

# Lektion 15

Objektorientierung

Explizite Vererbung (Spezialisierung, Generalisierung)

Abstrakte Klassen

Enumerations

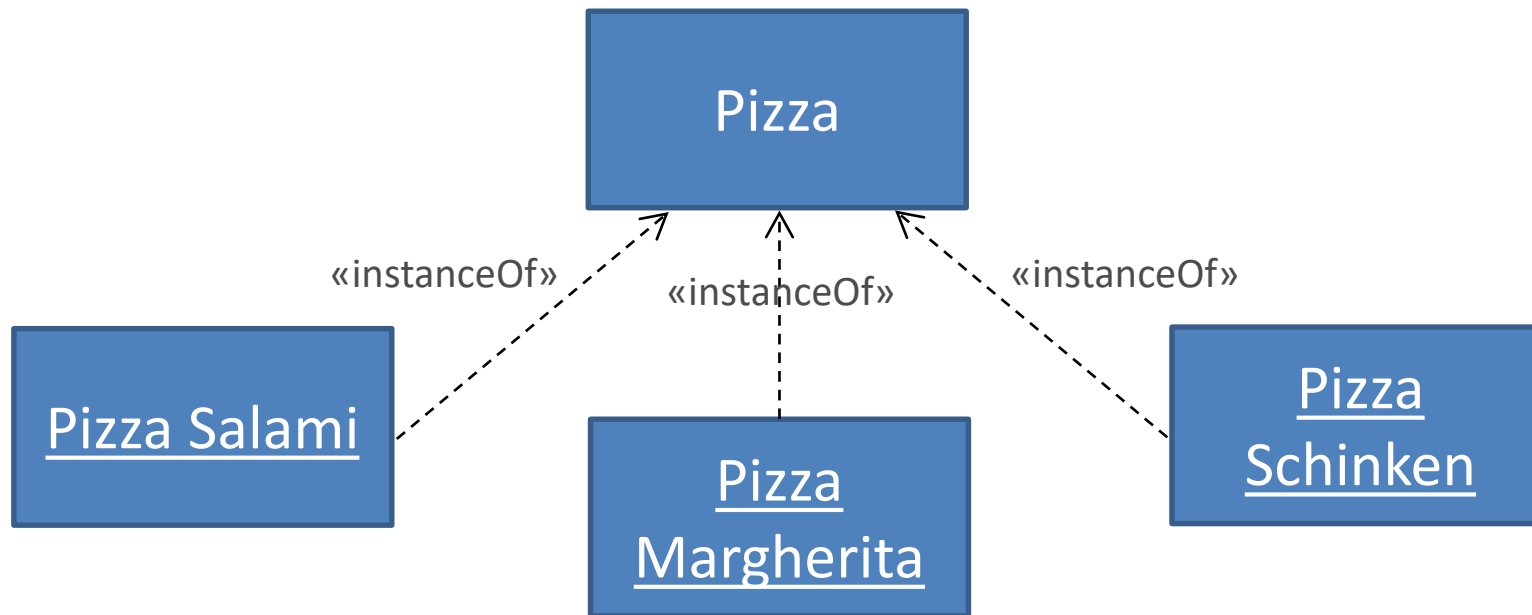
# Vererbung

Wir erinnern uns...

Um gleichartige Objekte der realen Welt zu beschreiben, bietet sich eine Abstraktion an...

...eine sogenannte **Klasse**.

Klassen dienen als Baupläne/Schablonen, nach denen **Objekte/Instanzen** erstellt werden können.

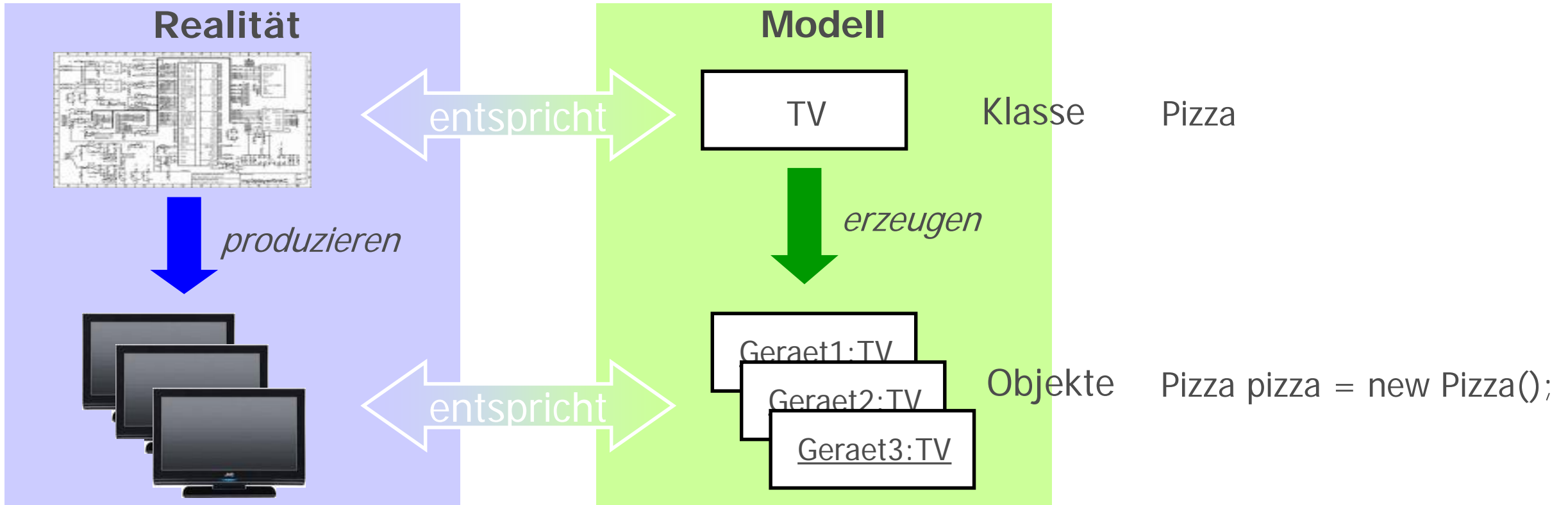


Der Vorgang, Objekte aus der Realität  
(und deren Beziehungen untereinander)  
darzustellen, heißt **Modellierung**.

Das Modell betrachtet i.d.R. nur einen  
wichtigen Ausschnitt der Realität.

➤ unwichtige Details werden weggelassen

Ein Objekt wird nach dem „Bauplan“  
einer Klasse gebaut.



Ein Beispiel...



