

# Programmieren I

## Vererbung



Heusch 13  
Ratz 9

Institut für Automation und angewandte Informatik

```
final List<String> allResults = new ArrayList<String>();  
final Map<String, Integer> typeWordResultCount = new HashMap<String, Integer>();  
final Map<String, Integer> typePoints = new HashMap<String, Integer>();  
evaluation.put(type, typePoints);  
  
for (final Sheet sheet : this.sheets) {  
    final String sheetResult = sheet.getPlayerInput(type);  
    if (sheetResult.startsWith(start) && this.isValidWord(sheetResult, type)) {  
        validWordCountForType++;  
        allResults.add(sheetResult);  
    }  
}
```

# Wiederholung: Klassen in Java

- Klassen beschreiben Daten (**Attribute**) und Verhalten (**Methoden**) von Objekten
- Klassennamen beginnen mit einem Großbuchstaben (Konvention)
- Konstruktoren sind z.B. für die Initialisierung der Variablen verantwortlich
- **Konstruktoren** sind Klassenmethoden, die bei der Erzeugung eines Objektes aufgerufen werden.

```
Automotive ente = new Automotive(/* ... */);
```

- Objekterzeugung und Initialisierung sind miteinander verknüpft
- Standardkonstruktor wird von Java bereitgestellt, wenn vom Programmierer kein Konstruktor angegeben wird (und nur dann!)
- Konstruktoren haben Namen der Klasse und keinen Rückgabewert
- Beliebige Anzahl von Konstruktoren möglich (auch Überladen)

# Aufruf von Methoden

- Instanzmethoden hängen an einem Objekt.
- Implizites `this`. stellt ein Handle auf das Objekt dar.

```
class Fruit {  
    String name;  
  
    Fruit(String name) {  
        this.name = name;  
    }  
    void prt() {  
        System.out.println("I am a " + this.name);  
    }  
  
    public static void main(String args[]) {  
        Fruit a = new Fruit("strawberry");  
        Fruit b = new Fruit("cherry");  
        Fruit c = new Fruit("lemon");  
        a.prt();  
        b.prt();  
        c.prt();  
    }  
}
```

**Namenskonflikt mit `this`. auflösen**

**Hier wird `prt()` jeweils „passend für das Objekt“ aufgerufen**

# Struktur eines Java-Programmes mit mehreren Klassen („Komposition“)

```
class Class1 {  
  
    int coefficient;  
    int power;  
  
    // Konstruktoren  
    // ...  
    // Methoden  
    // ...  
    public static void main(String[] args) {  
        // ...  
    }  
}
```

```
class Class2 {  
  
    Class1[] class1Array; // benutzt Class1  
  
    // Konstruktoren  
    // ...  
    // Methoden  
    // ...  
    public static void main(String[] args) {  
        // ...  
    }  
}
```

# Java-Klassen auf Dateien verteilen

- Definition der Klassen `Class1` und `Class2` können prinzipiell in einer oder auch in zwei Dateien stehen:
  - `Class2.java` (mit Klasse `Class2` und `Class1`)
  - `Class2.java` (mit Klasse `Class2`)  
und  
`Class1.java` (mit Klasse `Class1`)
- Wenn mehrere Klassen in derselben Datei stehen, darf nur eine davon „`public`“ sein
- Jede Klasse darf eine als

```
public static void main(String[] args) { /* ... */ }
```

definierte Methode besitzen
- Pro Klasse wird (durch `javac`) eine `.class`-Datei erzeugt, d.h. ggf. mehrere `.class`-Dateien pro `.java`-Datei

# Modifikatoren

- Java kennt die folgenden Modifikatoren:
  - `public`
  - `protected`
  - `private`
  - *package scoped, friendly* (Standard, kein Modifikator)
  
  - `static`
  - `final`
  - `transient`
  - `volatile`
- Diese wirken sich auf die Sichtbarkeit, Lebensdauer und Veränderbarkeit einer Klasse oder eines Elements der Klasse (Variable oder Methode) aus.



**Heusch 12.3.8**  
**Ratz 9.8.2**

# Modifikatoren für die Sichtbarkeit

- `public`, `protected`, `package scoped` und `private` regeln die **Sichtbarkeit** von Klassen\*, Variablen und Methoden entsprechend der folgenden Tabelle:

	<code>public</code>	<code>protected</code>	<code>Package (Standard)</code>	<code>private</code>
Eigene Klasse	ja	ja	ja	ja
Package	ja	ja	ja	nein
Unterklasse	ja	ja	nein**	nein
Fremde Klasse	ja	nein	nein**	nein

- Sie werden bei der Definition vorangestellt, z.B.:

```
private final double pi = 3.1415;
public static void main(String[] args){/*...*/}
```

\* Bei Klassen können nur innere Klassen (2. Semester) `protected/private`-Modifikator haben

\*\* sofern in anderem Package

# Weitere Modifikatoren (1)

## ■ **static**

Variable oder Methode ist nicht an die Existenz eines Objekts gebunden. Lassen sich über die Referenzierung des Klassennamens ansprechen.

Variablen existieren nur 1x je Klasse!

