

# Grundlagen der Programmierung

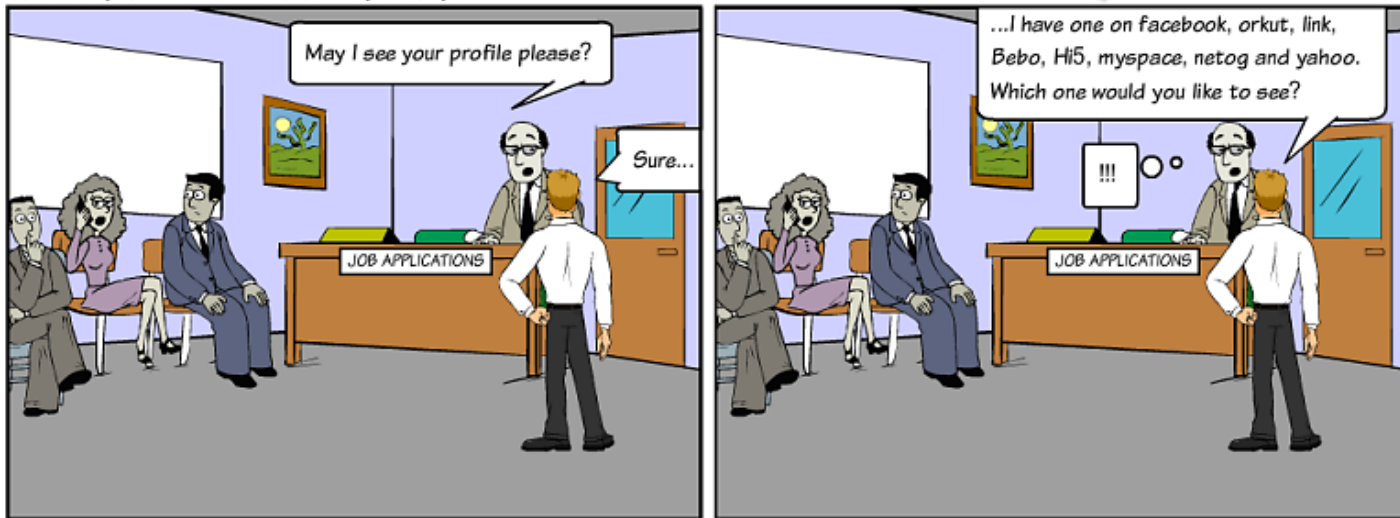
## VL14: Polymorphismus

Prof. Dr. Samuel Kounev  
Jóakim von Kistowski  
Norbert Schmitt

**HIGH PROFILED - BY MEERASAPRA**



www.toondoo.com



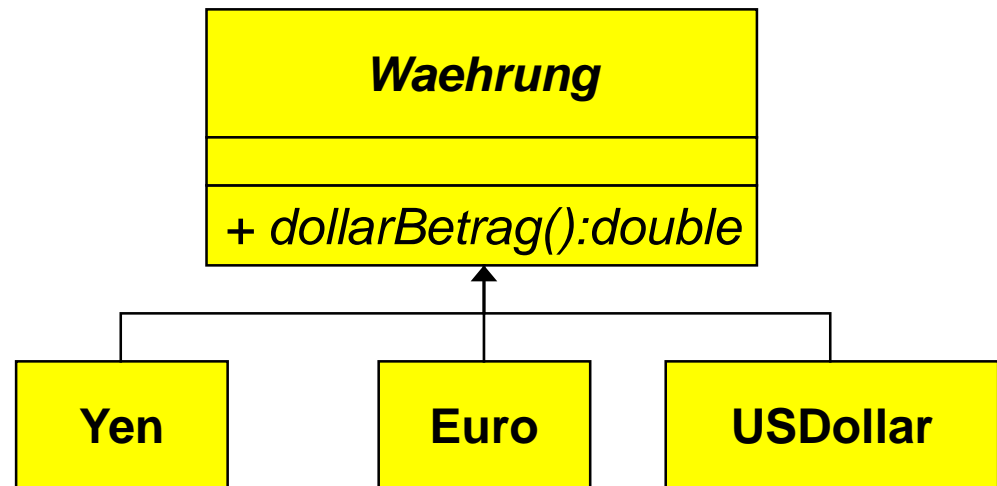
- Überschreiben von Methoden
- Typumwandlungen innerhalb einer Objekthierarchie
- Überschreiben von Variablen
- Die Klasse `java.lang.Object`
- Interfaces



Wir haben am Beispiel einer internationalen Hotelkette eine Hierarchie von Klassen entwickelt, mit deren Hilfe wir verschiedene Währungen miteinander in Verbindung bringen und automatisch in eine Referenzwährung (den US Dollar) umrechnen konnten.

Als Basis verwendeten wir eine Klasse namens **Währung**, die wir als **abstrakt** (Schlüsselwort: **abstract**) erklärt hatten.