## Realität



Aston Martin DB5

◀ fährt

James Bond

jagt ▶

Auric Goldfinger

## Modell



Beißer

Ernst Stavro Blofeld

Dr No

Auric Goldfinger

Elektra King

Odd Job

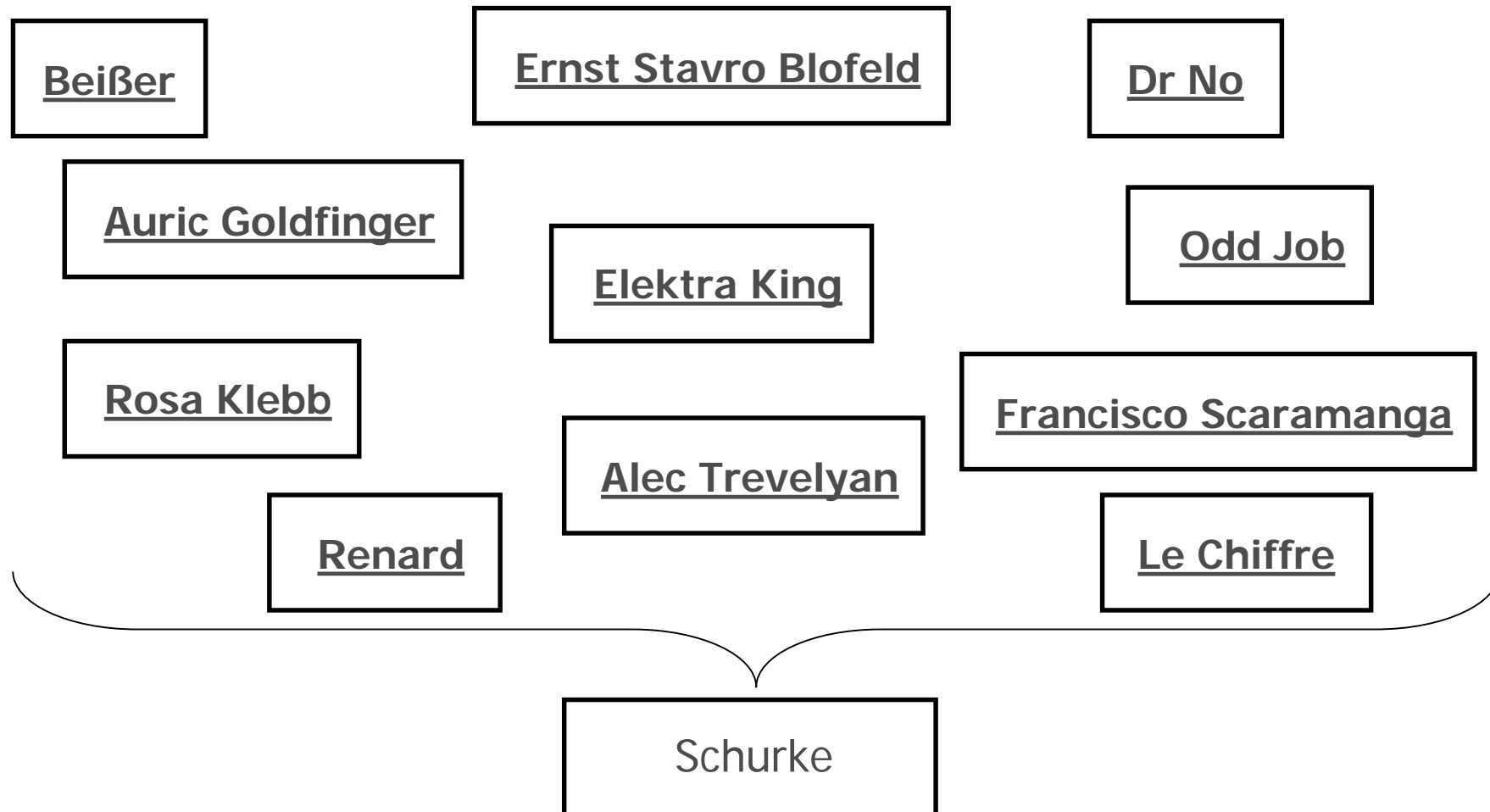
Rosa Klebb

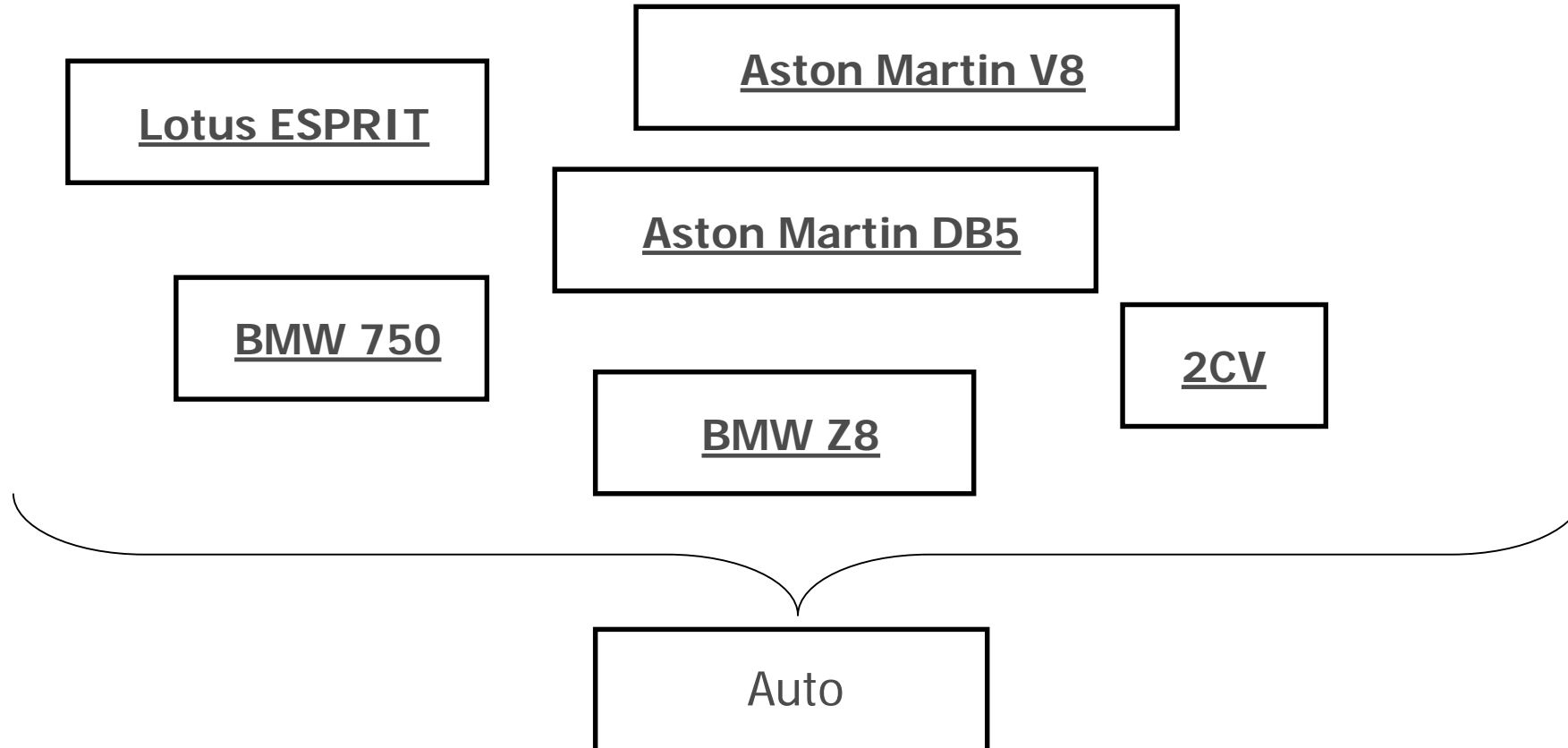
Francisco Scaramanga

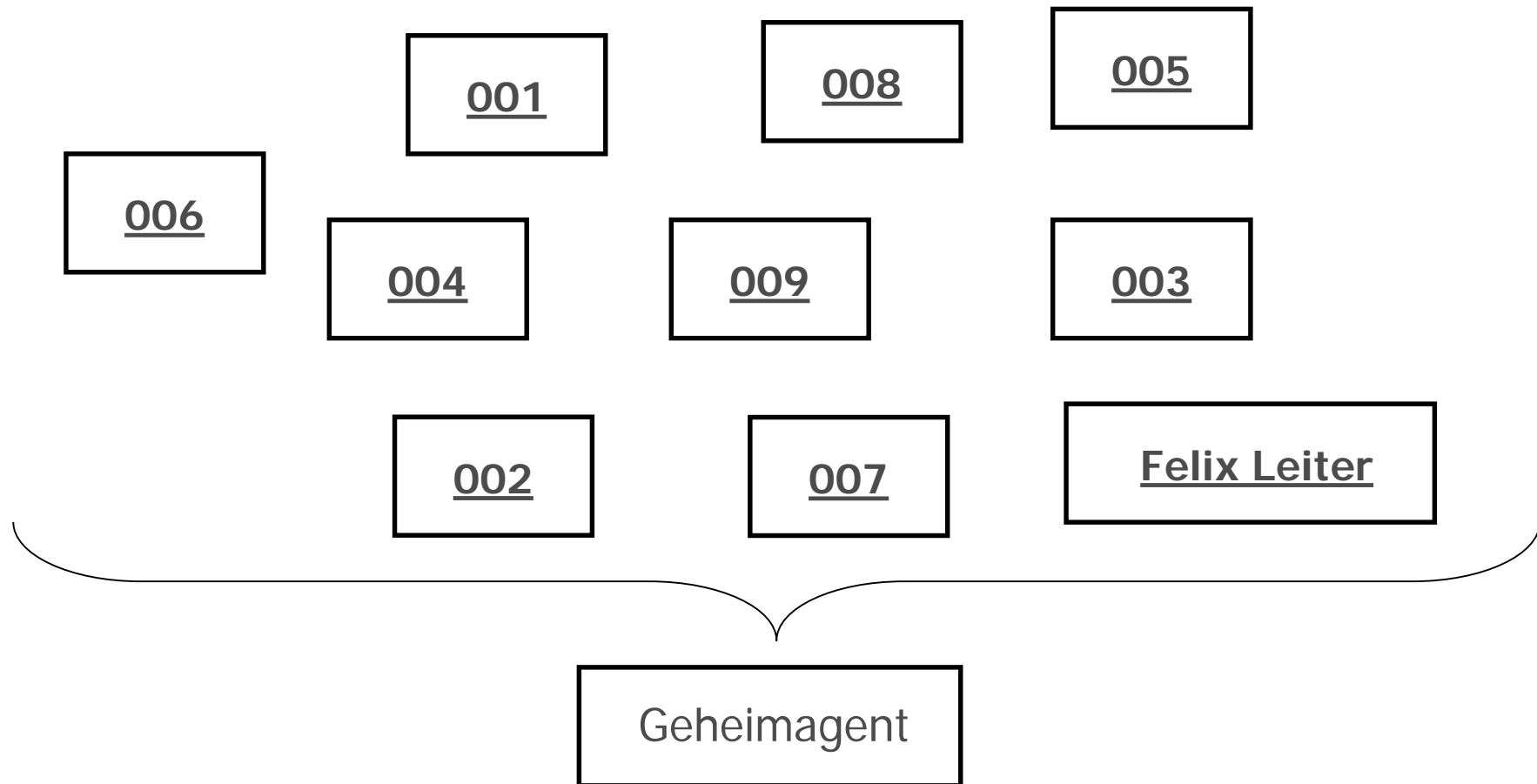
Renard

Alec Trevelyan

Le Chiffre







## Eine abstrahierte Darstellung:



Wir sehen hier eine **Assoziation** zwischen den Klassen

- Geheimagent und Auto
- Geheimagent und Schurke

Eine Assoziation repräsentiert eine (semantische, evt. strukturelle) Beziehung zwischen Klassen.