## ■ **final**

Als `final` deklarierte Elemente gelten als konstant und können nachträglich nicht mehr verändert werden.

Von finale Klassen können keine Klassen abgeleitet werden.



## Weitere Modifikatoren (2)

### ■ **transient**

Alle als `transient` markierte Elemente einer Klasse markieren diese als *nicht persistent*.

Sie werden beim Vorgang der Serialisierung und Deserialisierung ignoriert.

### ■ **volatile**

Diese Elemente signalisieren, dass das aktuelle Element asynchron ist, sprich von außerhalb des aktuellen Threads verändert werden kann.

# Vererbung: Motivation (1)

- Beispiel: Benutzerverwaltung (z.B. im Rechenzentrum) mit mehreren Kategorien von Benutzern (Studenten, Mitarbeiter und externe Personen).
- Für Personen aus den verschiedenen Gruppen werden unterschiedliche Informationen erhoben:

```
class Person {  
  
    private int userId;  
    private String name;  
    private String phoneNo;  
  
    public Person(int userId, String name, String phoneNo) {  
        this.userId = userId;  
        this.name = name;  
        this.phoneNo = phoneNo;  
    }  
  
}
```

# Vererbung: Motivation (2)

## ■ Benutzergruppe Student

```
class Student {  
  
    private int userId;  
    private String name;  
    private String phoneNo;  
    private String matrNo;  
  
    public Student(int userId, String name, String phoneNo, String matrNo) {  
        this.userId = userId;  
        this.name = name;  
        this.phoneNo = phoneNo;  
        this.matrNo = matrNo;  
    }  
}
```

# Vererbung: Motivation (3)

## ■ Benutzergruppe Mitarbeiter

```
class Staff {  
  
    private int userId;  
    private String name;  
    private String phoneNo;  
    private String grade;  
  
    public Staff(int userId, String name, String phoneNo, String grade) {  
        this.userId = userId;  
        this.name = name;  
        this.phoneNo = phoneNo;  
        this.grade = grade;  
    }  
}
```

## Vererbung: Motivation (4)

### ■ Frage:

Können wir alle drei Benutzergruppen in einem einzigen Feld (Array) gemeinsam verwalten?

### ■ Einfache Antwort:

Nein, denn es handelt sich um drei verschiedene Klassen; die Objekte haben verschiedenen Typ.

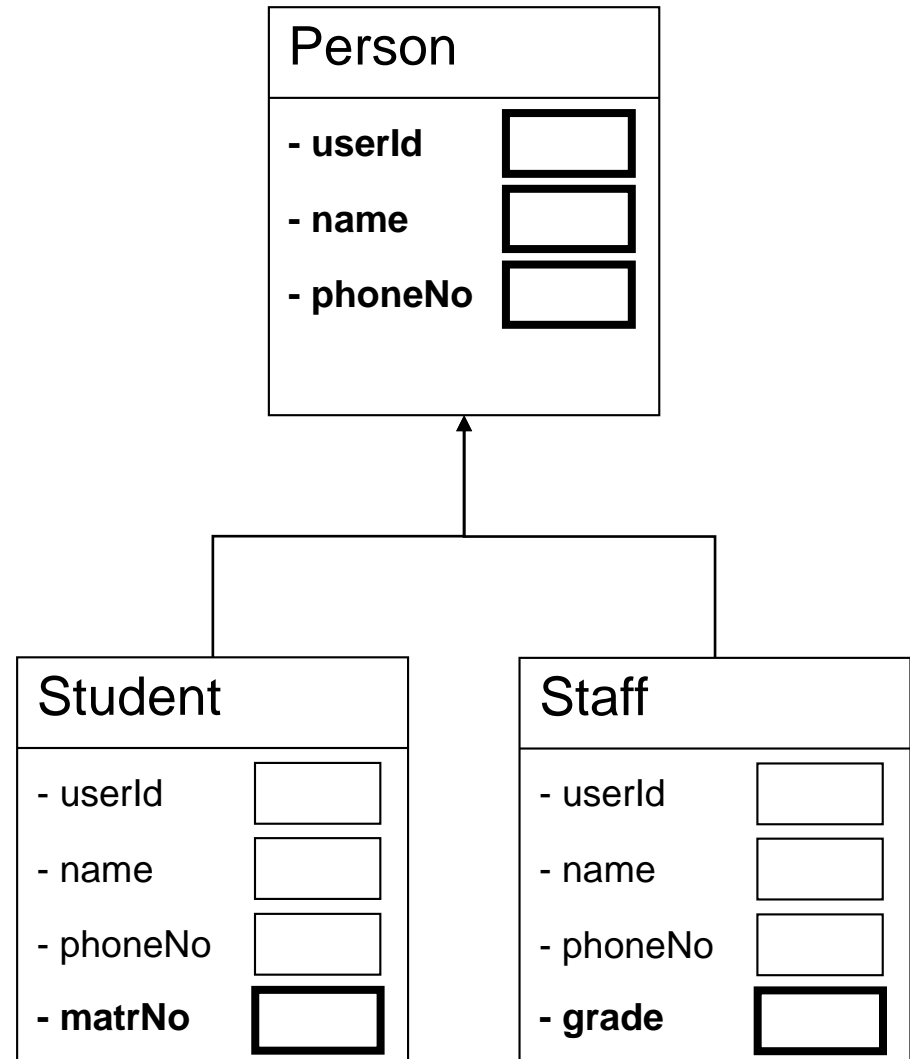
### ■ Differenzierte Antwort:

Doch, wir können die Objekte in einem einzigen Feld (Array) speichern, wenn wir die Gemeinsamkeiten der drei Klassen extrahieren!