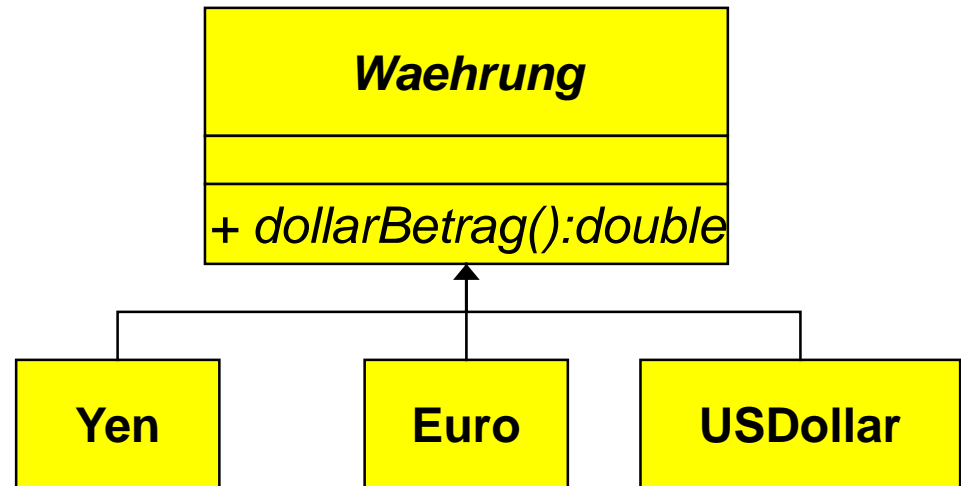
Die Klassen **Yen**, **Euro** und **USDollar** **erweitern** die abstrakte Klasse.

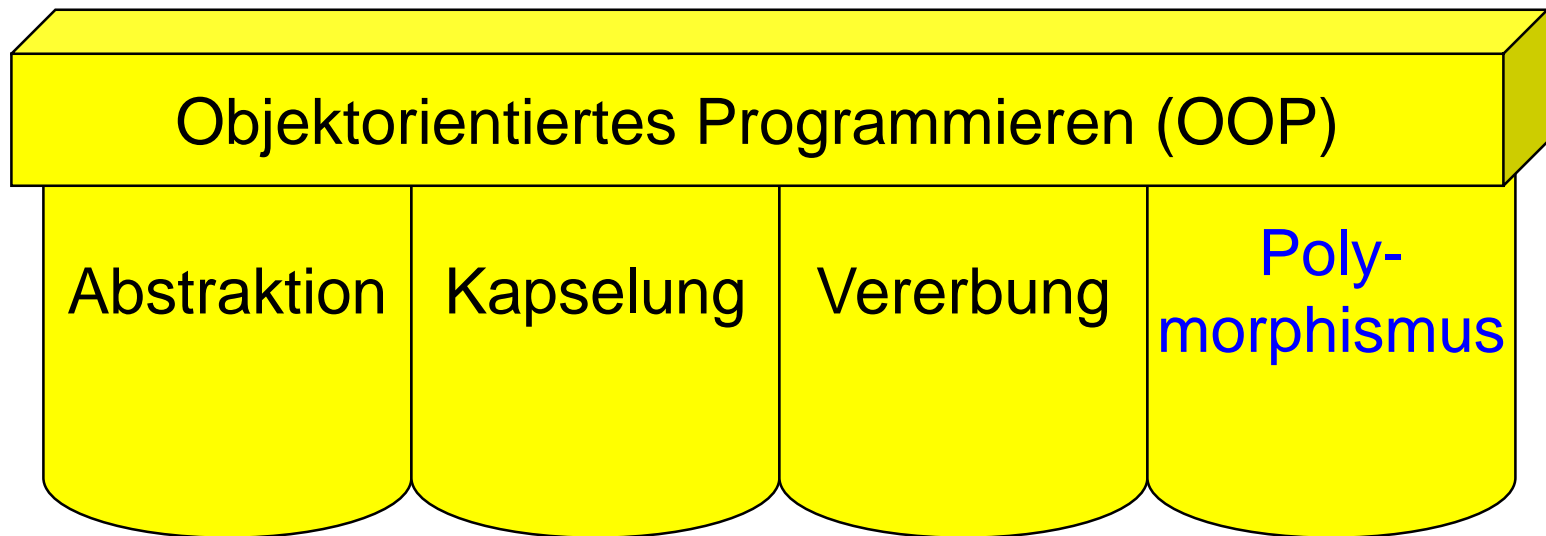


Die Methode `dollarBetrag` wird in jeder Subklasse **überschrieben** (overridden), d.h. durch eine andere, spezielle Methode ersetzt.

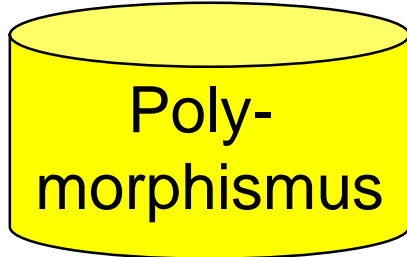
Obwohl sich die Implementierungen hierbei unterscheiden, können doch alle Objekte über die einheitliche **Schnittstelle** angesprochen werden, welche sie von der Klasse `Wahrung` erben.

