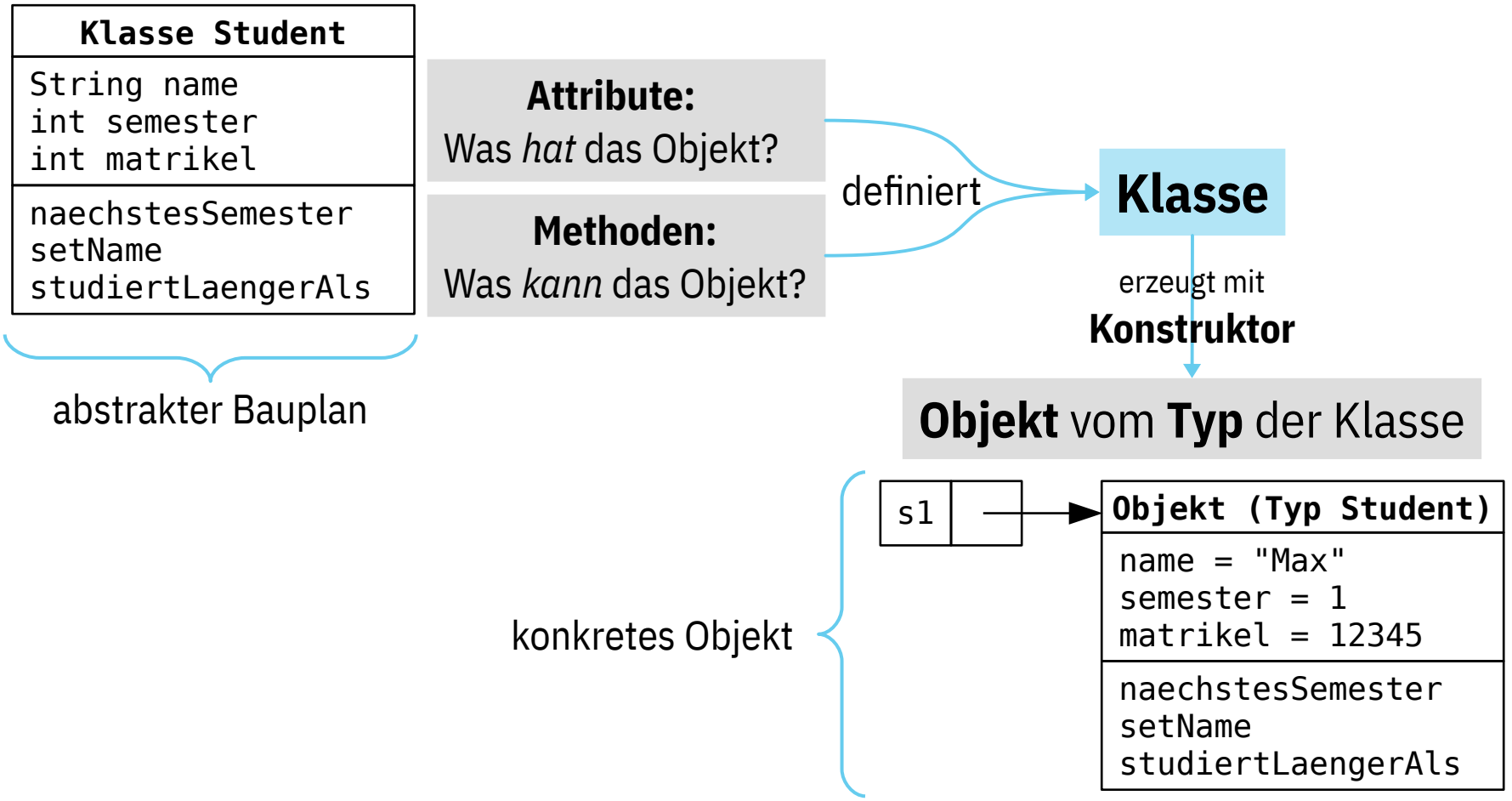
Propädeutikum: Programmierung in der Bioinformatik