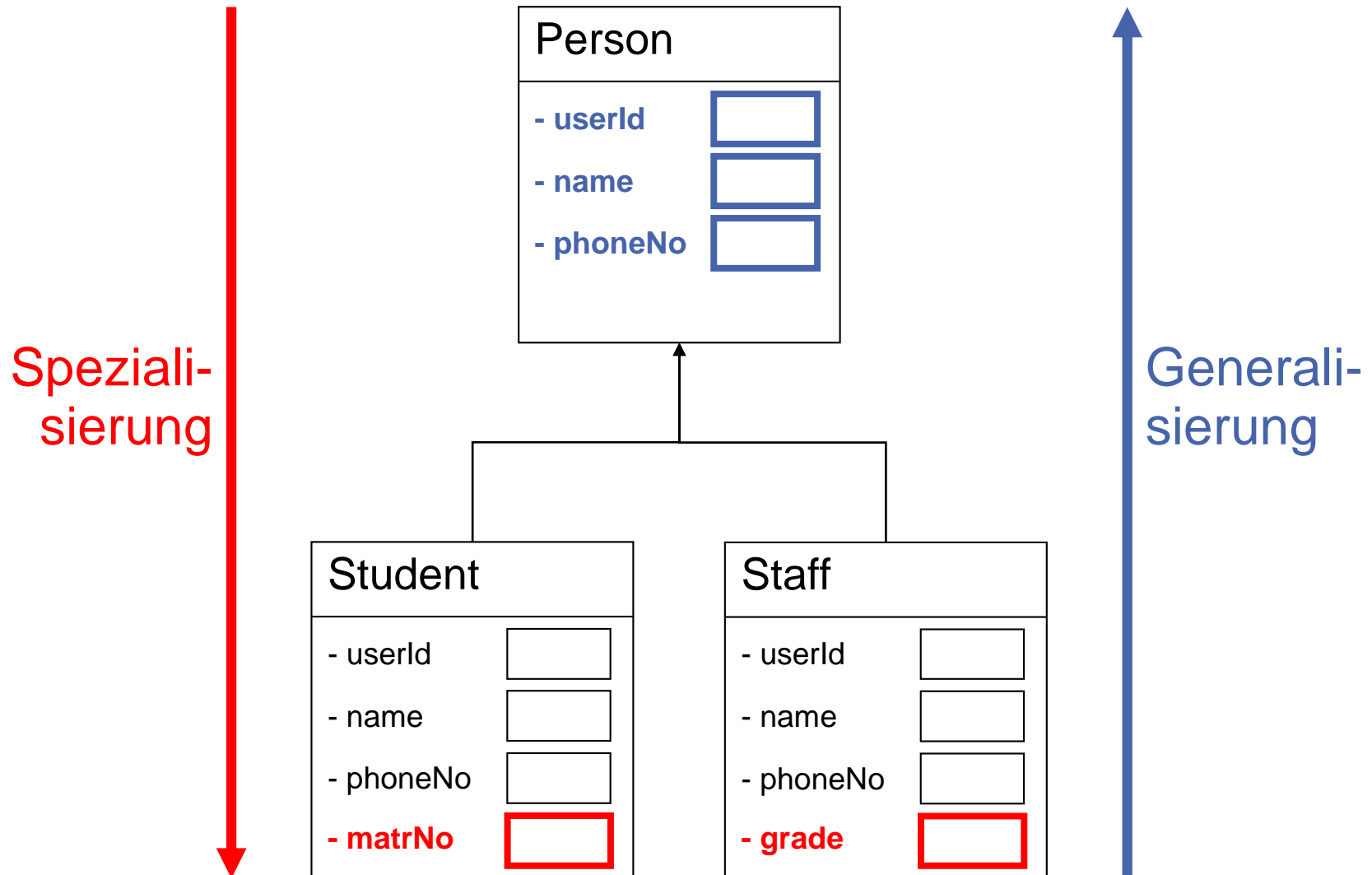
Beobachtung: Die gemeinsame Information von Studenten und Mitarbeitern sind für Personen gespeichert.

# Vererbung: Grundprinzip

- Student und Staff erben alle Attribute von Person
- alles was man mit einer Instanz der Klasse Person „machen kann“, kann man auch mit einer Instanz der Klassen Staff und Student machen!