Die neuen Methoden nahmen den Platz ihres abstrakten Vorgängers ein - sie wurden an seiner statt aufgerufen.





Zur Wiederholung



Die Möglichkeit, Methoden innerhalb der Objekthierarchie zu **überschreiben** (*override*) und somit individuelles Verhalten verschiedener Klassen auf dieselbe Anweisung zu modellieren.

Mit Hilfe des Polymorphismus kann der Entwickler:

- ohne Probleme **Besonderheiten** bei den verschiedenen Subklassen **berücksichtigen**,
- **einheitliche Schnittstellen** erzeugen und somit
- Entwicklungszeit sparen und
- eine Vielzahl von Fehlerquellen ausschalten.

## Beispiel

### Konzert im Hinterhof

Pünktlich zu jedem Vollmond trifft sich eine Meute von vier braven Haushunden, um „den Wolf“ in sich zu spüren. A capella jaulen sie dann gemeinsam den Vollmond an.

Der erste Hund ist eine lebensfrohe Promenadenmischung, die man keiner besonderen Rasse zuordnen kann. Hund Nr. 2 ist ein kleiner Pekinese - er sorgt für den Sopran. Der dritte Hund ist ein Bernhardiner mit tiefer, sonorer Bassstimme. Der letzte im Bunde gehört zur seltenen Gattung der Echohunde: er klingt zwar wie jeder andere Hund, pflegt sich aber stets zu wiederholen.

Jeder der vier Hunde tritt hervor, um einmal zu jaulen, zu bellen und dann wieder zu jaulen. Danach geben die Tiere bis zum nächsten Vollmond Ruhe.

## Beispiel

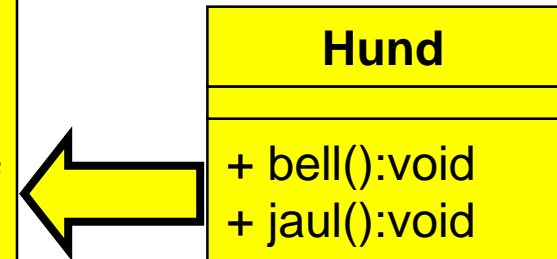
- Da wir natürlich nicht bis zum nächsten Vollmond warten wollen, möchten wir das Konzert auf dem Computer simulieren.
- Zu diesem Zweck entwerfen wir eine Klasse **Hund**, die ein Mitglied aus dem stimmkräftigen Quartett darstellen soll.
- Die Darstellung des Gesangs (bellen und jaulen) soll auf dem Bildschirm erscheinen.

Hund
+ bell():void + jaul():void



## Beispiel

```
public class Hund {  
  
    public void bell() {  
        System.out.println("WAU  ");  
    }  
  
    public void jaul() {  
        System.out.println("JAUL  ");  
    }  
  
}
```



## Beispiel

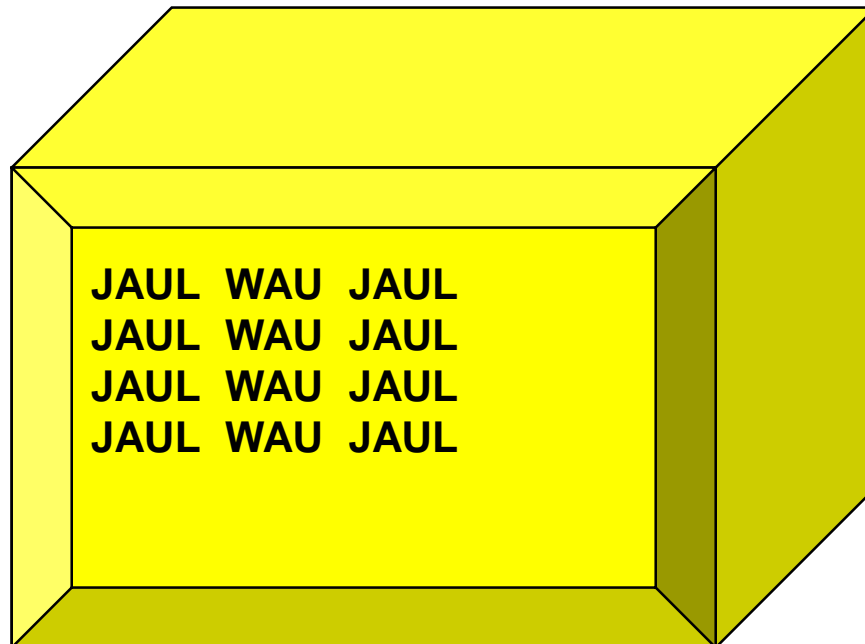
```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute = new Hund[4];  
        meute[0]=new Hund();  
        meute[1]=new Hund();  
        meute[2]=new Hund();  
        meute[3]=new Hund();  
        for (int i=0; i < meute.length; i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```

Hund
+ bell():void + jaul():void

Leider „klingt“ die Ausgabe auf dem Bildschirm im Moment noch wenig realistisch.

Grund hierfür ist ein Mangel an Individualität. Ein Bernhardiner sollte nicht so klingen wie ein Pekinese!

Wir müssen unser Klassenmodell also verfeinern...



Beispiel

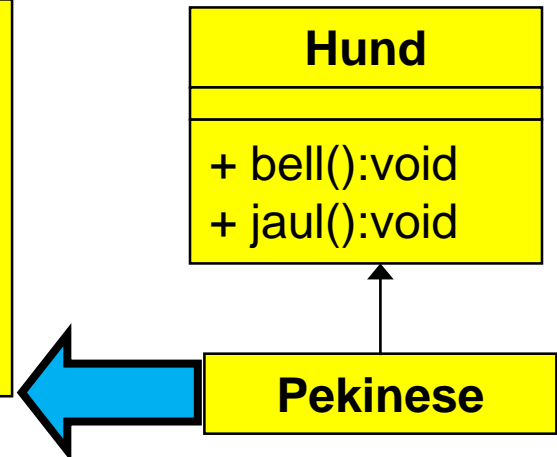
Hund
+ bell():void + jaul():void

## Beispiel

Für die Promenadenmischung (Hund Nr. 1) können wir auch weiterhin die Klasse **Hund** verwenden - diese ist eh nicht weiter zu klassifizieren.

Wir kümmern uns deshalb um den zweiten Hund und entwerfen eine Klasse namens **Pekinese**.

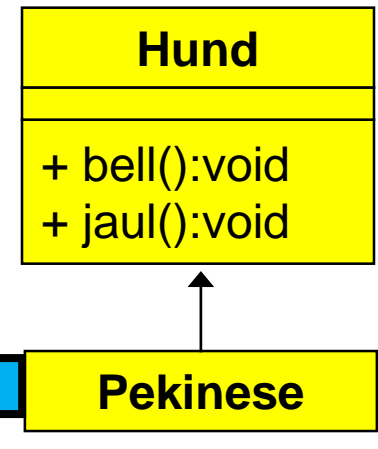
```
public class Pekinese extends Hund {  
  