Oft haben verschiedene Klassen  
**gleiche** Eigenschaften.

```
public class Geheimagent
{
    String name;
}
```

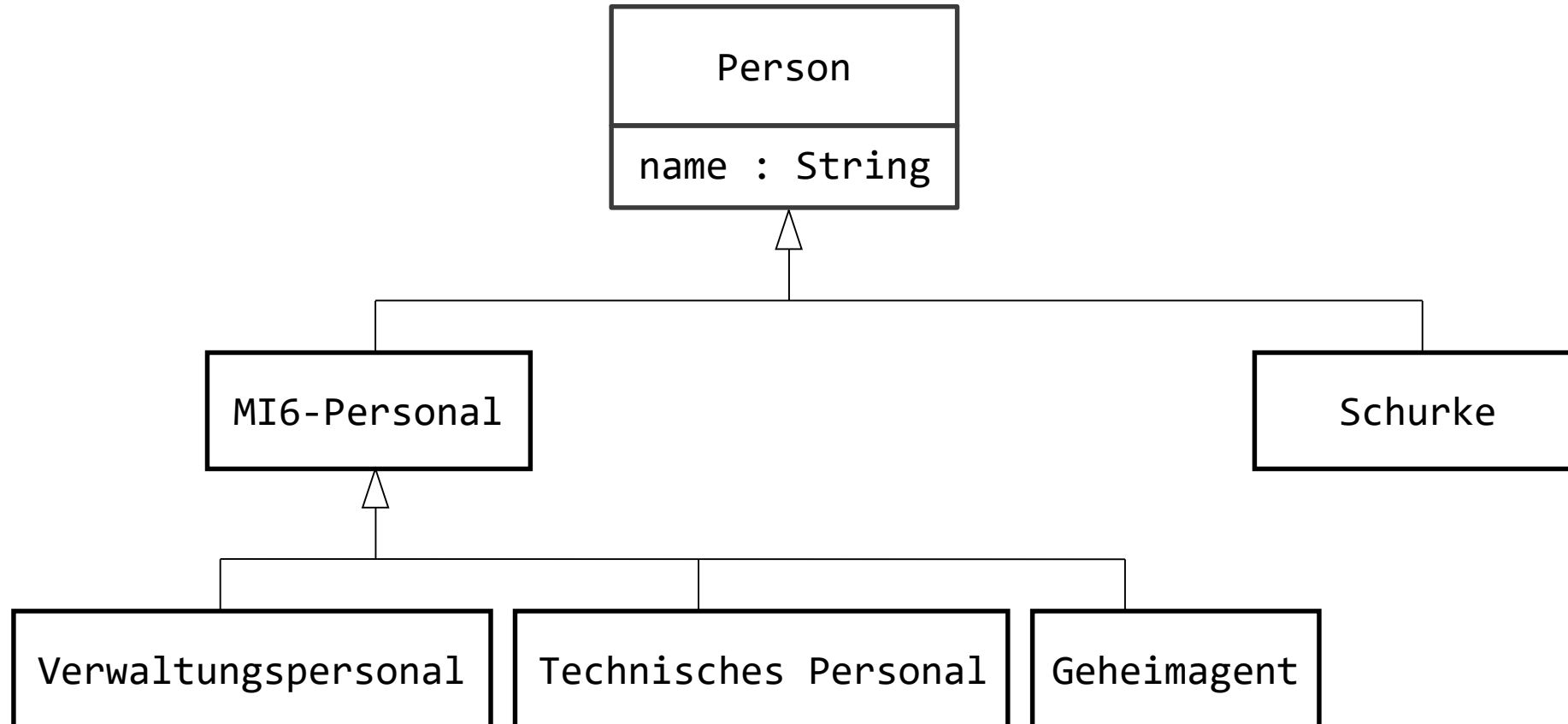
```
public class Schurke
{
    String name;
}
```



Um doppelten Code zu vermeiden  
(Don't-Repeat-Yourself (DRY)-Prinzip),  
kann man die gemeinsamen Eigenschaften  
in eine eigene Klasse auslagern.

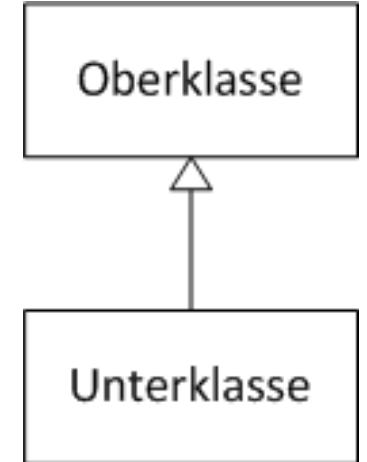


Wie in der Biologie können Eigenschaften vererbt werden,  
bspw. das Attribut **name**.

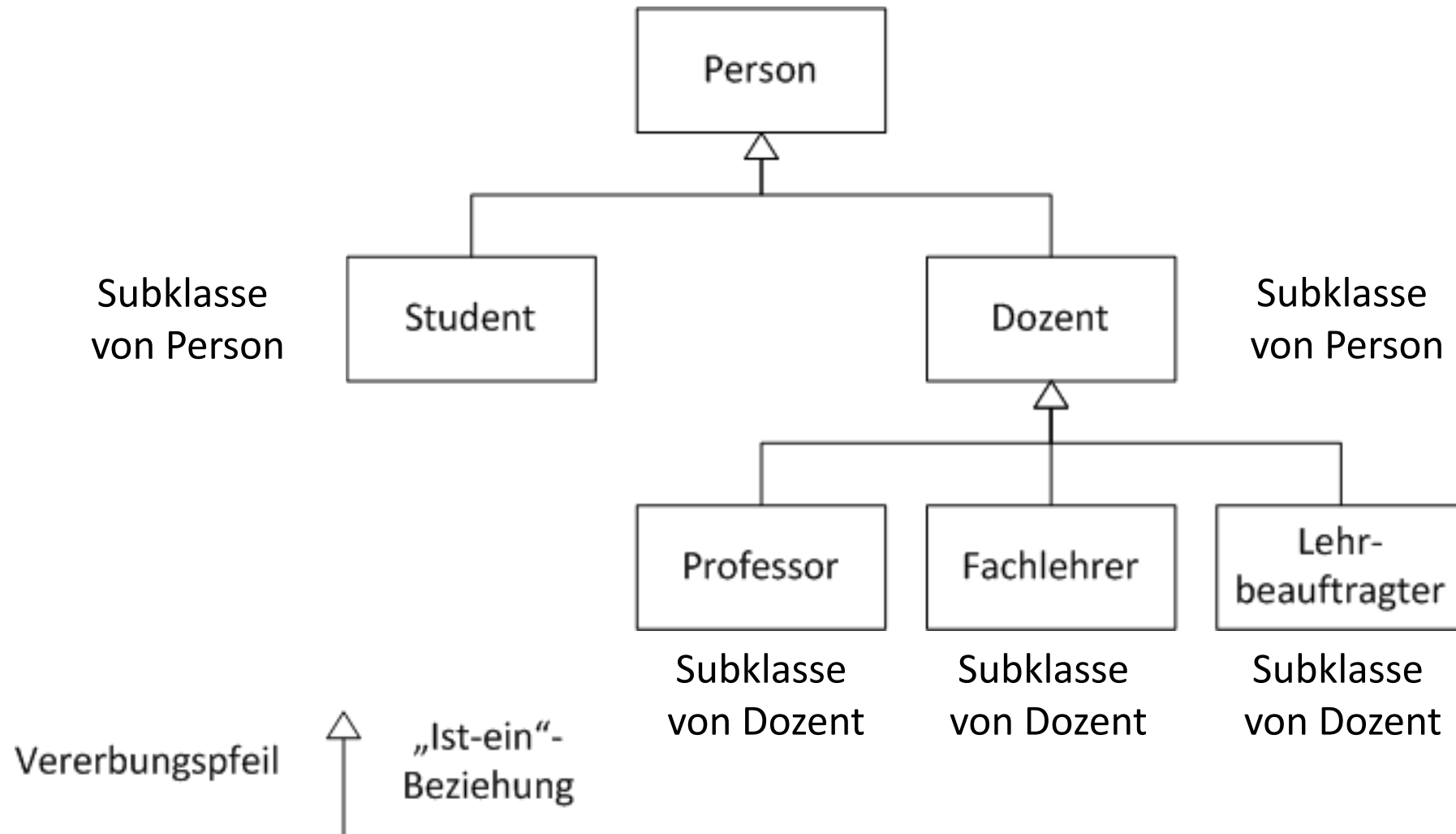


# Beziehungen zwischen Klassen: Vererbung

- Klassen können in Hierarchien angeordnet werden. Eine Oberklasse vererbt **alle Methoden und Attribute** an die Unterklasse.
- Eine Unterklasse kann maximal von **einer** Oberklasse explizit erben (d.h. eine explizite Oberklasse haben).
- Eine Unterklasse **spezialisiert** dabei die Oberklasse oder
- eine Oberklasse **generalisiert** (verallgemeinert) die Eigenschaften (einer oder) mehrerer Unterklassen.



# Beispiel: Vererbung

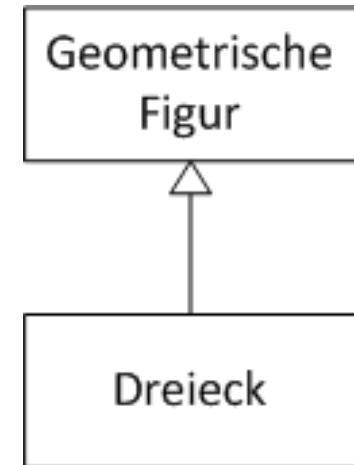


# (Explizite) Vererbung in Java

Wir haben eine geometrische Figur, die eine Methode an eine Klasse Dreieck vererben will.

Zunächst sollten wir die “**Ist-Ein/e**”-Beziehung überprüfen!

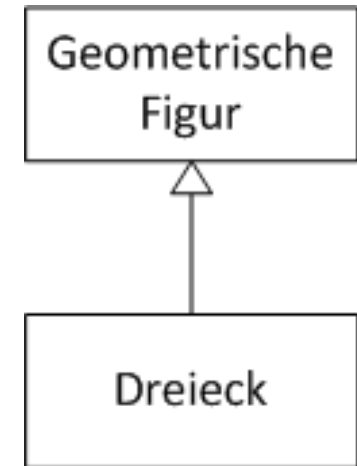
Ein Dreieck **ist eine** geometrische Figur. ✓



# (Explizite) Vererbung in Java

Durch das Schlüsselwort **extends** wird in Java eine (explizite) Vererbungsbeziehung ausgedrückt:

```
public class GeometrischeFigur {  
    public double berechneFlaeche() {  
        ...  
    }  
}  
  
public class Dreieck extends GeometrischeFigur {  
  
}  
  
public class Main {  
    public static void main(String[] args) {  
        Dreieck d = new Dreieck();  
        System.out.println(d.berechneFlaeche());  
    }  
}
```



Die Methode  
`berechneFlaeche()` ist jetzt  
auch im Dreieck verfügbar.

# Vererbung: Spezialisierung

## Gegeben sei folgende Rechteck-Klasse!

```
public class Rechteck
{
    double laenge;
    double breite;

    public Rechteck() {

    }

    public Rechteck(double laenge, double breite)
    {
        this.laenge = laenge;
        this.breite = breite;
    }

    public double berechneFlaeche()
    {
        return laenge * breite;
    }
}
```

Wir wissen bei dem  
Rechteck, wie die Fläche  
berechnet wird.