# Vererbung: Spezialisierung und Generalisierung



# Vererbung: Beispiel (1)

```
class Person {  
    protected int userId;  
    String name;  
    String phoneNo;  
  
    public Person(){ }  
  
    public Person(int userId, String name, String phoneNo) {  
        this.userId = userId;  
        this.name = name;  
        this.phoneNo = phoneNo;  
    }  
}
```

Die Klasse `Person` wird fast wie gehabt definiert. Aus einem Grund, den wir später noch kennen lernen werden, benötigen wir im Augenblick noch den *Standardkonstruktor*.



## Vererbung: Beispiel (2)

```
class Student extends Person {  
  
    String matrNo;  
  
    public Student(int userId, String name, String phoneNo, String matrNo) {  
        this.userId = userId;  
        this.name = name;  
        this.phoneNo = phoneNo;  
        this.matrNo = matrNo;  
    }  
}
```

- `extends` besagt, dass die Klasse `Student` von der Klasse `Person` erbt.
- Weil die Klasse `Student` von der Klasse `Person` erbt, besitzt die Klasse `Student` auch alle Attribute der Klasse `Person` (ohne dass die Attribute explizit definiert sind).

## Vererbung: Beispiel (3)

```
class Staff extends Person {  
  
    String grade;  
  
    public Staff(int userId, String name, String phoneNo, String grade) {  
        this.userId = userId;  
        this.name = name;  
        this.phoneNo = phoneNo;  
        this.grade = grade;  
    }  
}
```

- `extends` besagt wieder, dass die Klasse `Staff` von der Klasse `Person` erbt.
- Weil die Klasse `Staff` ebenfalls von der Klasse `Person` erbt, besitzt auch die Klasse `Staff` alle Attribute der Klasse `Person`.

## Vererbung: Beispiel (4)

- Die neuen Definitionen sind im wesentlichen identisch mit den alten separaten der Definitionen der Klassen.
- Der Unterschied ist: Jetzt können Objekte der Klasse `Student` und `Staff` an eine Variable des (gemeinsamen Ober-)Typs `Person` zugewiesen werden, da sie diese Klasse spezialisieren!

```
Person[] pers = new Person[3];
```

```
pers[0] = new Staff(1, "Doe", "(012) 345 678", "RA");  
pers[1] = new Student(2, "Bloggs", "(098) 4711", "0815");  
pers[2] = new Person(5, "Citizen", "(054) 4242");
```

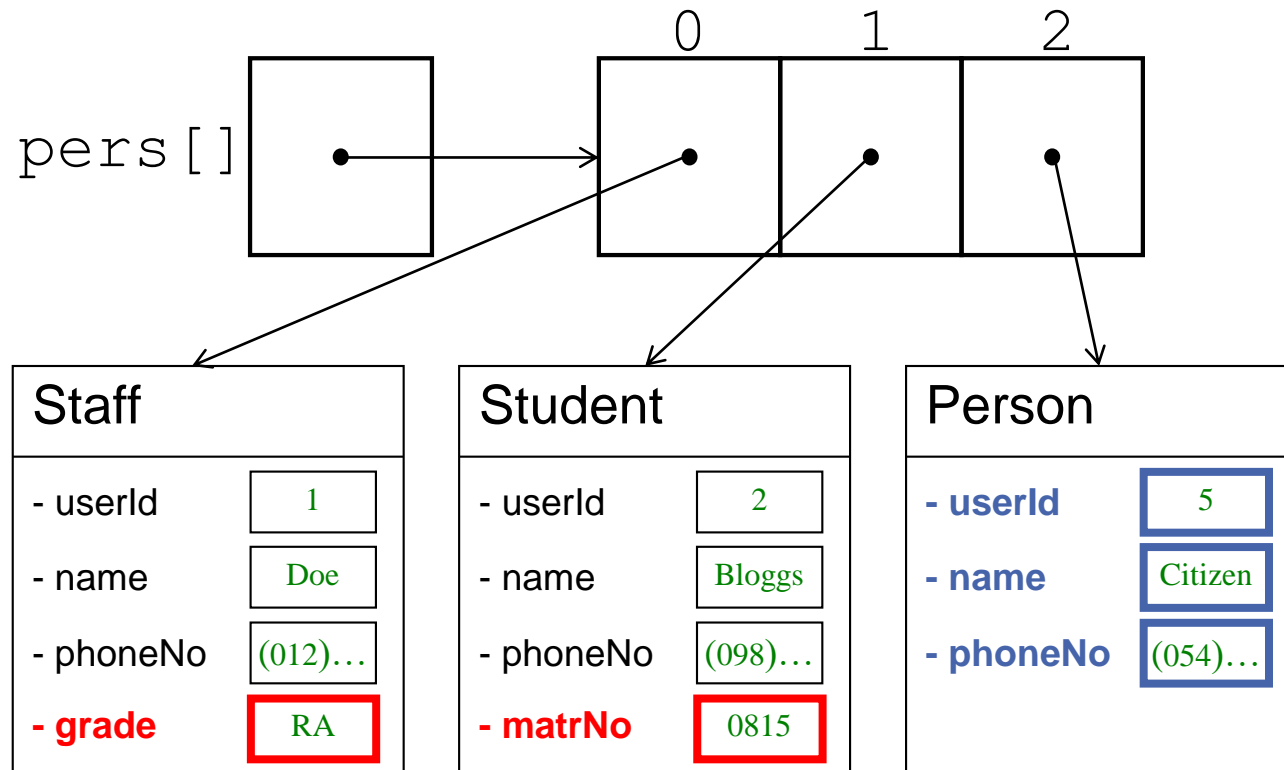
# Vererbung: Beispiel (5)

```
Person[] pers = new Person[3];
```

```
pers[0] = new Staff(1, "Doe", "(012) 345 678", "RA");
```

```
pers[1] = new Student(2, "Bloggs", "(098) 4711", "0815");
```

```
pers[2] = new Person(5, "Citizen", "(054) 4242");
```