  
  
  
  
  
  
  
  
}
```



## Beispiel

Wir wollen also, dass sich Instanzen der Klasse **Pekinese** beim Aufruf der Methode `bell()` anders verhalten als sonstige Instanzen der Klasse **Hund**. Wir müssen die Methode `bell()` für diese Klasse also **überschreiben**:

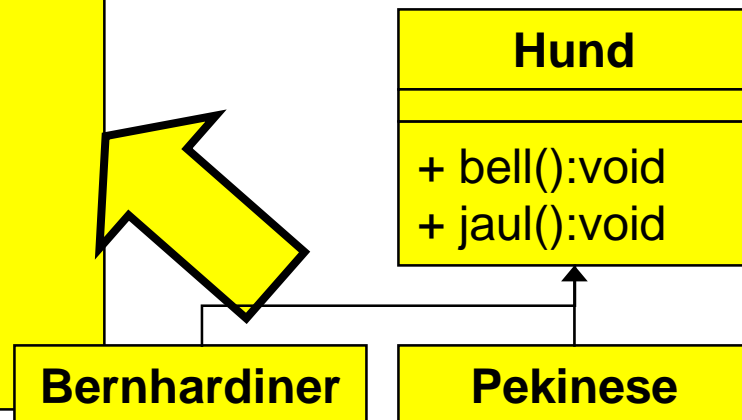
```
public class Pekinese extends Hund {  
  
    public void bell() {  
        System.out.println("Wiff ");  
    }  
}
```



## Beispiel

Analog hierzu entwerfen wir für Hund Nr. 3 eine Klasse namens **Bernhardiner**.

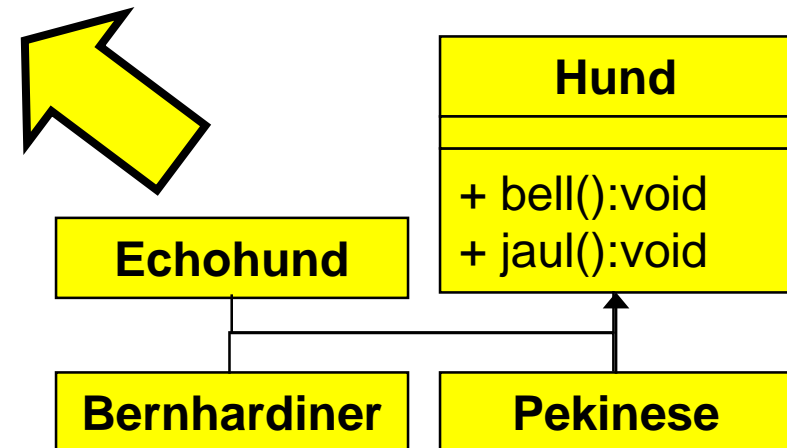
```
public class Bernhardiner extends Hund {  
  