Kann dieses Wissen auch  
für die Volumenberechnung  
eines Quaders genutzt  
werden?

# Entwicklung einer Quader-Klasse

## Normale Vorgehensweise

```
public class Quader
{
    double laenge;
    double breite;
    double tiefe;

    public Quader(double laenge,
                  double breite, double tiefe)
    {
        this.laenge = laenge;
        this.breite = breite;
        this.tiefe = tiefe;
    }

    public double berechneVolumen()
    {
        return laenge*breite*tiefe;
    }
}
```

```
public class Main
{
    public static void main(String[] args)
    {
        Quader q = new Quader(10, 5, 5);
        System.out.println(q.berechneVolumen());
    }
}
```



# Spezialisierung: Die Unterklasse Quader spezialisiert die Oberklasse Rechteck

```
public class Rechteck
{
    double laenge;
    double breite;

    public Rechteck() {
    }

    public Rechteck(double laenge,
                    double breite) {
        this.laenge = laenge;
        this.breite = breite;
    }

    public double berechneFlaeche() {
        return laenge * breite;
    }
}
```

```
public class Quader extends Rechteck
{
    double tiefe;

    public Quader(double laenge,
                  double breite, double tiefe)
    {
        this.laenge = laenge;
        this.breite = breite;
        this.tiefe = tiefe;
    }

    public double berechneVolumen()
    {
        return berechneFlaeche()*tiefe;
    }
}
```

von Rechteck geerbt

# Vergleich zwischen normalem Quader und spezialisiertem Quader

```
public class Quader {  
    double laenge;  
    double breite;  
    double tiefe;  
  
    public Quader(double laenge,  
        double breite, double tiefe) {  
        this.laenge = laenge;  
        this.breite = breite;  
        this.tiefe = tiefe;  
    }  
  
    public double berechneVolumen() {  
        return laenge*breite*tiefe;  
    }  
}
```

Die Schnittstelle nach außen bleibt bestehen:

```
public class Main {  
    public static void main(String[] args) {  
        Quader q = new Quader(10, 5, 5);  
        System.out.println(q.berechneVolumen());  
    }  
}
```

```
public class Quader extends Rechteck {  
  
    double tiefe;  
  
    public Quader(double laenge,  
        double breite, double tiefe) {  
        this.laenge = laenge;  
        this.breite = breite;  
        this.tiefe = tiefe;  
    }  
  
    public double berechneVolumen() {  
        return berechneFlaeche()*tiefe;  
    }  
}
```

Das Hauptprogramm funktioniert mit beiden Implementierungen.

Was fällt ansonsten auf?

# Spezialisierung: Die Unterklasse Quader spezialisiert die Oberklasse Rechteck

```
public class Rechteck
{
    double laenge;
    double breite;

    public Rechteck() {
    }

    public Rechteck(double laenge, double breite) {
        this.laenge = laenge;
        this.breite = breite;
    }

    public double berechneFlaeche() {
        return laenge * breite;
    }
}
```

```
public class Quader extends Rechteck {

    double tiefe;

    public Quader(double laenge,
        double breite, double tiefe) {
        this.laenge = laenge;
        this.breite = breite;
        this.tiefe = tiefe;
    }

    public double berechneVolumen() {
        return berechneFlaeche()*tiefe;
    }
}
```

**Doppelter Code.**

Kann auch der Konstruktor der Oberklasse wiederverwendet werden?



# super

## Zugriff auf Methoden und Konstruktoren der Oberklasse

- Mit dem Schlüsselwort `super` kann man auf die Attribute, **Methoden** und **Konstruktoren** der Oberklasse zugreifen.

```
public class Rechteck
{
    double laenge;
    double breite;
```

```
    public Rechteck(double laenge,
                     double breite) {
        this.laenge = laenge;
        this.breite = breite;
    }
```

```
    public double berechneFlaeche() {
        return laenge*breite;
    }
}
```

```
public class Quader extends Rechteck
{
    double tiefe;           ruft Konstruktor der
                           Oberklasse auf
```

```
    public Quader(double laenge, double
                     breite, double tiefe) {
        super(laenge, breite);
        this.tiefe = tiefe;
    }
```

```
    public double berechneVolumen()
    {
        return super.berechneFlaeche()*tiefe;
    }
}
```

ruft Methode der  
Oberklasse auf

# super

## Zugriff auf Attribute der Oberklasse

```
public class Rechteck
{
    private double laenge;
    private double breite;

    public Rechteck(double laenge,
        double breite) {
        this.laenge = laenge;
        this.breite = breite;
    }

    public double berechneFlaeche() {
        double area;
        area = laenge*breite;
        return area;
    }
}
```

```
public class Quader extends Rechteck
{
    double tiefe;

    public Quader(double laenge, double
        breite, double tiefe)
    {
        this.laenge = laenge;
        this.breite = breite;
        this.tiefe = tiefe;
    }
    ...
}
```

**Zugriff ist nicht erlaubt,  
da private!**



# super

## Zugriff auf Attribute der Oberklasse

```
public class Rechteck
{
    double laenge;
    double breite;

    public Rechteck(double laenge,
        double breite) {
        this.laenge = laenge;
        this.breite = breite;
    }

    public double berechneFlaeche() {
        double area;
        area = laenge*breite;
        return area;
    }
}
```

```
public class Quader extends Rechteck
{
    double tiefe;

    public Quader(double laenge, double
        breite, double tiefe)
    {
        this.laenge = laenge;
        this.breite = breite;
        this.tiefe = tiefe;
    }
    ...
}
```

**Zugriff ist erlaubt, wenn beide  
Klassen im selben Package sind**

# super

## Zugriff auf Attribute der Oberklasse

- Damit eine Oberklasse Unterklassen einen direkten Zugriff auf ihre **Attribute** erlaubt, muss sie diese als **protected** deklarieren.

```
public class Rechteck
{
    protected double laenge;
    protected double breite;
```

```
    public Rechteck(double laenge,
        double breite) {
        this.laenge = laenge;
        this.breite = breite;
    }
```

```
    public double berechneFlaeche() {
        double area;
        area = laenge*breite;
        return area;
    }
}
```

```
public class Quader extends Rechteck
{
    double tiefe;
```

```
    public Quader(double laenge, double
        breite, double tiefe)
```

```
    {
        this.laenge = laenge;
        this.breite = breite;
        this.tiefe = tiefe;
```

```
    }
    ...
}
```

**Zugriff ist erlaubt, da Quader  
Unterklasse von Rechteck**



# Zugriffsschutz durch Modifier

`private` < *default* < `protected` < `public`

- `private`
  - nur Zugriff aus gleicher Klasse
- *default* (*package*)
  - Beinhaltet `private`. Zusätzlich Zugriff aus Klassen des gleichen Packages (wenn kein Modifier angegeben).
- `protected`
  - Beinhaltet *default*. Zusätzlich Zugriff aus Unterklassen.
- `public`
  - uneingeschränkter Zugriff



# super

## Konstruktoraufruf der Oberklasse

- Wird im Konstruktor weder `this` noch `super` für einen Konstruktoraufruf verwendet, wird `super()` beim Übersetzen als erste Anweisung im Konstruktor ergänzt.

```
public class A {  
    public A() {  
        System.out.println("A: Created A");  
    }  
}
```

**wird automatisch vom Compiler ergänzt  
(Übrigens auch bei `public A()`)**

```
public class B extends A {  
    public B(String test) {  
        super();  
        System.out.println("B: " + test);  
    }  
    public static void main(String[] args) {  
        B b = new B("!");  
    }  
}
```

- Der Aufruf eines Konstruktors muss immer die erste Anweisung im Konstruktor sein (sowohl bei `super` als auch bei `this`).

# Generalisierung

Häufig kommt es vor, dass verschiedene Klassen ähnliche Eigenschaften haben.

Beispiel: geometrische Figuren

Nehmen wir an, wir wollen ein Programm schreiben, das für verschiedene geometrische Figuren die Fläche berechnen kann!

D.h. es gibt verschiedene aus der Realität abzubildende Objekte:  
Dreieck, Rechteck, Quadrat

# Dreieck

```
public class Dreieck
{
    double grundseite;
    double hoehe;

    public Dreieck(double grundseite, double hoehe)
    {
        this.grundseite = grundseite;
        this.hoehe = hoehe;
    }

    public double berechneFlaeche()
    {
        return 0.5 * grundseite * hoehe;
    }
}
```

# Quadrat

```
public class Quadrat
{
    double seitenlaenge;

    public Quadrat(double seitenlaenge)
    {
        this.seitenlaenge = seitenlaenge;
    }

    public double berechneFlaeche()
    {
        return seitenlaenge*seitenlaenge;
    }
}
```

# Rechteck

```
public class Rechteck
{
    double laenge;
    double breite;

    public Rechteck(double laenge, double breite)
    {
        this.laenge = laenge;
        this.breite = breite;
    }

    public double berechneFlaeche()
    {
        return laenge * breite;
    }
}
```

Welche Gemeinsamkeiten  
fallen auf?

# Rechteck

```
public class Rechteck
{
    double laenge;
    double breite;

    public Rechteck(double laenge, double breite)
    {
        this.laenge = laenge;
        this.breite = breite;
    }

    public double berechneFlaeche()
    {
        return laenge * breite;
    }
}
```

Welche Gemeinsamkeiten  
fallen auf?

Alle drei geometrischen  
Figuren haben eine  
Methode **berechneFlaeche**.

Kann man diese Eigenschaft  
verallgemeinern?

Wir erzeugen eine Oberklasse Geometrische Figur, die die Gemeinsamkeiten kapselt.

```
public class GeometrischeFigur {  
    public double berechneFlaeche() {  
        return -1;  
    }  
}
```

eine sinnvolle  
Flächenberechnung können  
wir noch nicht vorgeben,  
daher geben wir -1 zurück.

