatur (gibt aber natürlich das erstellte Objekt zurück)

Zugriff auf Objektvariablen und -methoden: Punktnotation und `this`

Punktnotation

Wird verwendet um auf die Variablen (Attribute) oder Methoden (Prozeduren) eines Objekts zuzugreifen:

```
this.name
s.increaseSemester()
```

Referenzieren Variablen bzw. Methoden die in der entsprechenden Klasse für die Objekte `s` bzw. `this` definiert wurden:

```
objekt.variable
objekt.methode()
```

Selbstreferenz: `this`

`this` bezieht sich innerhalb einer Klasse auf das Objekt selbst. Warum sinnvoll?

- Beim schreiben der Klasse: Kein konkretes Objekt vorhanden
- Ich will aber Methoden schreiben die sich auf ein Objekt beziehen
- Deswegen: `this` um sich auf das Objekt zu beziehen, das in Zukunft auf die Methode zugreift

Beispiel: Imperative Deklaration → OOP-Klasse

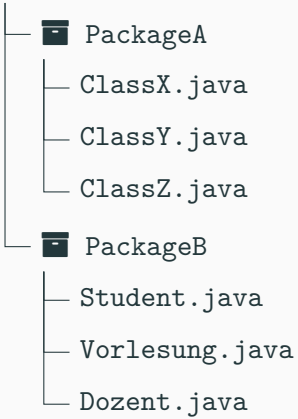
```java
public class Student {
  // Beispiel Imperativ
  String s1_name = "Max";
  int s1_semester = 1;
  int s1_matrikel = 12345;
  String s2_name = "Hans";
  int s2_semester = 8;
  int s2_matrikel = 87654;
  String s3_name = "Paula";
  int s3_semester = 4;
  int s3_matrikel = 87651;

  public static void main(String[]
      ↪ args) {
    s1_semester++;
    s3_name = "Pauline";
  }
}
```

```java
public class Student {
  // ATTRIBUTE: Was *hat* das Objekt?
  private String name;
  private int semester;
  private int matrikel;
  // PROZEDUREN: Was *kann* das Objekt?
  public Student(String n, int s, int m) {
    this.name = n;
    this.semester = s;
    this.matrikel = m;
  }
  public void setName(String name) {
    this.name = name;
  }
  public void increaseSemester() {
    this.semester++;
  }
}
```

Beispiel: Klasse `Student` in Action

```java
public class Student {
  // ATTRIBUTE: Was *hat* das Objekt?
  private String name;
  private int semester;
  private int matrikel;
  // PROZEDUREN: Was *kann* das Objekt?
  public Student(String n, int s, int
      ↪ m) {
    this.name = n;
    this.semester = s;
    this.matrikel = m;
  }
  public void setName(String name) {
    this.name = name;
  }
  public void increaseSemester() {
    this.semester++;
  }
  public static void main(String[]
      ↪ args) {
    Student s1 = new Student("Max", 1,
        ↪ 12345);
    Student s2 = new Student("Hans",
        ↪ 8, 87654);
    Student s3 = new Student("Paula",
        ↪ 4, 87651);
    s1.increaseSemester();
    s3.setName("Pauline");
  }
}
```

Pakete



- Klassen werden oft thematisch/logisch zu **Paketen** gebündelt
- Klassen in verschiedenen Paketen müssen evtl. miteinander kommunizieren
- Java ermöglicht Regulation der Kommunikation mit **Sichtbarkeitsmodifizierern**. Warum? Beispiele:
  - fremdes Paket soll nicht auf ein Passwort o.ä. zugreifen das in einem Objekt gespeichert ist
  - Methoden die außerhalb ihrer Klasse nicht aufgerufen werden dürfen/sollten, weil sie isoliert keinen Sinn machen

Sichtbarkeitsmodifizierer

Sichtbarkeitsmodifizierer gibt es für:

- Klassen: `public class Student`
- Variablen: `private int matrikel;`
- Methoden: `public void increaseSemester()`

und bestimmen **von wo aus** die Eigenschaft gesehen werden kann.

	sichtbar in/zugreifbar aus			
Eigenschaft ist	**Klasse**	**Package**	**Unterklasse**	**Welt**
`public`	✔	✔	✔	✔
`protected`	✔	✔	✔	✖
ohne	✔	✔	✖	✖
`private`	✔	✖	✖	✖

Empfehlung: Sichtbarkeitsmodifizierer so restriktiv wie möglich wählen

Fremdzugriff: Getter und Setter

- **Goldstandard:** Attribute alle auf `private` setzen.
- **Problem:** Wie greife ich dann auf die Variablen zu?
- **Lösung:** `public` Getter- und Setter-Methoden für entsprechende Variablen erstellen
- **Vorteil:** z.B. Möglichkeit in einem Setter eine Validierung einzubauen, was bei direktem Zugriff nicht geht.

```
public class Beispiel {
  private String foo;
  public String getFoo() {
    return this.foo;
  }
  public void setFoo(String s) {
    if (validForm(s) == true) {
      this.foo = s;
    } else {
    System.out.println("Fehler!");
    }
  }
}
```

Statische Variablen und Methoden

Was wenn ich aber **möchte**, dass sich Objekte etwas teilen?