## Vererbung: Beispiel (6)

```
Person[] pers = new Person[3];

pers[0] = new Staff(1, "Doe", "(012) 345 678", "RA");
pers[1] = new Student(2, "Bloggs", "(098) 4711", "0815");
pers[2] = new Person(5, "Citizen", "(054) 4242");

for (Person p : pers) {
    System.out.println(p.name + ": " + p.phoneNo);
}
```

### ■ Das Prinzip:

- Ein Objekt einer Klasse B, die von Klasse A abgeleitet ist (von A erbt), besitzt zusätzlich zu den in der Klasse B definierten Attributen und Methoden alle Attribute und Methoden der Klasse A.
- Ein Objekt einer Klasse B kann an eine Variable der Klasse A zugewiesen werden, wenn Klasse B von Klasse A abgeleitet ist.

# Vererbung: Beispiel (7)

## ■ Das Prinzip...

- Die umgekehrte Richtung ist nicht zulässig!

```
Staff staff;  
staff = new Person(1, "Doe", "012 345 678");
```



- Ein Objekt, das in einer Variablen der Klasse A gespeichert ist, kann ein Objekt jeder Klasse B sein, die von A erbt.
- Für ein in einer Variablen der Klasse A gespeichertes Objekt können alle Attribute und Methoden der Klasse A benutzt werden – und nur diese! Selbst dann, wenn es sich um ein Objekt der Klasse B handelt, das mehr Attribute (und/oder Methoden) besitzt.

# Beispiel starke Typisierung (1)

```
Person pers;  
pers = new Staff(1, "Doe", "(012) 345 678", "RA");  
System.out.println(pers.name + ": " + pers.phoneNo);
```

```
System.out.println(pers.grade);
```



- Das Objekt, das in `pers` gespeichert ist, hat zwar ein Attribut `grade`, aber die Variable `pers` hat „nur“ den Typ `Person`! Da die Klasse `Person` selbst kein Attribut `grade` besitzt, dürfen wir es nicht benutzen.
- Bei **starker Typisierung** wird die Korrektheit eines Zugriffs anhand des Variablentyps geprüft!

## Beispiel starke Typisierung (2)

```
class Inheritance {  
    public static void main(String[] args) {  
        Person[] pers = new Person[3];  
        pers[0] = new Staff(1, "Doe", "(012) 345 678", "RA");  
        pers[1] = new Student(2, "Bloggs", "(098) 4711", "0815");  
        pers[2] = new Person(5, "Citizen", "(054) 4242");  
  
        System.out.println(pers[0].grade);  
    }  
}
```

### ■ Ausgabe vom Compiler:

```
Inheritance.java:50: cannot resolve symbol  
symbol : variable grade  
location: class Person  
    System.out.println(pers[0].grade);  
                           ^  
1 error
```



# Beispiel starke Typisierung (3)

```
class Inheritance {  
  
    public static void main(String[] args) {  
        Person[] pers = new Person[3];  
        pers[0] = new Staff(1, "Doe", "(012) 345 678", "RA");  
        pers[1] = new Student(2, "Bloggs", "(098) 4711", "0815");  
        pers[2] = new Person(5, "Citizen", "(054) 4242");  
  
        System.out.println(((Staff)pers[0]).grade);  
        System.out.println(((Staff)pers[1]).grade);  
    }  
}
```