Wir erzeugen eine Oberklasse Geometrische Figur, die die Gemeinsamkeiten kapselt.

```
public class GeometrischeFigur {  
    public double berechneFlaeche() {  
        throw new UnsupportedOperationException();    oder werfen eine Exception  
    }  
}
```

Wir erzeugen eine Oberklasse Geometrische Figur, die die Gemeinsamkeiten kapselt.

```
public class GeometrischeFigur {  
    public double berechneFlaeche() {  
        throw new UnsupportedOperationException();    oder werfen eine Exception  
    }  
}
```


```
public class Dreieck extends GeometrischeFigur  
{
```

```
    double hoehe;  
    double grundseite;
```

```
@Override
```

```
    public double berechneFlaeche() {  
        return 0.5 * grundseite * hoehe;  
    }  
}
```

**@Override** ist eine sog. Annotation und **kann** benutzt werden, um zu zeigen, dass eine geerbte Methode (einer Oberklasse) **überschrieben** wird.





# Dreieck

```
public class Dreieck extends GeometrischeFigur
{
    double grundseite;
    double hoehe;

    public Dreieck(double grundseite, double hoehe)
    {
        this.grundseite = grundseite;
        this.hoehe = hoehe;
    }

    @Override
    public double berechneFlaeche()
    {
        return 0.5 * grundseite * hoehe;
    }
}
```

# Quadrat

```
public class Quadrat extends GeometrischeFigur
{
    double seitenlaenge;

    public Quadrat(double seitenlaenge)
    {
        this.seitenlaenge = seitenlaenge;
    }

    @Override
    public double berechneFlaeche()
    {
        return seitenlaenge*seitenlaenge;
    }
}
```

# Rechteck

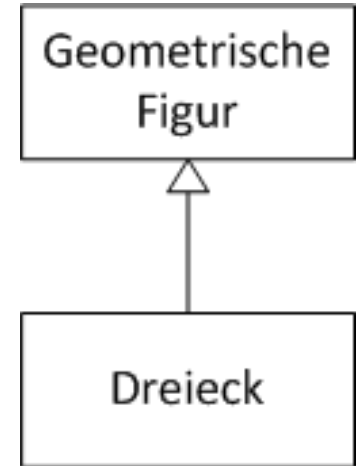
```
public class Rechteck extends GeometrischeFigur
{
    double laenge;
    double breite;

    public Rechteck(double laenge, double breite)
    {
        this.laenge = laenge;
        this.breite = breite;
    }

    @Override
    public double berechneFlaeche()
    {
        return laenge * breite;
    }
}
```

# Substitutionsprinzip

- Substitutionsprinzip: Eine Instanz einer Unterklasse ist überall dort verwendbar, wo eine Instanz der Oberklasse verwendet werden kann.
- oder: Anstelle einer Instanz einer Klasse kann auch eine Instanz von deren Unterklasse verwendet werden.
- Java unterstützt das Substitutionsprinzip. Beispiel:



```
public class Main {
    ...
    public static void main(String[] args) {
        //Vereinbarung einer Referenz auf GeometrischeFigur
        GeometrischeFigur g;

        //zulässige Zuweisung. Dreieck-Instanz substituiert GeometrischeFigur
        g = new Dreieck(10, 5); //automatischer Upcast

        //Nutzung der Referenz erfolgt so, als ob es sich um eine Instanz von GeometrischeFigur handele
        System.out.println(g.berechneFlaeche());
    }
}
```

Jetzt gibt es die Methode  
`berechneFlaeche()` mehrmals:

```
public class GeometrischeFigur {  
    ...  
    public double berechneFlaeche() {  
        ...  
    }  
}
```



```
public class Dreieck  
    extends GeometrischeFigur {  
    ...  
    public double berechneFlaeche() {  
        ...  
    }  
}
```

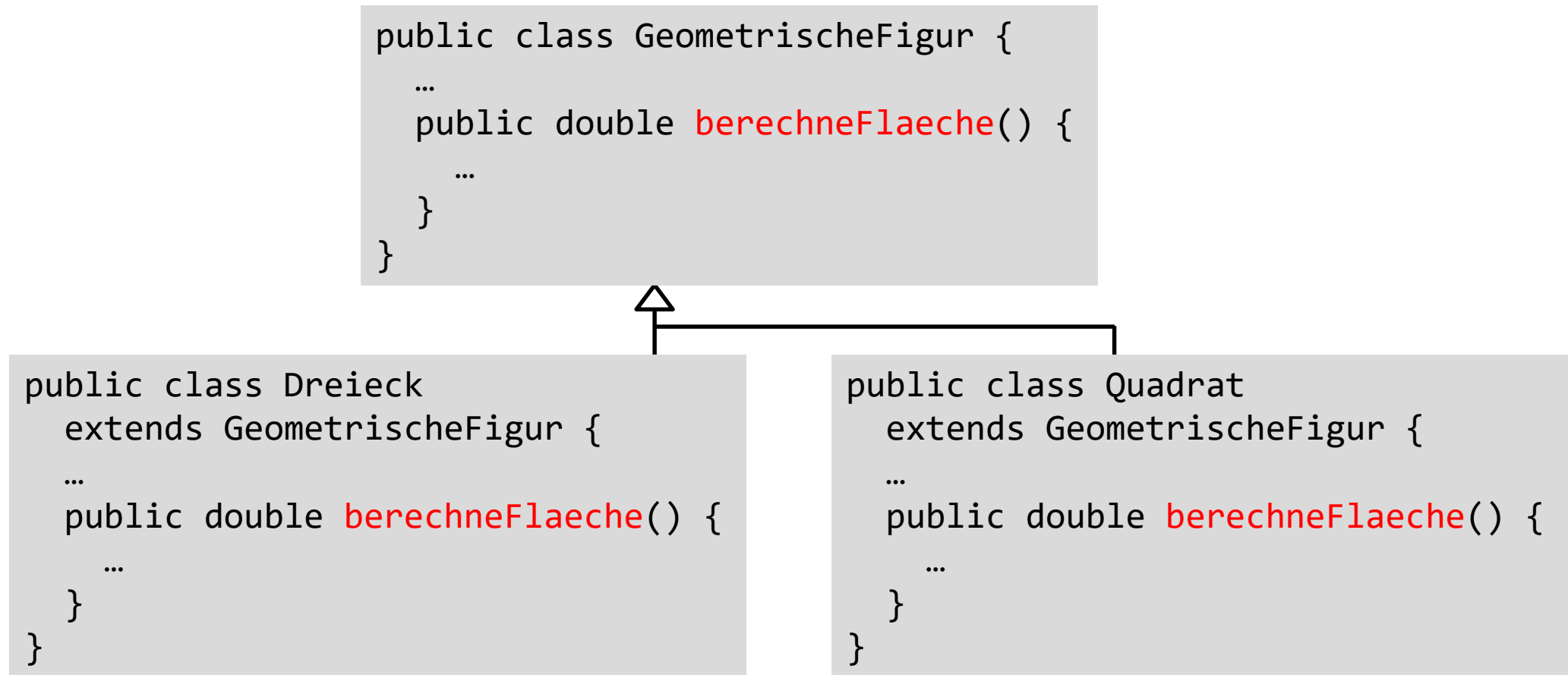
```
public class Quadrat  
    extends GeometrischeFigur {  
    ...  
    public double berechneFlaeche() {  
        ...  
    }  
}
```

Man sagt auch die Methode `berechneFlaeche` ist **polymorph** (vielgestaltig),  
d.h. sie hat verschiedene Implementierungen in der gleichen Vererbungshierarchie.

# Polymorphismus

## (Inclusion polymorphism, Enthaltender Polymorphismus)

Polymorphismus ist also das Auftreten verschiedener Implementierungen einer Methode in der gleichen Vererbungshierarchie.



## Aber:

Welche Methode wird aufgerufen?  
Das wird zur Laufzeit entschieden.

Zunächst wird bei der Instanz (im Beispiel Dreieck) nach einer Methode gesucht.



Wenn es dort die Methode nicht gibt, wird bei der nächsten Oberklasse in der Klassenhierarchie geschaut.



Wenn es dort die Methode nicht gibt, wird der Aufstieg in der Klassenhierarchie solange fortgesetzt, bis eine Implementierung gefunden wurde.



Die gefundene Implementierung wird ausgeführt, weitere Implementierungen in Oberklassen ignoriert.

Dieser Vorgang heißt **dynamisches/spätes Binden**.

# Was bringt uns das?

```
public static void main(String[] args) {  
    GeometrischeFigur[] figuren = new GeometrischeFigur[2];  
    figuren[0] = new Dreieck(2, 4);  
    figuren[1] = new Rechteck(2, 4);  
  
    for (int i = 0; i < figuren.length; i++) {  
        System.out.println(figuren[i].berechneFlaeche());  
    }  
}
```

berechneFlaeche() wird beim  
ersten Durchlauf von Dreieck,  
beim zweiten von Rechteck  
aufgerufen.

Wir können Objekte verschiedener Klassen, wenn sie  
gleiche Eigenschaften haben, gleich behandeln.

Durch die Generalisierung können die unterschiedlichen Objekte auf gleiche  
Weise verwendet und bspw. in einem Array zusammengefasst werden.

```
public class Main
{
    public static double berechneGesamtflaeche(GeometrischeFigur[] figuren)
    {
        double summe = 0;
        for (int i = 0; i < figuren.length; i++)
        {
            summe += figuren[i].berechneFlaeche();
        }
        return summe;
    }
}
```

Jede unterschiedliche  
GeometrischeFigur weiß selbst am  
besten, wie sie die Fläche  
berechnet.

```
public static void main(String[] args)
{
    GeometrischeFigur[] figuren = new GeometrischeFigur[3];
    figuren[0] = new Dreieck(10, 5);
    figuren[1] = new Dreieck(20, 10);
    figuren[2] = new Rechteck(10, 10);
    double gesamtflaeche = berechneGesamtflaeche(figuren);
    System.out.println(gesamtflaeche);
}
}
```



Wie schützt man sich davor, dass jemand  
einfach ein Objekt der Klasse  
GeometrischeFigur anlegt und deren  
Fläche berechnet?

```
public static void main(String[] args) {  
    GeometrischeFigur geo = new GeometrischeFigur();  
    System.out.println(geo.berechneFlaeche());  
}
```

-1 oder Exception

Eine allgemeine Berechnung des  
Flächeninhalts ohne Kenntnis der Figur  
ergibt keinen Sinn.