Klassen und Objekte II

Thomas Mauermeier
26.11.2019
Ludwig-Maximilians-Universität München

Rückblick


Klasse Student
String name
int semester
int matrikel
naechstesSemester
setName
studiertLaengerAls
abstrakter Bauplan
Attribute:
Was *hat* das Objekt?
Methoden:
Was *kann* das Objekt?
definiert
Klasse
erzeugt mit
Konstruktor
Objekt vom Typ der Klasse
konkretes Objekt
s1
Objekt (Typ Student)
name = "Max"
semester = 1
matrikel = 12345
naechstesSemester
setName
studiertLaengerAls

Heute

Getter, Setter und Pakete

Keyword `static`

Overloading von Methoden

Getter und Setter: Warum eigentlich?

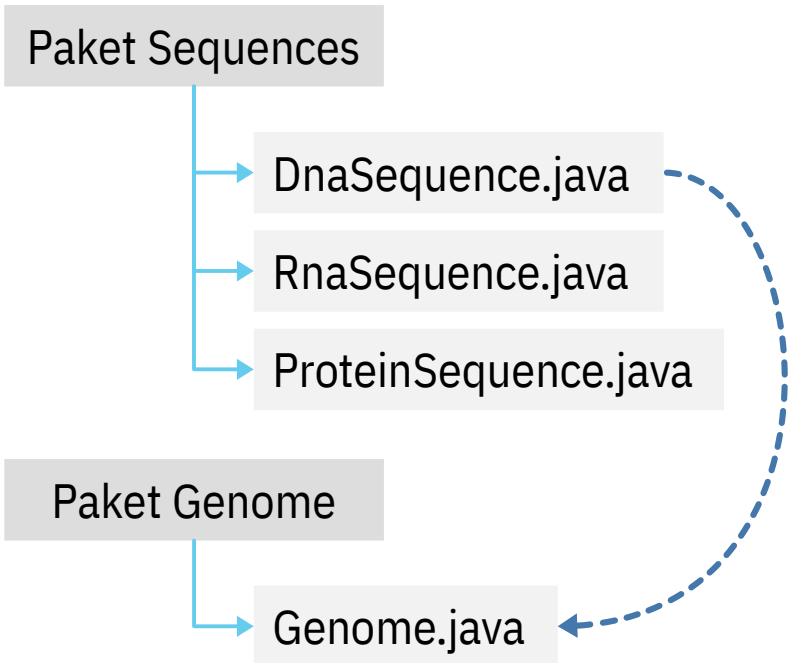
```
public class Student {
    String name;
    int semester;
    int matrikel;

    public Student(String name, int semester, int matrikel) {
        this.name = name;
        this.semester = semester;
        this.matrikel = matrikel;
    }
    public void naechstesSemester() {
        this.semester++;
    }
    public void setName(String name) {
        this.name = name;
    }
    public boolean studiertLaengerAls(Student s) {
        return this.semester > s.semester;
    }
    // main-Methode rauskommentiert
}
```

Warum eigentlich?

Man könnte ja auch ohne **`setName(String name)`** einfach auf die Variable **`name`** zugreifen?

Pakete



Ich möchte die Klasse **DnaSequence** in der Klasse **Genome** verwenden!

Dazu sind zwei Dinge notwendig:

- **Sichtbarkeit:**
  DnaSequence muss von *Genome* aus sichtbar sein
- **Import:** (wie z.B. bei ArrayList)
  In *Genome* muss *DnaSequence* importiert werden

Ausführlicher: http://openbook.rheinwerk-verlag.de/javainsel/03_006.html#u3.6

Sichtbarkeitsmodifikatoren

```java
public class Student {
    String name;
    int semester;
    int matrikel;
    public Student(String name, int semester, int matrikel) {
        this.name = name;
        this.semester = semester;
        this.matrikel = matrikel;
    }
    public void naechstesSemester() {
        this.semester++;
    }
    public void setName(String name) {
        this.name = name;
    }
    public boolean studiertLaengerAls(Student s) {
        return this.semester > s.semester;
    }
}
```