    public void bell() {  
        System.out.print("WRRRUFF!  ");  
    }  
  
    public void jaul() {  
        System.out.print("JAUUUUUUUL  ");  
    }  
  
}
```



## Beispiel

Da der **Echohund** wie ein normaler Hund bellt (und jault), seine Laute allerdings immer wiederholt, liegt der Gedanke nahe, die Original-Methoden einfach zweimal aufzurufen.

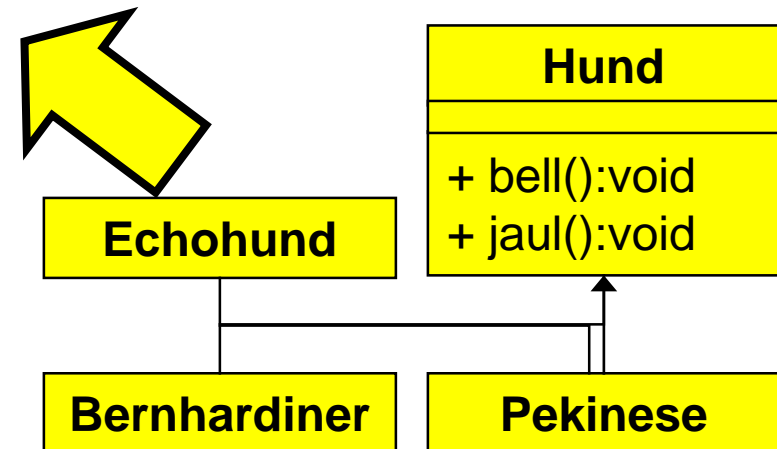
```
public class Echohund extends Hund {  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaul() {  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    }  
}
```



## Beispiel

Würden wir jedoch bei einer derart gestalteten Klasse die Methoden `bell()` oder `jaul()` aufrufen, so würde das Programm mit einer **Fehlermeldung** abbrechen.

```
public class Echohund extends Hund {  
  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaul() {  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    }  
}
```





## Beispiel

Würden wir jedoch bei einer derart gestalteten Klasse die Methoden `bell()` oder `jaul()` aufrufen, so würde das Programm mit einer **Fehlermeldung** abbrechen.

```
public class Echohund extends Hund {  
  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
}
```



Hund

Der Grund hierfür liegt in einer Endlosschleife die wir produziert haben: Die neue Methode `bell()` ruft anstelle der originalen Methode (aus der Klasse `Hund`) erneut die in der Klasse `Echohund` überschriebene Methode `bell()` auf - also sich selbst! Das Programm bricht ab, sobald der für Methodenaufrufe reservierte **Speicher überläuft**.

## Beispiel

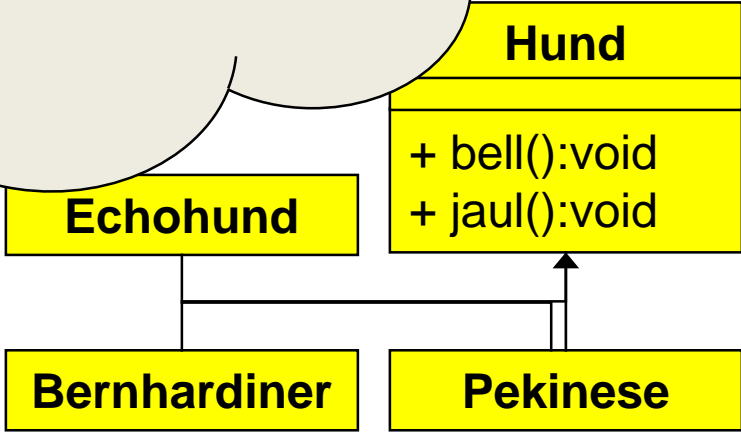
Würden wir jedoch bei einer derart gestalteten Klasse die Methoden `bell()` oder `jaul()` aufrufen, so würde das Programm mit einer **Fehlermeldung** abbrechen.