# Abstrakte Klasse

- Der Modifier **abstract** auf eine Klasse angewandt macht diese abstrakt. Eine abstrakte Klasse kann nicht instanziiert werden.
- Der Modifier **abstract** auf eine Methode angewandt macht diese abstrakt. Eine abstrakte Methode muss in einer Unterklasse implementiert werden, wenn die Klasse nicht wiederum abstrakt ist.
- Eine Klasse mit abstrakten Methoden muss selbst als abstrakt deklariert werden.

```
public abstract class GeometrischeFigur {  
    public abstract double berechneFlaeche();  
}
```

kein  
Methodenrumpf!

# Abstrakte Klasse - Beispiel

```
public abstract class GeometrischeFigur {  
    public abstract double berechneFlaeche();  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        GeometrischeFigur geo = new GeometrischeFigur();  
    }  
}
```



Cannot instantiate the type GeometrischeFigur

Eine abstrakte Klasse darf auch  
nicht-abstrakte Methoden haben.

Nehmen wir an, wir wollen die Dreiecke, Rechtecke,  
etc. als Teil eines Malprogramms nutzen.

Jede Figur soll mit einem Text beschriftet werden  
können.

Ein Editor stellt Methoden zur Verfügung, um neue  
geometrische Figuren zu erstellen und diese zu  
beschriften.

```
public abstract class GeometrischeFigur {
    String text;

    public abstract double berechneFlaeche();

    public String getText()
    {
        return text;
    }

    public void setText(String text)
    {
        this.text = text;
    }
}
```

```
public class Editor {
    public GeometrischeFigur erzeugeRechteck(int laenge,
        int breite) {
        ...
        return new Rechteck(laenge, breite);
    }

    public void beschrifte(GeometrischeFigur figur,
        String text) {
        figur.setText(text);
    }

    public static void main(String[] args) {
        Editor editor = new Editor();
        GeometrischeFigur r = editor.erzeugeRechteck(10, 5);
        editor.beschrifte(r, "Hausfront");
    }
}
```

# Überschreiben von Methoden

- Der return type der überschreibenden Methode (i.e. Methode der Unterklasse) darf vom return type der überschriebenen Methode (i.e. Methode der Oberklasse) abgeleitet sein.

```
public class A
{
    public A()
    {
        System.out.println("A erstellt");
    }

    protected A retA()
    {
        return new A();
    }
}
```

```
public class B extends A
{

    @Override
    protected B retA()
    {
        return new B();
    }
}
```

# Überschreiben von Methoden

- Der return type der überschreibenden Methode (i.e. Methode der Unterklasse) darf vom return type der überschriebenen Methode (i.e. Methode der Oberklasse) abgeleitet sein.
- Der Zugriffsschutz darf gelockert werden.

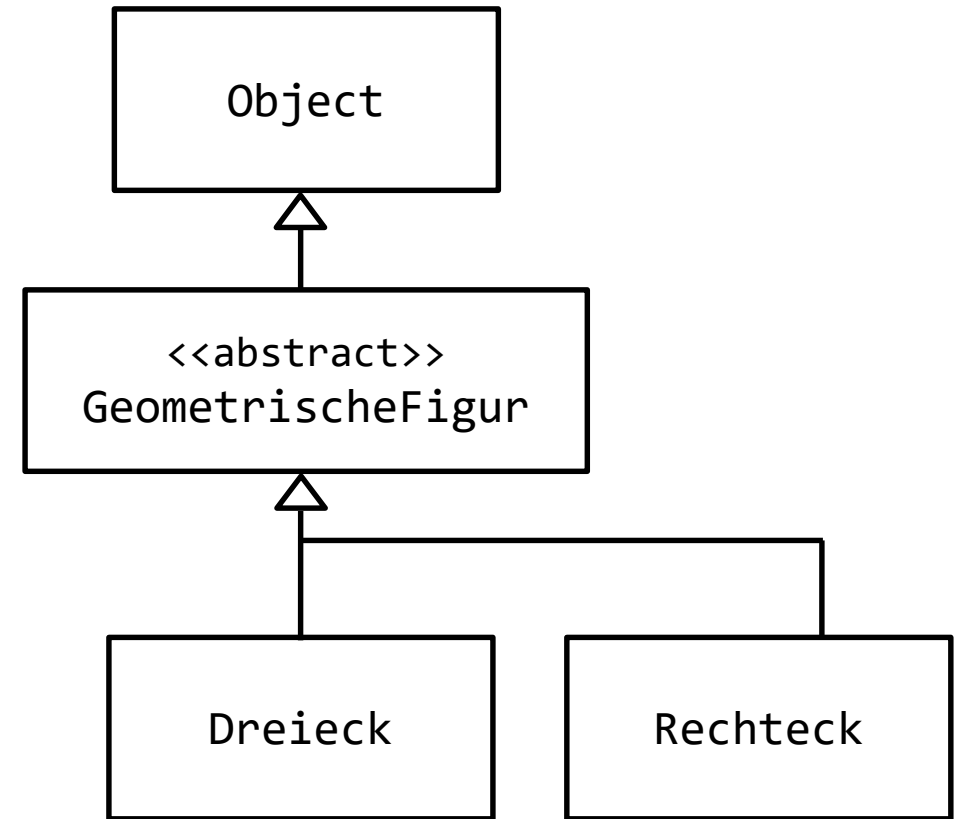
```
public class A
{
    public A()
    {
        System.out.println("A erstellt");
    }

    protected A retA()
    {
        return new A();
    }
}
```

```
public class B extends A
{

    @Override
    public B retA()
    {
        return new B();
    }
}
```

```
public class GeometrischeFigur {  
    protected GeometrischeFigur gibInstanz()  
    {  
        return this;  
    }  
  
    public static void main(String[] args) {  
        GeometrischeFigur[] figuren = new GeometrischeFigur[2];  
        figuren[0] = new Dreieck(2, 4);  
        figuren[1] = new Rechteck(2, 4);  
  
        for (int i = 0; i < figuren.length; i++) {  
            System.out.println(figuren[i].gibInstanz());  
        }  
    }  
}
```





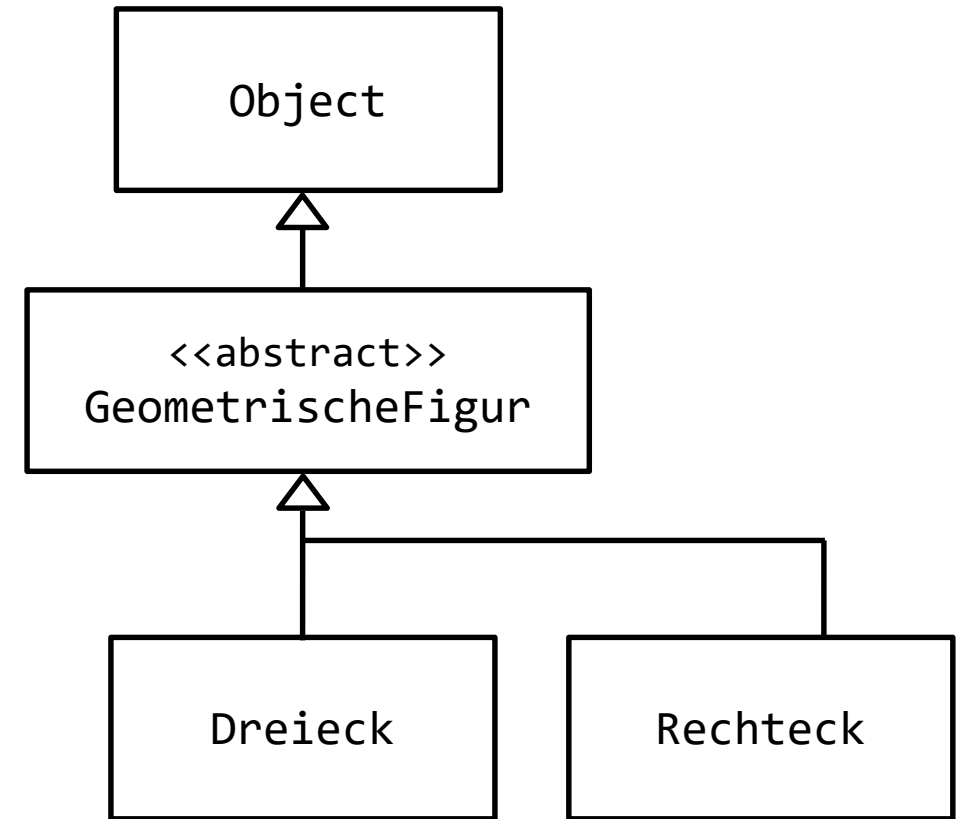
```

public class GeometrischeFigur {
    protected GeometrischeFigur gibInstanz()
    {
        return this;
    }

    public static void main(String[] args) {
        GeometrischeFigur[] figuren = new GeometrischeFigur[2];
        figuren[0] = new Dreieck(2, 4);
        figuren[1] = new Rechteck(2, 4);

        for (int i = 0; i < figuren.length; i++) {
            System.out.println(figuren[i].gibInstanz());
        }
    }
}

```



Welche Klassen wären als Rückgabetyp für gibInstanz der Klasse Dreieck denkbar?