Sichtbarkeitsmodifikatoren

```java
public class Student {
    String name;
    int semester;
    int matrikel;
    public Student(String name, int semester, int matrikel) {
        this.name = name;
        this.semester = semester;
        this.matrikel = matrikel;
    }
    public void naechstesSemester() {
        this.semester++;
    }
    public void setName(String name) {
        this.name = name;
    }
    public boolean studiertLaengerAls(Student s) {
        return this.semester > s.semester;
    }
}
```

→ Sichtbarkeitsmodifikatoren

Sichtbarkeitsmodifikatoren

Gibt es für verschiedene Bestandteile:

- Klassen: `public class Student { … }`
- Variablen: `private int matrikel;`
- Methoden: `public void setName(String name) { … }`

Modifikator ist:	Bestandteil ist sichtbar in/zugreifbar aus			
	selber Klasse	selbes Package	Unterklasse	"Welt"
`public`	✔	✔	✔	✔
`protected`	✔	✔	✔	✖
ohne	✔	✔	✖	✖
`private`	✔	✖	✖	✖

Ausführlicher: http://openbook.rheinwerk-verlag.de/javainsel/05_002.html#u5.2

Getter und Setter: Warum eigentlich?

```
public class Student {
    String name;
    int semester;
    int matrikel;

    public Student(String name, int semester, int matrikel) {
        this.name = name;
        this.semester = semester;
        this.matrikel = matrikel;
    }
    public void naechstesSemester() {
        this.semester++;
    }
    public void setName(String name) {
        this.name = name;
    }
    public boolean studiertLaengerAls(Student s) {
        return this.semester > s.semester;
    }
    // main-Methode rauskommentiert
}
```

Warum eigentlich?

Man könnte ja auch ohne **`setName(String name)`** einfach auf die Variable **`name`** zugreifen?

Getter und Setter: Deshalb!

```java
public class Student {
    private String name;
    private int semester;
    private int matrikel;
    // Konstruktor herauskommentiert
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSemester() {
        return semester;
    }
    public void setSemester(int semester) {
        this.semester = semester;
    }
    public int getMatrikel() {
        return matrikel;
    }
    public void setMatrikel(int matrikel) {
        this.matrikel = matrikel;
    }
}
```

Variablen üblicherweise auf private gesetzt:
Zugriff von Außen nur über Getter und Setter
→ Sicherstellen, *wie* zugegriffen/geändert wird

Getter und Setter: Deshalb!

```java
public class Student {
    private String name;
    private int semester;
    private int matrikel;
    // Konstruktor herauskommentiert
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public int getSemester() {
        return semester;
    }
    public void setSemester(int semester) {
        this.semester = semester;
    }
    public int getMatrikel() {
        return matrikel;
    }
    public void setMatrikel(int matrikel) {
        this.matrikel = matrikel;
    }
}
```

Variablen üblicherweise auf private gesetzt:
Zugriff von Außen nur über Getter und Setter
→ Sicherstellen, *wie* zugegriffen/geändert wird

Getter und Setter-Methoden für Variablen:
Für jede Variable:

- eine *getVariable*-Methode zum *holen*
- eine *setVariable*-Methode zum *verändern*