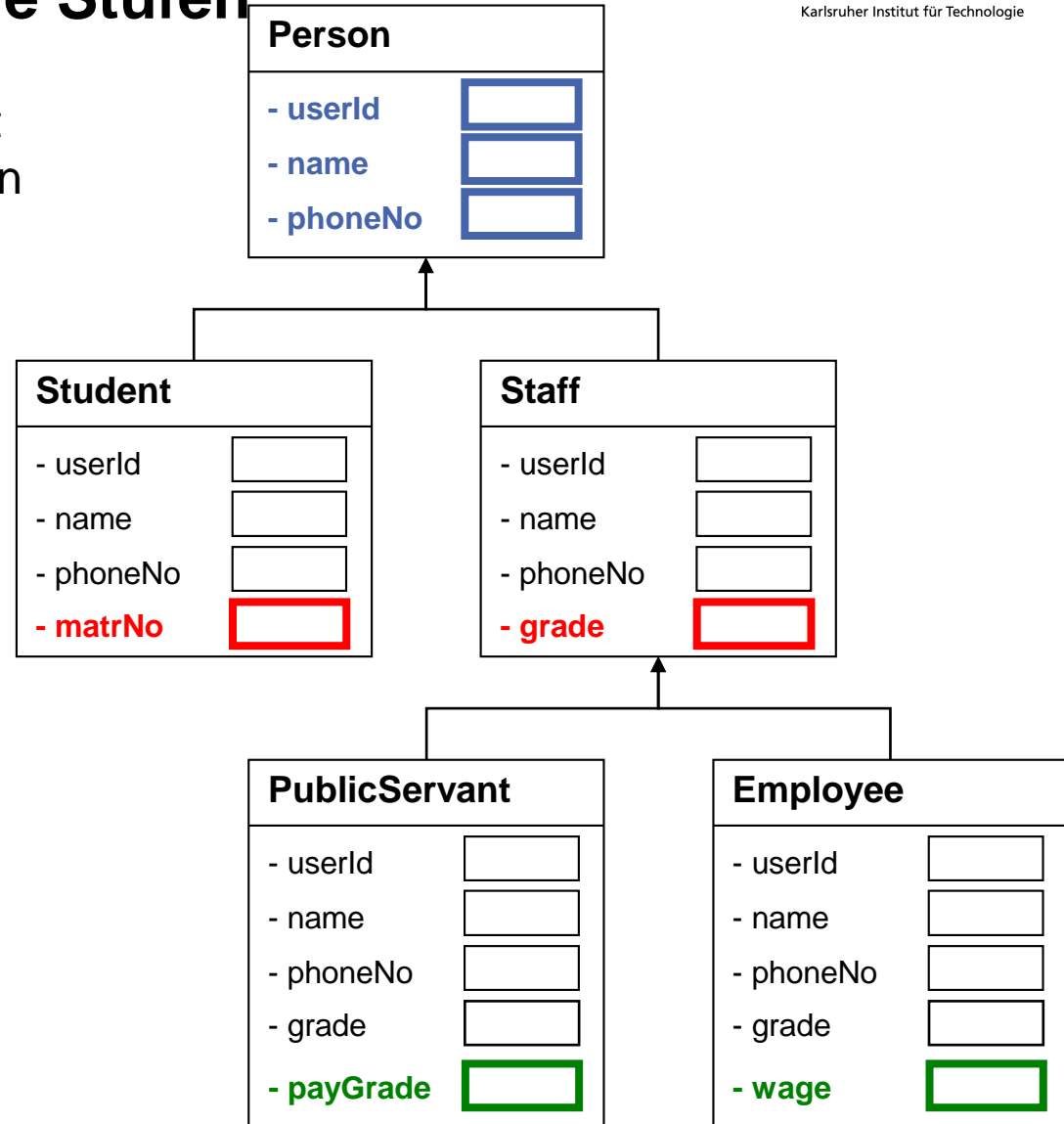
## ■ Ausgabe:

RA

Exception in thread "main" java.lang.ClassCastException: Student  
at Inheritance.main(Inheritance.java:50)

# Vererbung über mehrere Stufen

- `PublicServant` erbt indirekt über `Staff` alle Eigenschaften von `Person`
- Ein Objekt der Klasse `PublicServant` kann also sowohl einer Variablen vom Typ `Staff` als auch einer Variablen vom Typ `Person` zugewiesen werden.
- `extends` definiert eine „ist ein“-Beziehung: Ein `PublicServant` ist ein `Staff`, jedoch nicht umgekehrt.



# Vererbung von Methoden (1)


```
class Person {  
    int userId;  
    String name;  
    String phoneNo;  
  
    public Person() { }  
  
    public Person(int userId, String name, String phoneNo) {  
        /* wie gehabt */  
    }  
  
    public String toString() {  
        return name + ": " + phoneNo;  
    }  
}
```

```
public static void main(String[] args) {  
    Person pers = new Person(5, "Citizen", "(054) 4242");  
    System.out.println(pers.toString());  
    pers = new Person(6, "Lee", "(0987) 654321");  
    System.out.println(pers);  
}
```

? Ausgabe?

## Vererbung von Methoden (2)

```
Staff staff;  
staff = new Staff(1, "Doe", "(012) 345 678", "RA");  
System.out.println(staff);
```

 **Ausgabe**  
Doe: (012) 345 678

- Da `Staff` von `Person` erbt, besitzt jedes Objekt der Klasse `Staff` auch die Methode `toString()`

# Überschreiben von Methoden (1)

- Bisher: In den abgeleiteten Klassen wurden neue Attribute oder neue Methoden hinzugefügt
- Jetzt: In der abgeleiteten Klasse werden Methoden der ursprünglichen Klasse „überschrieben“

```
class Staff extends Person {  
  
    String grade;  
  
    public Staff(int userId, String name, String phoneNo, String grade) {  
        /* wie gehabt */  
    }  
  
    public String toString() {  
        return name + ": " + phoneNo + ", " + grade;  
    }  
  
}
```