- A: GeometrischeFigur
- B: Dreieck
- C: Object
- D: Rechteck

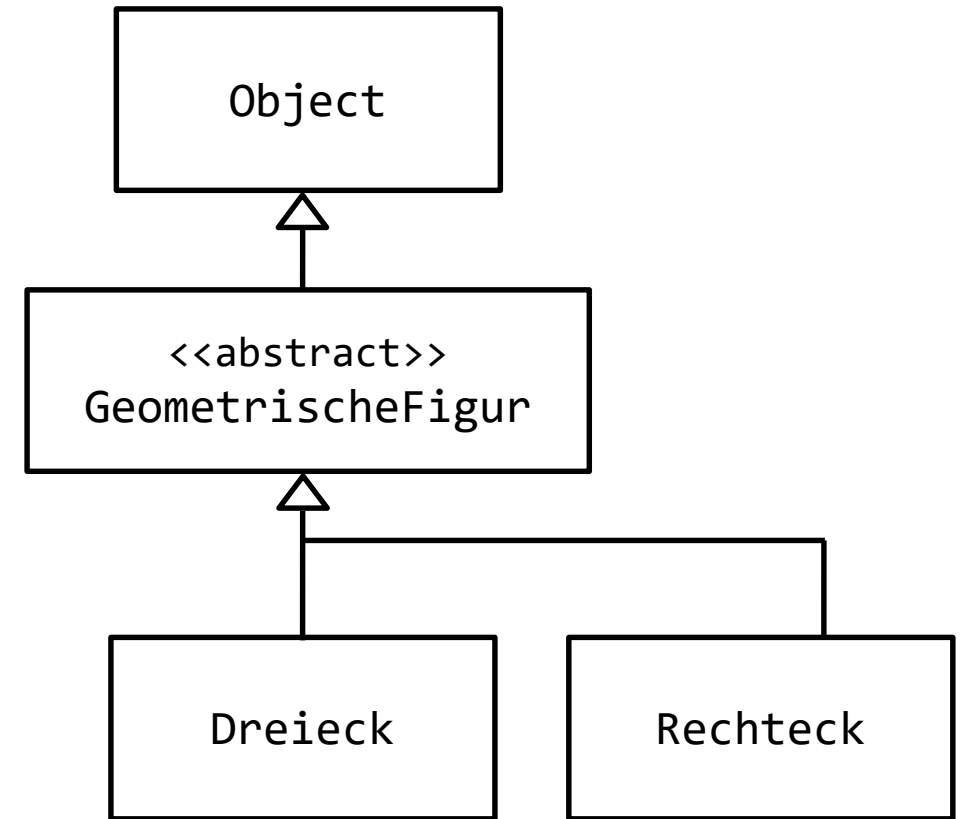
```

public class GeometrischeFigur {
    protected GeometrischeFigur gibInstanz()
    {
        return this;
    }

    public static void main(String[] args) {
        GeometrischeFigur[] figuren = new GeometrischeFigur[2];
        figuren[0] = new Dreieck(2, 4);
        figuren[1] = new Rechteck(2, 4);

        for (int i = 0; i < figuren.length; i++) {
            System.out.println(figuren[i].gibInstanz());
        }
    }
}

```



Welcher Zugriffsschutz wäre für giblInstanz der Klasse Dreieck denkbar?

- A: protected
- B: public
- C: default
- D: private

# Enumeration

Nehmen wir an, wir wollen eine  
Kalenderanwendung schreiben.

Die Wochentage hinterlegen wir als sprechende Konstanten:

```
public class Wochentag {  
    public static final String MONTAG = "Montag";  
    public static final String DIENSTAG = "Dienstag";  
    public static final String MITTWOCH = "Mittwoch";  
    public static final String DONNERSTAG = "Donnerstag";  
    public static final String FREITAG = "Freitag";  
    public static final String SAMSTAG = "Samstag";  
    public static final String SONNTAG = "Sonntag";
```

```
    public static void main(String[] args) {  
        System.out.println(Wochentag.FREITAG);  
    }  
}
```

Beispielzugriff auf einen  
Wochentag

```
public class Wochentag {  
    public static final String MONTAG = "Montag";  
    public static final String DIENSTAG = "Dienstag";  
    public static final String MITTWOCH = "Mittwoch";  
    public static final String DONNERSTAG = "Donnerstag";  
    public static final String FREITAG = "Freitag";  
    public static final String SAMSTAG = "Samstag";  
    public static final String SONNTAG = "Sonntag";
```

```
    public static final String[] wochentage = {MONTAG, DIENSTAG,  
        MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG};
```

```
    public static String welcherWochentagIstInXTagen(String wochentag, int anzahlTage) {  
        for (int i = 0; i < wochentage.length; i++) {  
            if (wochentage[i]==wochentag) return wochentage[(i+anzahlTage)%7];  
        }  
        return "fehlerhafter Tag";  
    }  
}
```

Funktioniert der Vergleich mit == ?

```
    public static void main(String[] args)  
    {  
        System.out.println(welcherWochentagIstInXTagen(Wochentag.FREITAG, 5));  
    }  
}
```

Nehmen wir an, wir wollen wissen,  
welcher Wochentag in 5 Tagen ist.

Für solche Zwecke gibt es Aufzählungstypen.

# Aufzählungstyp - enum

- Wird nur eine bestimmte Anzahl Zustände unterschieden, kann dies durch einen Aufzählungstypen erfolgen.
- Statt dem Schlüsselwort **class** wird das Schlüsselwort **enum** verwendet.

```
public enum Wochentag {  
    Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag  
}
```

wird **ungefähr** übersetzt in:

```
class Wochentag extends Enum {  
    public static final Wochentag Montag = new Wochentag("Montag", 0);  
    public static final Wochentag Dienstag = new Wochentag("Dienstag", 1);  
    ...  
    private Wochentag(String s, int i) {  
        super(s, i);  
    }  
    ...  
}
```

# Aufzählungstyp - enum

- Jedes enum stellt einen Standardsatz an Methoden zur Verfügung. Die wichtigsten:

Methode	Beschreibung
values()	statische Methode, die alle verschiedenen Werte als Array zurückgibt, z.B. Wochentag[]
toString()	gibt die Stringrepräsentation der Aufzählungskonstante als String zurück, z.B. Montag
ordinal()	gibt die Position des Wertes in der Enum-Deklaration zurück beginnend bei 0. Wochentag.Freitag.ordinal() gibt 4 zurück

```
Wochentag[] wochentage = Wochentag.values();
for (int i = 0; i < wochentage.length; i++)
{
    System.out.println(wochentage[i]);
}
```

Montag  
Dienstag  
Mittwoch  
Donnerstag  
Freitag  
Samstag  
Sonntag



Welcher Wochentag ist in 5 Tagen?

```
public static void main(String[] args)
{
    Wochentag wochentag = Wochentag.Freitag;
    System.out.println(Wochentag.values()[(wochentag.ordinal()+5)%7]);
}
```



nochmal ausführlich

```
public static void main(String[] args)
{
    Wochentag wochentag = Wochentag.Freitag;
    Wochentag[] wochentage = Wochentag.values();
    int enumIndexDesAktuellenWochentags = wochentag.ordinal();
    int enumIndexIn5Tagen = (enumIndexDesAktuellenWochentags + 5) % 7;
    System.out.println(wochentage[enumIndexIn5Tagen]);
}
```

# Aufzählungstyp – enum - valueOf

```
public enum Wochentag {  
    Montag, Dienstag, Mittwoch, Donnerstag, Freitag, Samstag, Sonntag  
}
```

- Zugriff und Vergleich können wie folgt durchgeführt werden:

```
public static void main(String[] args)  
{  
    String eingabe = new Scanner(System.in).nextLine();  
    Wochentag w = Wochentag.valueOf(eingabe);  
    if (w == Wochentag.Montag)  
        System.out.println("Ich hasse Montage.");  
    else  
        System.out.println("Heute ist " + w + "!");  
}
```



# Vergleich zwischen normalem Quader und spezialisiertem Quader

```
public class Quader {  
    double laenge;  
    double breite;  
    double tiefe;  
  
    public Quader(double laenge,  
        double breite, double tiefe) {  
        this.laenge = laenge;  
        this.breite = breite;  
        this.tiefe = tiefe;  
    }  
  
    public double berechneVolumen() {  
        return laenge*breite*tiefe;  
    }  
}
```

Die Schnittstelle nach außen bleibt bestehen:

```
public class Main {  
    public static void main(String[] args) {  
        Quader q = new Quader(10, 5, 5);  
        System.out.println(q.berechneVolumen());  
    }  
}
```

```
public class Quader extends Rechteck {  
  
    double tiefe;  
  
    public Quader(double laenge,  
        double breite, double tiefe) {  
        this.laenge = laenge;  
        this.breite = breite;  
        this.tiefe = tiefe;  
    }  
  
    public double berechneVolumen() {  
        return berechneFlaeche()*tiefe;  
    }  
}
```

Das Hauptprogramm funktioniert mit beiden Implementierungen.

Was fällt ansonsten auf?

# Vergleich zwischen normalem Quader und spezialisiertem Quader

```
public class Quader {  
    double laenge;  
    double breite;  
    double tiefe;  
  
    public Quader(double laenge,  
        double breite, double tiefe) {  
        this.laenge = laenge;  
        this.breite = breite;  
        this.tiefe = tiefe;  
    }  
  
    public double berechneVolumen() {  
        return laenge*breite*tiefe;  
    }  
}
```

Die Schnittstelle nach außen bleibt bestehen:

```
public class Main {  
    public static void main(String[] args) {  
        Quader q = new Quader(10, 5, 5);  
        System.out.println(q.berechneVolumen());  
    }  
}
```

```
public class Quader extends Rechteck {  
  
    double tiefe;  
  
    public Quader(double laenge,  
        double breite, double tiefe) {  
        this.laenge = laenge;  
        this.breite = breite;  
        this.tiefe = tiefe;  
    }  
  
    public double berechneVolumen() {  
        return berechneFlaeche()*tiefe;  
    }  
}
```

Der spezialisierte Quader hat zusätzlich die Methode berechneFlaeche von Rechteck geerbt, d.h. der Aufruf `q.berechneFlaeche()` ist mgl.