In diesen kann man dann den “richtigen” Zugriff sicherstellen

Tipp: Getter/Setter generieren in IntelliJ, Eclipse

Heute

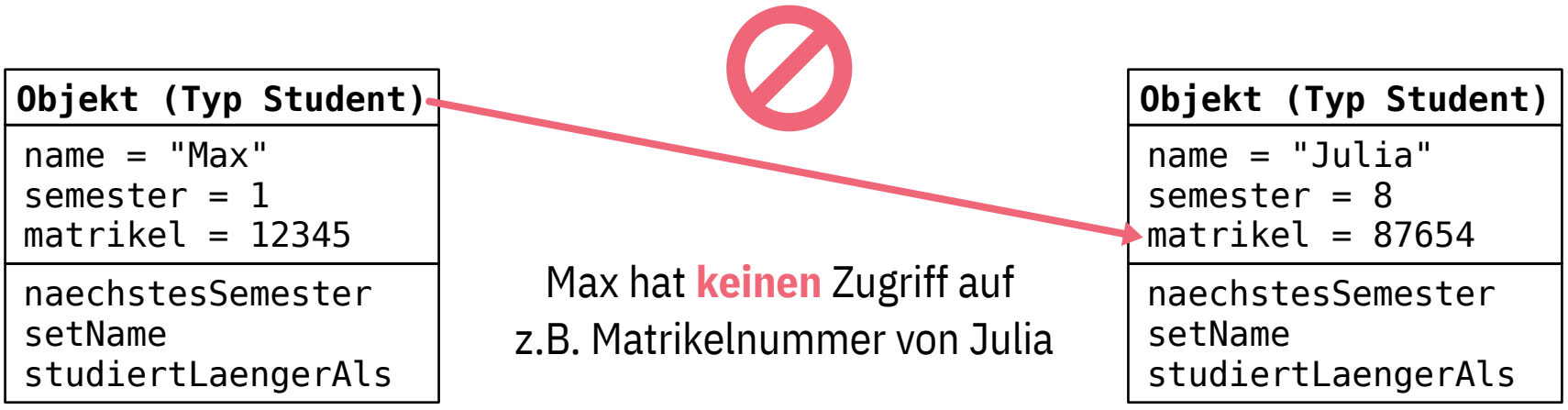
Getter, Setter und Pakete

Keyword `static`

Overloading von Methoden

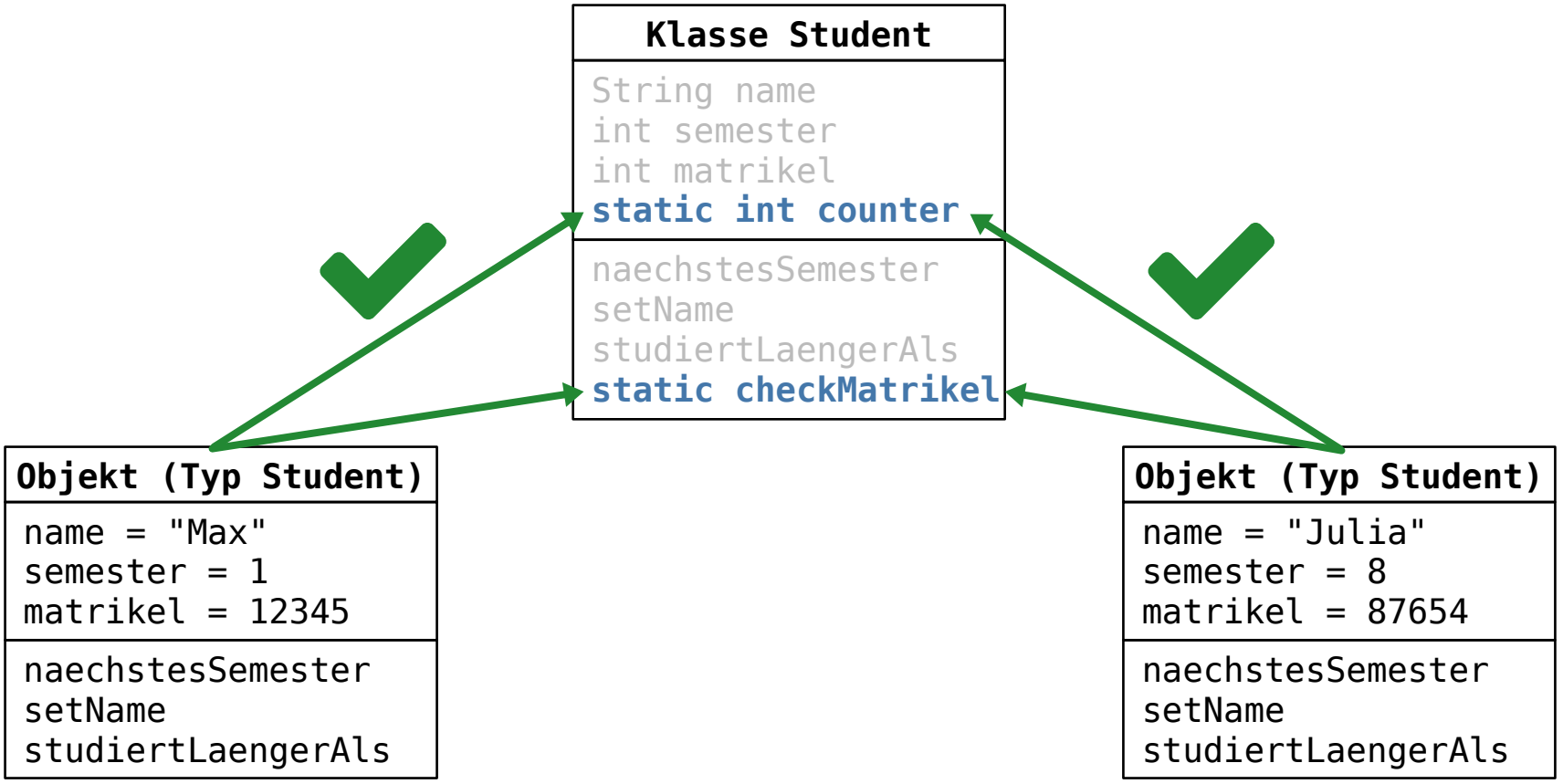
Keyword static

Bisher: Jedes Objekt hat seine “eigene” Instanz der Variablen und Methoden:



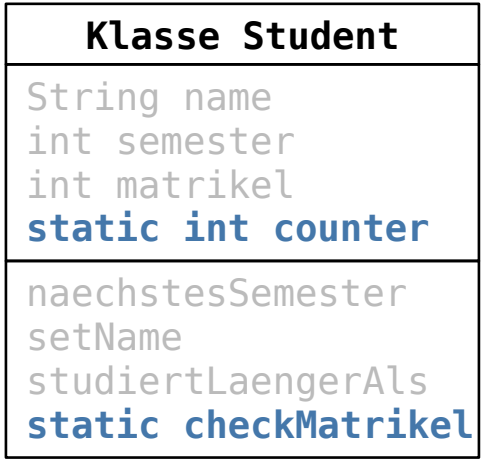
Aber: Was, wenn ich eigentlich *möchte*, dass sich Objekte eine Variable/Methode teilen?

Keyword static



Keyword static

Wenn eine Variable oder eine Methode als **static** deklariert ist, existiert sie *unabhängig* von Objekten.



Zugriff entweder wie gehabt über ein **Objekt**:

```
s1.counter
s1.checkMatrikel(...)
```

Achtung: Kein Zugriff auf das aufrufende Objekt mit this oder ähnlichem mehr möglich.
→ **Unabhängigkeit von Objekten!**

Oder über den **Klassennamen**:

```
Student.counter
Student.checkMatrikel(...)
```

Besser: Deutlich, dass es sich hier um etwas handelt, das sich *alle* Objekte des Typs teilen.

Beispiel: statische Variable im Einsatz

Idee: Ich möchte beim erstellen von Exons jedem Exon eine fortlaufende Zahl als ID geben

```
public class Exon {
    private int start, end;
    private int id;
    private static int counter = 0;

    public Exon(int start, int end, int id) {
        this.start = start;
        this.end = end;
        this.id = counter+1;
        counter++;
    }
}
```

NB: existiert unabhängig von jedem Objekt!

`id` wird auf Wert von **`counter`** gesetzt, danach wird **`counter`** für *nächstes* Objekt erhöht

Beispiel: statische Methode im Einsatz

Übung 1 *Erweiterung von* `DnaSequence`

Falls du bisher noch nicht Übung 1 vom Übungsblatt der letzten Woche gemacht hast, sollte diese erst fertiggestellt werden. Alternativ steht eine fertige `DnaSequence`-Klasse auf der Propädeutikumswebsite zum Download bereit. Versichere dich aber, dass du verstehst was die Klasse macht bevor du losprogrammierst.