```
public class Echohund extends Hund {

    public void bell() {
        bell(); // belle
        bell(); // belle
    }

    public void jaul() {
        jaul(); // jaule einmal
        jaul(); // jaule zweimal
    }
}
```

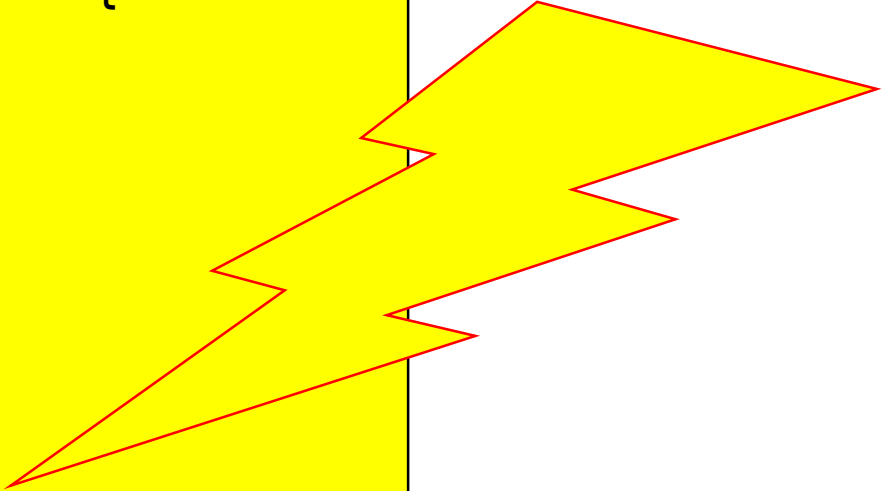
Wie lässt sich auf die Methode der Superklasse zugreifen?



## Beispiel

Ein erster Ansatz wäre der Versuch, die Instanz einer Typumwandlung zu unterziehen. So wie sich etwa eine `double`-Zahl wie `3.14` durch eine Umwandlung der Form `(int) 3.14` in eine ganze Zahl umwandeln lässt, kann man ähnlich auch für Objekte verfahren.

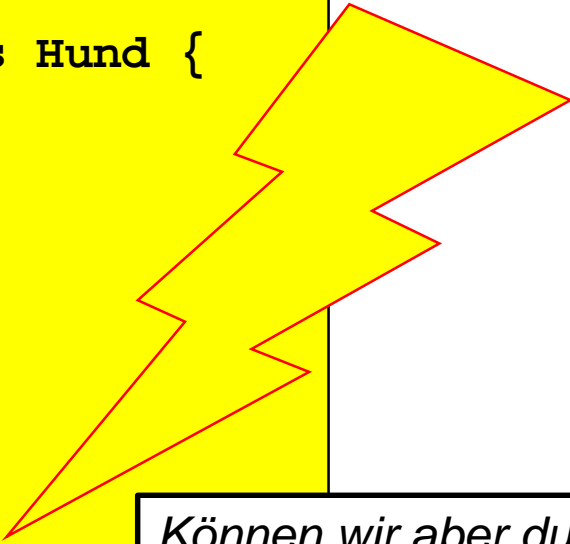
```
public class Echohund extends Hund {  
  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaul() {  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    }  
  
}
```



## Beispiel

Eine entsprechende Umwandlung existiert also auch für Objekte in einer Hierarchie. Wir können eine Instanz **Fido** der Klasse **Echohund** durch einen Ausdruck der Form **(Hund) Fido** als eine Instanz der Klasse **Hund** betrachten (implizit haben wir das bereits getan).

```
public class Echohund extends Hund {  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaul() {  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    }  
}
```



*Können wir aber durch diese Umwandlung auf die alten Methoden zurückgreifen?*

## Beispiel

Wir wandeln die betroffene Instanz (**this**) in eine Instanz der Superklasse **Hund** um und rufen für diese dann die Methode **bell** bzw. **jaul** auf.

*Würden wir nach dieser Verbesserung für eine Instanz der Klasse die Methode **bell** aufrufen, so würde...*

```
public class Echohund extends Hund {  
  
    public void bell() {  
        Hund hund=(Hund)this;  
        hund.bell();  
        hund.bell();  
    }  
  
    public void jaul() {  
        Hund hund=(Hund)this;  
        hund.jaul();  
        hund.jaul();  
    }  
}
```

**...unser Programm dennoch  
genauso abstürzen wie zuvor!**

## Beispiel

Grund ist erneut das Prinzip des **Polymorphismus**. Eine Subklasse ist eine spezialisierte Fassung ihrer Superklasse. Wenn wir eine Methode überschreiben, so wird das aus der Subklasse instantiierte Objekt immer diese neue Methode statt der alten aufrufen - **egal**, ob sie als Instanz der Superklasse aufgefasst wird oder nicht!

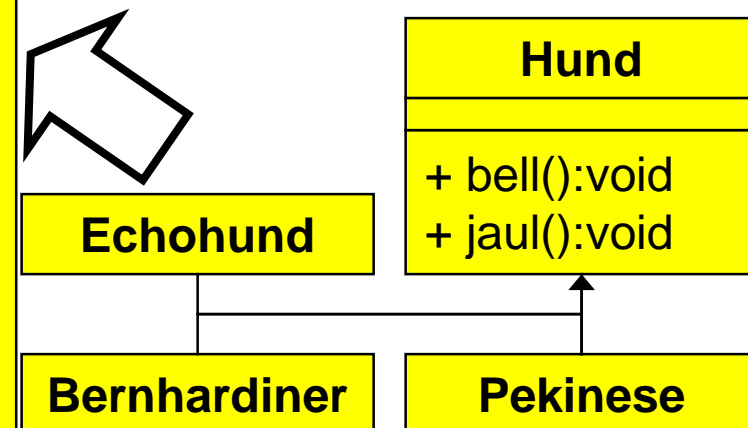
```
public class Echohund extends Hund {  
  
    public void bell() {  
        Hund hund=(Hund)this;  
        hund.bell();  
        hund.bell();  
    }  
  