Ein Quader ist **kein** Rechteck!  
Dazu später mehr!

# super

## Konstruktoraufruf der Oberklasse

- Wird im Konstruktor weder `this` noch `super` für einen Konstruktoraufruf verwendet, wird `super()` beim Übersetzen als erste Anweisung im Konstruktor ergänzt.

```
public class A {  
    public A() {  
        System.out.println("A: Created A");  
    }  
}
```

**wird automatisch vom Compiler ergänzt  
(Übrigens auch bei `public A()`)**

```
public class B extends A {  
    public B(String test) {  
        super();  
        System.out.println("B: " + test);  
    }  
    public static void main(String[] args) {  
        B b = new B("!");  
    }  
}
```

- Der Aufruf eines Konstruktors muss immer die erste Anweisung im Konstruktor sein (sowohl bei `super` als auch bei `this`).

```
public class Wochentag {  
    public static final String MONTAG = "Montag";  
    public static final String DIENSTAG = "Dienstag";  
    public static final String MITTWOCH = "Mittwoch";  
    public static final String DONNERSTAG = "Donnerstag";  
    public static final String FREITAG = "Freitag";  
    public static final String SAMSTAG = "Samstag";  
    public static final String SONNTAG = "Sonntag";
```

```
    public static final String[] wochentage = {MONTAG, DIENSTAG,  
        MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG};
```

```
    public static String welcherWochentagIstInXTagen(String wochentag, int anzahlTage) {  
        for (int i = 0; i < wochentage.length; i++) {  
            if (wochentage[i]==wochentag) return wochentage[(i+anzahlTage)%7];  
        }  
        return "fehlerhafter Tag";  
    }  
}
```

Funktioniert der Vergleich mit == ?

```
    public static void main(String[] args)  
    {  
        System.out.println(welcherWochentagIstInXTagen(Wochentag.FREITAG, 5));  
        System.out.println(welcherWochenTagIstInXTagen("Freitag", 5));  
        System.out.println(welcherWochenTagIstInXTagen(new Scanner(System.in).nextLine(), 5));  
    }  
}
```

```
public class Wochentag {  
    public static final String MONTAG = "Montag";  
    public static final String DIENSTAG = "Dienstag";  
    public static final String MITTWOCH = "Mittwoch";  
    public static final String DONNERSTAG = "Donnerstag";  
    public static final String FREITAG = "Freitag";  
    public static final String SAMSTAG = "Samstag";  
    public static final String SONNTAG = "Sonntag";
```

```
    public static final String[] wochentage = {MONTAG, DIENSTAG,  
        MITTWOCH, DONNERSTAG, FREITAG, SAMSTAG, SONNTAG};
```

```
    public static String welcherWochentagIstInXTagen(String wochentag, int anzahlTage) {  
        for (int i = 0; i < wochentage.length; i++) {  
            if (wochentage[i].equals(wochentag)) return wochentage[(i+anzahlTage)%7];  
        }  
        return "fehlerhafter Tag";  
    }  
}
```

Besser equals

```
    public static void main(String[] args)  
    {  
        System.out.println(welcherWochentagIstInXTagen(Wochentag.FREITAG, 5));  
        System.out.println(welcherWochenTagIstInXTagen("Freitag", 5));  
        System.out.println(welcherWochenTagIstInXTagen(new Scanner(System.in).nextLine(), 5));  
    }  
}
```

Nehmen wir an, wir wollen wissen,  
welcher Wochentag in 5 Tagen ist.



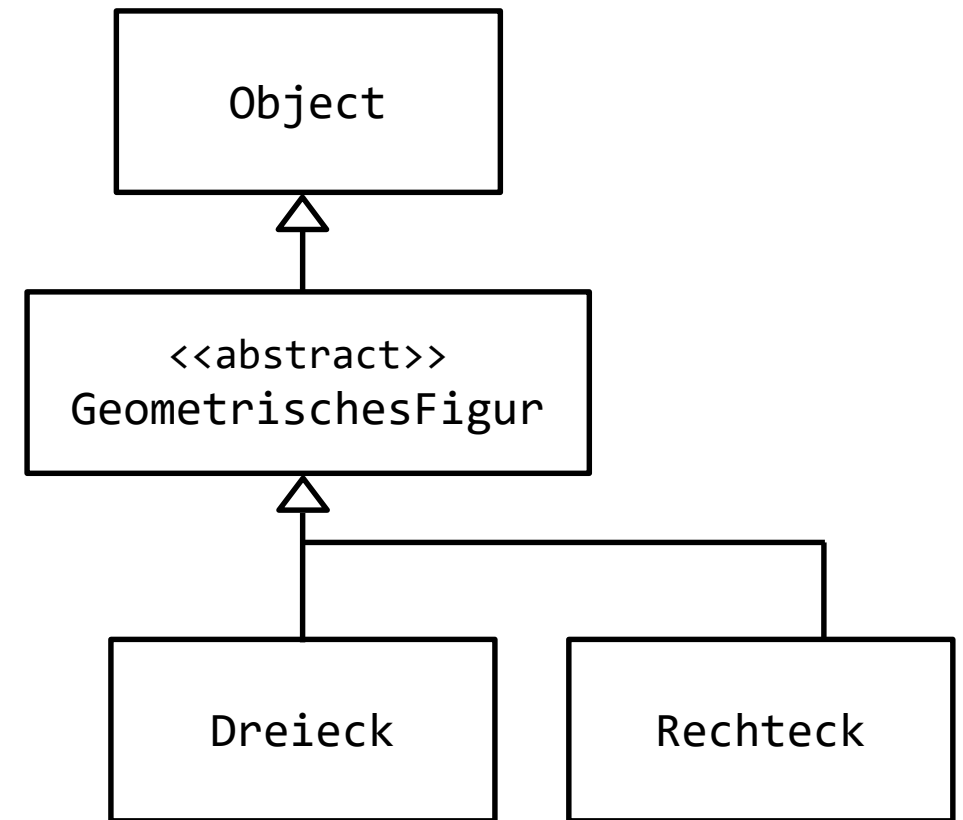
```

public class GeometrischeFigur {
    protected GeometrischeFigur gibInstanz()
    {
        return this;
    }

    public static void main(String[] args) {
        GeometrischeFigur[] figuren = new GeometrischeFigur[2];
        figuren[0] = new Dreieck(2, 4);
        figuren[1] = new Rechteck(2, 4);

        for (int i = 0; i < figuren.length; i++) {
            System.out.println(figuren[i].gibInstanz());
        }
    }
}

```



Welche Klassen wären als Rückgabetyt für gibInstanz der Klasse Dreieck denkbar?

- A: GeometrischeFigur
- B: Dreieck
- C: Object
- D: Rechteck

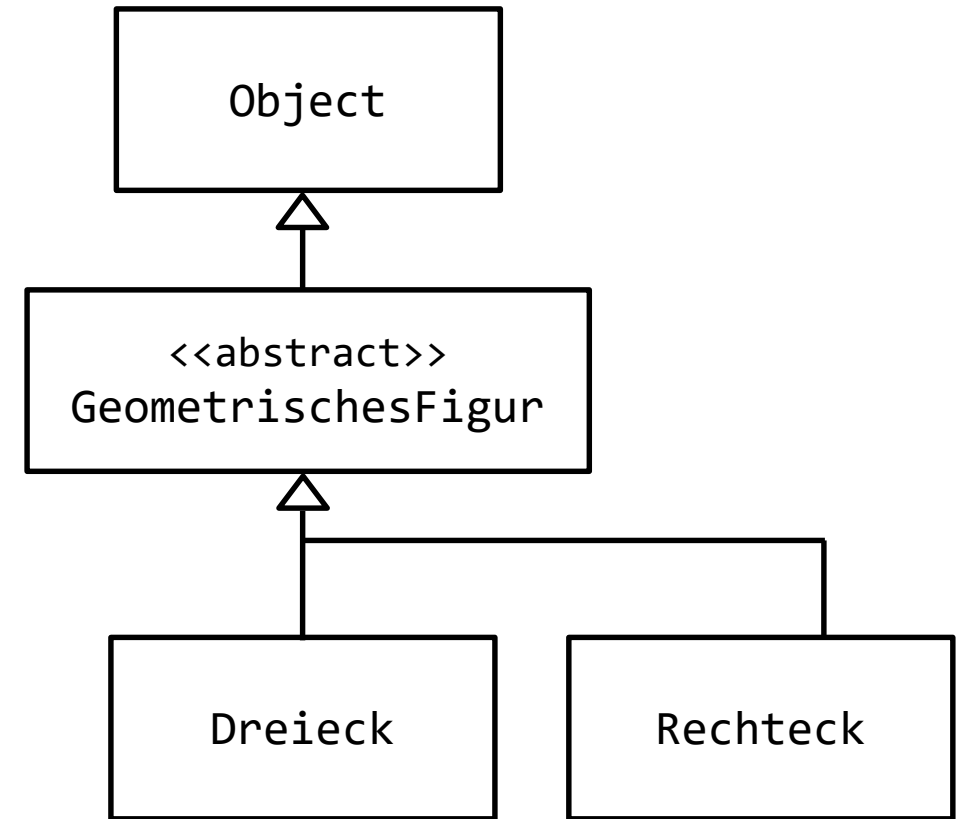
```

public class GeometrischeFigur {
    protected GeometrischeFigur gibInstanz()
    {
        return this;
    }

    public static void main(String[] args) {
        GeometrischeFigur[] figuren = new GeometrischeFigur[2];
        figuren[0] = new Dreieck(2, 4);
        figuren[1] = new Rechteck(2, 4);

        for (int i = 0; i < figuren.length; i++) {
            System.out.println(figuren[i].gibInstanz());
        }
    }
}

```



Welcher Zugriffsschutz wäre für giblInstanz der Klasse Dreieck denkbar?

- A: protected
- B: public
- C: default
- D: private