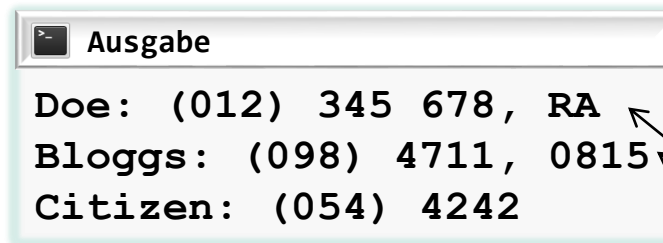
# Überschreiben von Methoden (2)

## ■ Genauso:

```
class Student extends Person {  
  
    String matrNo;  
  
    public Student(int userId, String name, String phoneNo, String matrNo) {  
        /* wie gehabt */  
    }  
  
    public String toString() {  
        return name + ": " + phoneNo + ", " + matrNo;  
    }  
  
}
```

## Überschreiben von Methoden (3)

```
Person[] pers = new Person[3];  
  
pers[0] = new Staff(1, "Doe", "(012) 345 678", "RA");  
pers[1] = new Student(2, "Bloggs", "(098) 4711", "0815");  
pers[2] = new Person(5, "Citizen", "(054) 4242");  
  
for (Person p : pers) {  
    System.out.println(p);  
}
```



```
Ausgabe  
Doe: (012) 345 678, RA  
Bloggs: (098) 4711, 0815  
Citizen: (054) 4242
```

- Hier wurden offensichtlich die neuen `toString()`-Methoden der Klassen `Staff` und `Student` aufgerufen (**Polymorphie**).

# Zusammenfassung Vererbung

- Eine Unterklassenbeziehung lässt sich explizit in der folgenden Form angeben:

```
class SubClass extends SuperClass {  
    // ...  
}
```

- Unterklasse enthält alle Komponenten der Oberklasse
- Unterklasse ist erweiterbar um neue Komponenten
- Unterklasse ist ein Spezialfall der Oberklasse
- Komponenten der Oberklasse sind von der Unterklasse aus mit dem Schlüsselwort `super` erreichbar. Dies gilt auch für den Aufruf des Konstruktors der Oberklasse.
- Klassen, für die keine Beziehung zu anderen Klassen festgelegt wurde, sind Unterklassen von `Object`



# Beispiel: Auch Chefs sind Personen

```
class Person {
    private String name;
    private String prename;
    private int staffNo;
    Person(String name, String prename, int staffNo) {
        this.name = name;
        this.prenome = prename;
        this.staffNo = staffNo;
    }
    void print(){
        System.out.println("Name: " + this.name);
        System.out.println("Prenome: " + this.prenome);
        System.out.println("Staff number: " + this.staffNo);
    }
}

class Boss extends Person {
    private String department; // zusätzliches Attribut
    Boss(String name, String prename, int staffNo, String department) {
        super(name, prename, staffNo);
        this.department = department;
    }
    void print() { // Überschreibt die Methode print() in Person
        super.print();
        System.out.println("Head of department: " + this.department);
    }
}
```

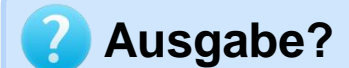
# Beispiel: Verwendung der Klassen Person und Boss

```
class Company {  
  
    public static void main(String[] args) {  
        Person[] staff = new Person[4];  
        staff[0] = new Person("McCoy", "Leonard", 987);  
        staff[1] = new Person("Scott", "Montgomery ", 654);  
        staff[2] = new Person("Uhura", "Nyota", 321);  
        staff[3] = new Boss("Kirk", "James T.", 123, "USS Enterprise");  
  
        for (Person s : staff) {  
            s.print(); // Polymorphie  
            System.out.println("---");  
        }  
    }  
}
```

? Ausgabe?

# Beispiel: Fahrzeug (Automotive)

```
class Automotive {  
    private String brand;  
    private String model;  
    private float price;  
    private float weight;  
  
    Automotive(String br, String mo, float pr, float we) {  
        this.brand = br;  
        this.model = mo;  
        this.price = pr;  
        this.weight = we;  
    }  
  
    public String toString() {  
        return "brand " + brand + ", model " + model + ", weight " + weight + "kg";  
    }  
  
    public static void main(String[] args) {  
        Automotive ente = new Automotive("Citroen", "2CV", 8399.9f, 680.5f);  
        System.out.println("My automotive: " + ente);  
    }  
}
```



# Beispiel: Erweiterung der Klasse Automotive

```
class Car extends Automotive {  
    private String fuel;  
    private int horsepower;  
  
    Car(String br, String mo, float pr, float we, String fu, int hp) {  
        super(br, mo, pr, we);  
        this.fuel = fu;  
        this.horsePower = hp;  
    }  
  
    public String toString() {  
        return super.toString() + ", fueled by " + fuel  
            + ", " + horsepower + " HP";  
    }  
  
    public static void main(String[] args) {  
        Car ente = new Car("Citroen", "2CV", 8399.9f, 680.5f, "regular gas", 28);  
        System.out.println("My car: " + ente);  
    }  
}
```

? Ausgabe?