    public void jaul() {  
        Hund hund=(Hund)this;  
        hund.jaul();  
        hund.jaul();  
    }  
}
```

*Wie lässt sich also auf die Methoden der Superklasse zugreifen?*

## Beispiel

Die Antwort steckt in dem Schlüsselwort **super**. So wie wir mit **this** auf alle Bestandteile einer Instanz zugreifen können, haben wir mit **super** Zugriff auf alle Bestandteile seiner Superklasse.

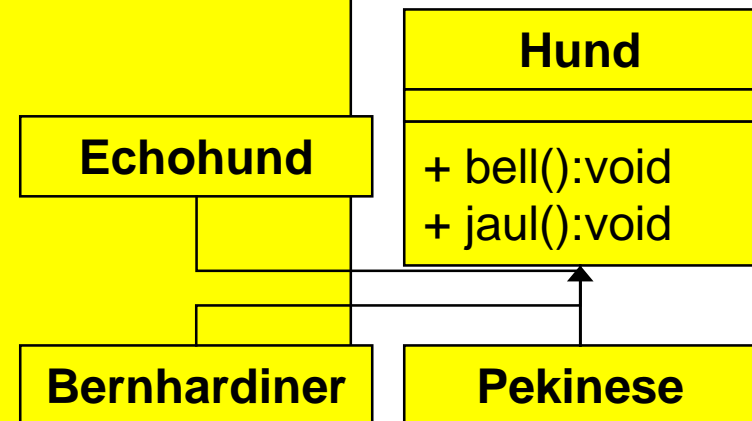
```
public class Echohund extends Hund {  
  
    public void bell() {  
        super.bell();  
        super.bell();  
    }  
  
    public void jaul() {  
        super.jaul();  
        super.jaul();  
    }  
  
}
```



## Beispiel

Werfen wir nun noch einen Blick auf unser Hauptprogramm:

```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute=new Hund[4];  
        meute[0]=new Hund();  
        meute[1]=new Hund();  
        meute[2]=new Hund();  
        meute[3]=new Hund();  
        for (int i=0;i<meute.length;i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```