⇓

Schlüsselwort `static`

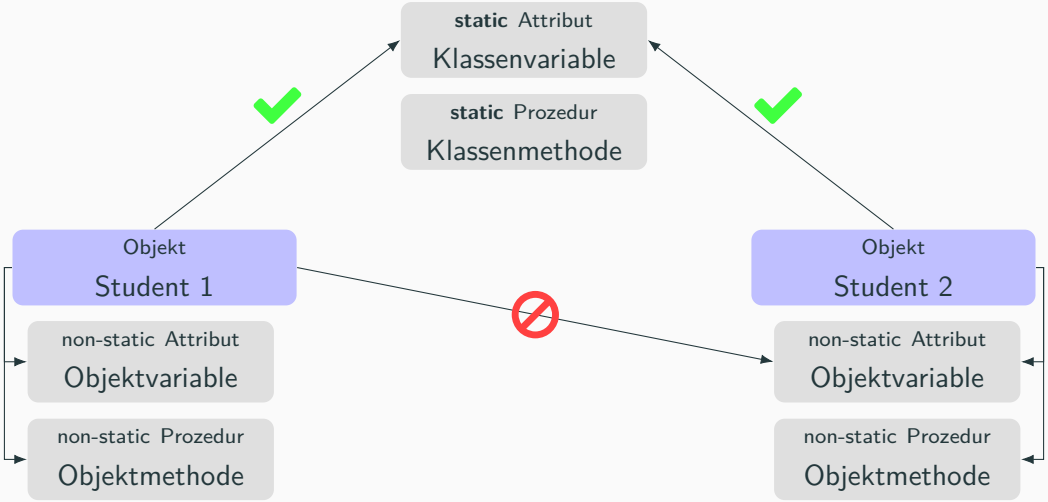
```
private static int foo
```

- Erzeugt **eine** Variable auf die **alle** Objekte zugreifen können
- Variable existiert bereits **vor** der Erzeugung eines Objekts
- Zugriff via `objekt.foo` oder `Klassenname.foo`

```
public static void bar()
```

- Analog zu statischen Variablen
- `this` im Körper der Methode sinnlos, keine Verbindung zu konkretem Objekt
- Zugriff über `objekt.bar()` möglich, **aber** um zu verdeutlichen dass es keine **Objektmethode** ist: Aufruf via `Klassenname.bar()`

Statische Variablen und Methoden



Overloading von Methoden

Java unterscheidet Methoden mit:

- **selbem** Namen
- **selbem** Return Type
- und **unterschiedlichen** Parametern

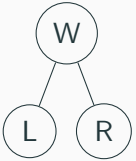
```
public class Foo {
  public int foo(int a) {
    return a;
  }
  public int foo(int a, int b) {
    return a+b;
  }
}
```

Wenn nun z.B. `foo(42)` aufgerufen wird, erkennt Java an den Parametern, dass die erste Methode gemeint ist.

Analog würde z.B. bei `foo(18,24)` erkannt, dass die zweite Methode gemeint ist.

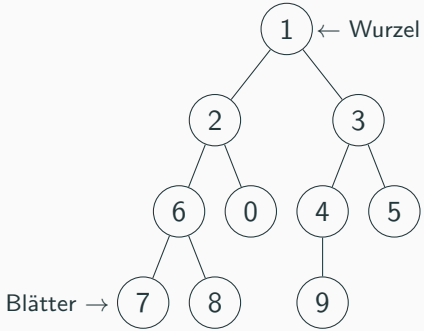
Ausflug in Datenstrukturen: Binärbaum

```
public class BinaryNode {
  private BinaryNode left;
  private BinaryNode right;
}
```



- **W** = Wurzelknoten
- **L** = Linkes Kind der Wurzel
- **R** = Rechtes Kind der Wurzel

Verknüpfung mehrerer `BinaryNodes`:



⚠ Kindknoten `left` und `right` dürfen auch `null` sein!