1. Setze den Sichtbarkeitsmodifikator von allen Attributen in `DnaSequence` auf `private`. Schreibe dann zu allen Attributen der Klasse jeweils einen Getter und einen Setter.
2. Schreibe danach eine statische Variante von `isValidSequence()` mit der Signatur `isValidSequence(String seq)`. Überlege dir warum es sinnvoll sein kann, gerade diese Methode statisch zu implementieren.
3. Schreibe einen *zusätzlichen* Konstruktor `DnaSequence(String sequence)`. Dieser soll, nachdem keine `id` übergeben wird, der Sequenz einfach eine fortlaufende Zahl als `id` zuweisen.

Heute

Getter, Setter und Pakete

Keyword `static`

Overloading von Methoden

Overloading?

Angenommen ich habe diesen Konstruktor:

```
public DnaSequence(String id, String sequence) {
        this.id = id;
        this.sequence = sequence;
}
```

Jetzt will ich aber z.B. ein **DnaSequence**-Objekt *ohne* **id** erstellen. Aber wie?

Eine Idee wäre:

```
DnaSequence dnaseqNoId = new DnaSequence(null, "GATTACA");
```

Problem: Nicht wirklich elegant, macht eventuell Schwierigkeiten.
Lösung: **Overloading!**

Overloading?

Ich “**überlade**” (→ Overloading) den Konstruktor indem ich einen zweiten Konstruktor schreibe:

```
public DnaSequence(String id, String sequence) {
        this.id = id;
        this.sequence = sequence;
}

public DnaSequence(String sequence) {
        this.id = "NO_ID"; // default value definieren
        this.sequence = sequence;
}
```

Was entscheidet welcher Konstruktor genutzt wird?

Reihenfolge der Typen und *Anzahl* der Argumente beim Aufruf

Overloading

Gleichzeitiges definieren von **gleichnamigen** Methoden mit:

- unterschiedlicher **Anzahl** von Parametern
- unterschiedlicher **Reihenfolge** der Typen der Parameter