# ÜBUNG

# Oberklasse Object

## Constructor Summary

### Constructors

#### Constructor and Description

`Object()`

## Method Summary

### All Methods

### Instance Methods

### Concrete Methods

#### Modifier and Type

#### Method and Description

protected `Object`

`clone()`

Creates and returns a copy of this object.

boolean

`equals(Object obj)`

Indicates whether some other object is "equal to" this one.

protected void

`finalize()`

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

`Class<?>`

`getClass()`

Returns the runtime class of this `Object`.

int

`hashCode()`

Returns a hash code value for the object.

void

`notify()`

Wakes up a single thread that is waiting on this object's monitor.

void

`notifyAll()`

Wakes up all threads that are waiting on this object's monitor.

`String`

`toString()`

Returns a string representation of the object.

void

`wait()`

Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.

void

`wait(long timeout)`

Causes the current thread to wait until either another thread invokes the `notify()` method or the `notifyAll()` method for this object, or a specified amount of time has elapsed.

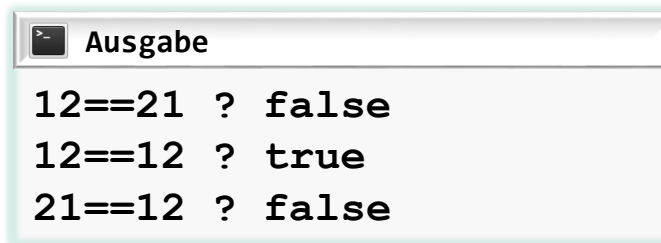


**Heusch 13.7**  
**Ratz 9.4**

# Identität und Vergleich von Objekten (1)

## ■ Vergleich von primitiven Datentypen:

```
int i1 = 12;  
int i2 = 21;  
int i3 = 12;  
System.out.println(i1 + "==" + i2 + " ? " + (i1 == i2));  
System.out.println(i1 + "==" + i3 + " ? " + (i1 == i3));  
System.out.println(i2 + "==" + i3 + " ? " + (i2 == i3));
```



```
Ausgabe  
12==21 ? false  
12==12 ? true  
21==12 ? false
```

## Vergleich von Objekten (2)

- Eine primitive Klasse `Fraction`:

```
class Fraction {  
  
    int numerator;  
    int denominator;  
  
    Fraction(int numerator, int denominator) {  
        this.numerator = numerator;  
        this.denominator = denominator;  
    }  
  
    @Override  
    public String toString() {  
        return numerator + "/" + denominator;  
    }  
}
```



# Vergleich von Objekten (3)

## ■ Vergleich von Objekten:

```
public static void main(String[] args) {  
    Fraction f1 = new Fraction(13,3);  
    Fraction f2 = new Fraction(3,13);  
    Fraction f3 = new Fraction(13,3);  
    Fraction f4 = f2;  
    System.out.println(f1+"==" + f2+ " ? " + (f1==f2));  
    System.out.println(f1+"==" + f3+ " ? " + (f1==f3));  
    System.out.println(f1+"==" + f4+ " ? " + (f1==f4));  
    System.out.println(f2+"==" + f4+ " ? " + (f2==f4));  
    System.out.println();  
    System.out.println(f1+".equals("+f2+") ? "+(f1.equals(f2)));  
    System.out.println(f1+".equals("+f3+") ? "+(f1.equals(f3)));  
    System.out.println(f1+".equals("+f4+") ? "+(f1.equals(f4)));  
    System.out.println(f2+".equals("+f4+") ? "+(f2.equals(f4)));  
}
```

? Ausgabe?

## Vergleich von Objekten (4)

Ausgabe

`13/3==3/13 ? false`

`13/3==13/3 ? false`

`13/3==3/13 ? false`

`3/13==3/13 ? true`

`13/3.equals(3/13) ? false`

`13/3.equals(13/3) ? false`

`13/3.equals(3/13) ? false`

`3/13.equals(3/13) ? true`

## Vergleich von Objekten (5) mit `.equals`

- Die neue Methode `equals` in der Klasse `Fraction` soll genau dann `true` liefern, wenn beide Objekte Brüche sind und sowohl Zähler als auch Nenner gleich sind.  
(Es muss sich nicht um dieselbe Instanz handeln!)

```
public boolean equals(Object o) {  
    if (o instanceof Fraction){  
        Fraction f = (Fraction)o;  
        return this.numerator==f.numerator && this.denominator==f.denominator;  
    } else {  
        return false;  
    }  
}
```

- Diese Methode überschreibt `.equals` aus der Klasse `Object`
- Mit `instanceof` lässt sich die Klasse eines Objektes bestimmen

## Vergleich von Objekten (6) mit `.equals`

- Für die geänderte Klasse `Fraction` mit neuem `.equals` ändert sich die Ausgabe des vorigen Programms:

Ausgabe

```
12==21 ? false
12==12 ? true
21==12 ? false

13/3==3/13 ? false
13/3==13/3 ? false # == überprüft Identität
13/3==3/13 ? false
3/13==3/13 ? true  # == überprüft Identität

13/3.equals(3/13) ? false
13/3.equals(13/3) ? true  # equals wie implementiert
13/3.equals(3/13) ? false
3/13.equals(3/13) ? true  # equals wie implementiert
```