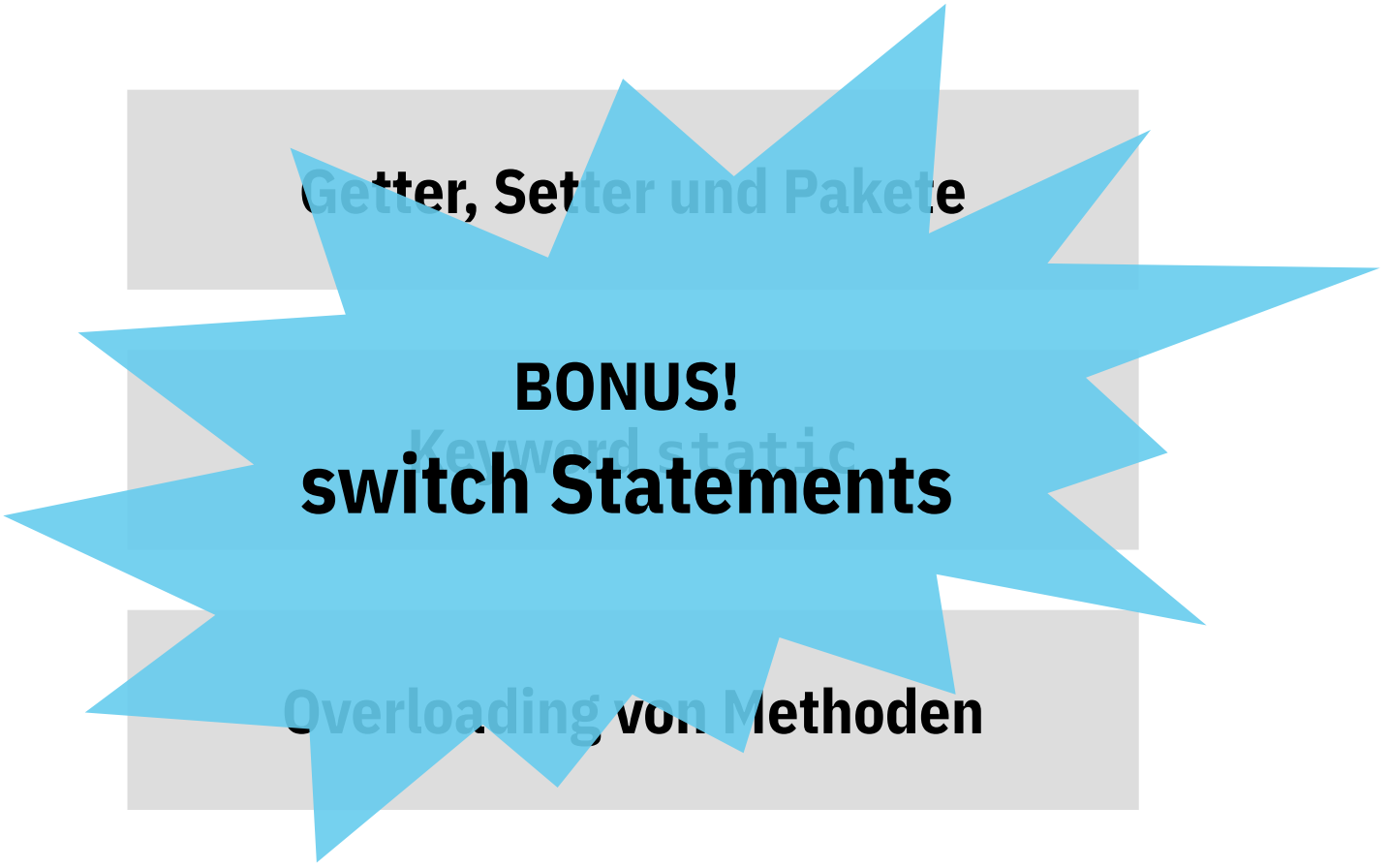
```
public int foo(int a, int b) {
    return a + b;
}
public int foo(int a) {
    return a;
}
public boolean foo() {
    return true;
}
public int foo(String a, int b) {
    return b;
}
public int foo(int b, String a) {
    return b;
}
```

Unterschiedliche Anzahl von Parametern

zur Demonstration: anderer Return Type ist OK

Reihenfolge unterschiedlich; Typen, Anzahl gleich

Heute



switch Statement

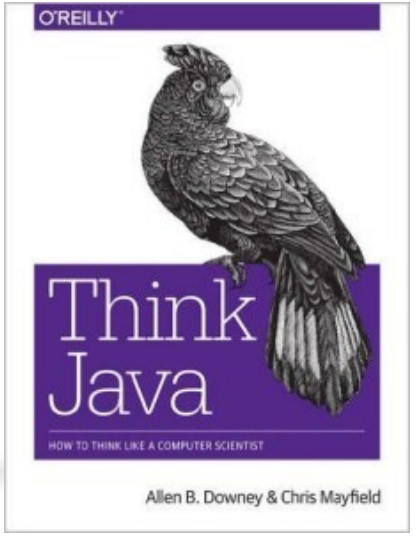
```java
if (obst == "Apfel") {
    System.out.println("Lecker!");
} else if (obst == "Birne") {
    System.out.println("Naja.");
} else if (obst == "Orange") {
    System.out.println("Och nö");
} else {
    System.out.println("Hunger!");
}
```

```java
switch (obst) {
        case "Apfel":
            System.out.println("Lecker!");
            break;
        case "Birne":
            System.out.println("Naja.");
            break;
        case "Orange":
            System.out.println("Och nö");
            break;
        default:
            System.out.println("Hunger!");
            break;
    }
}
```

Ausführlicher: http://openbook.rheinwerk-verlag.de/javainsel/02_005.html#u2.5.4

Buchempfehlungen

Onlineversion



kostenloses E-Book:

- 1. Auflage
- 2. Auflage (Draft)

E-Book zugänglich über Uni-Bibliothek

u.v.m.

(findet etwas, das zu eurem Lernstil passt)