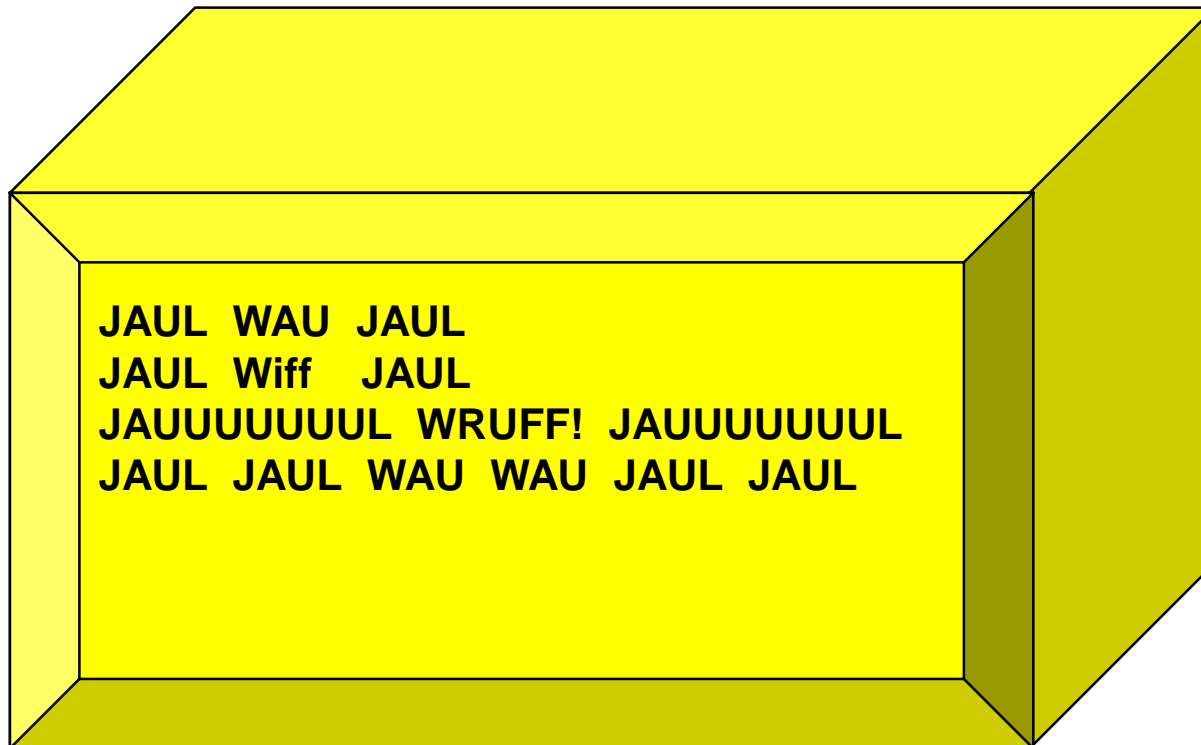
## Beispiel

Werfen wir nun noch einen Blick auf unser Hauptprogramm:

```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute=new Hund[4];  
        meute[0]= new Hund(); // Promenadenmischung  
        meute[1]= new Pekinese(); // Hund Nr.2  
        meute[2]= new Bernhardiner(); // Hund Nr.3  
        meute[3]= new Echohund(); // Hund Nr.4  
        for (int i=0;i<meute.length;i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```

## Beispiel

Das Prinzip des Polymorphismus erlaubt durch das Überschreiben von Methoden eine individuelle Ausprägung **verschiedener Objektklassen** unter der einheitlichen Schnittstelle **einer gemeinsamen Superklasse**.



- Es wird erst zur Laufzeit entschieden welche Version der Methode aufgerufen wird.
- Dies war abhängig davon, ob das jeweilige Objekt eine Instanz der Ober- oder einer Unterklasse ist.
- Bei diesem Vorgang des Überschreibens spricht man auch von **dynamischer Bindung** (**dynamic binding**).

## Fazit

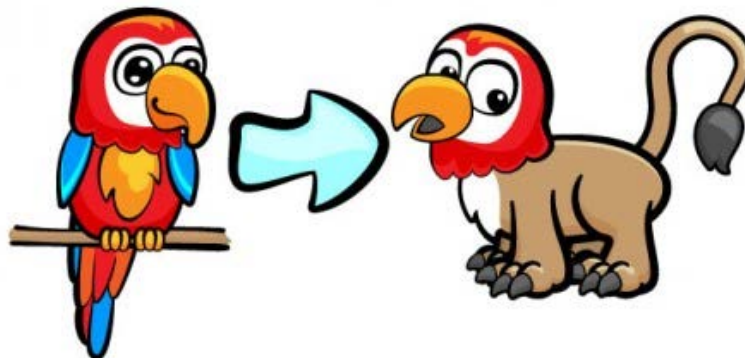
In einer Subklasse ist es möglich, Methoden neu zu verfassen, die mit derselben Schnittstelle bereits in der Superklasse definiert wurden. Man bezeichnet diesen Vorgang als **Überschreiben**.

In Verbindung mit dem Prinzip des **Polymorphismus** wird dieses Verfahren zu einem der wichtigsten Anwendungsgebiete der objektorientierten Programmierung:

Auf diese Weise können wir Algorithmen, die auf der allgemeineren Form einer Superklasse definiert sind, direkt auch auf alle ihre Subklassen anwenden, *obwohl sich diese in ihren Methoden vielleicht vollkommen unterschiedlich verhalten*.

- Kann man das Überschreiben von Methoden verhindern?
- Der Modifizierer **final** vor einer Methoden-Deklaration verhindert das Überschreiben dieser Methode in einer Unterklasse.
- Der compilierte Programmcode wird **effizienter**, weil kein dynamisches Binden mehr durchgeführt werden muss.
- So wird auch für einige Methoden aus der nachfolgend beschriebenen Klasse **Object** durch den Modifizierer **final** sichergestellt, dass diese Methoden für alle Java-Objekte **das gleiche Verhalten** aufweisen.

- Überschreiben von Methoden
- **Typumwandlungen innerhalb einer Objekthierarchie**
- Überschreiben von Variablen
- Die Klasse `java.lang.Object`
- Interfaces



## Konzert im Hinterhof, Teil 2

Die letzte „Mondscheinsonate“ unseres Hundequartetts blieb nicht ohne Folgen. Ein Nachbar beschwerte sich wegen des Lärms, so dass die Anwohnerschaft gezwungen war, drastische Schritte einzuleiten. Man tat sich zusammen und versah das Gelände mit einer engmaschigen Umzäunung, so dass kein Hund mehr in der Lage sein würde, den Hinterhof zu betreten. So glaubte man zumindest...

Als die vier Hunde in der nächsten Vollmondnacht zusammentrafen, fanden sie den Hof umzäunt und das Tor verschlossen. Der Zaun war so hoch, dass lediglich der Bernhardiner mit einem gewaltigen Satz hinüberhechten konnte. Der Pekinese, die Promenadenmischung und der Echohund zogen enttäuscht von dannen...

## Beispiel

```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute=new Hund[4];  
        meute[0]= new Hund(); // Promenadenmischung  
        meute[1]= new Pekinese(); // Hund Nr.2  
        meute[2]= new Bernhardiner(); // Hund Nr.3  
        meute[3]= new Echohund(); // Hund Nr.4  
        for (int i=0;i<meute.length;i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```

Wir wollen unser Programm **so wenig wie möglich verändern**, um es weiterhin so allgemein wie möglich zu halten.

Vielleicht findet der Pekinese ja in einem Monat ein engmaschiges Loch, so dass auch er wieder am Konzert teilnehmen kann.

Die Belegung des Feldes und damit die Methode **main** soll also unangetastet bleiben.



## Beispiel

Da wir in der `main`-Methode keine Veränderungen vornehmen wollen, müssen wir eine geeignete Erweiterung in der Methode `klaeffDichAus` vornehmen.

```
public static void klaeffDichAus(Hund h) {  
    if (h ist ein Bernhardiner) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
}
```

Wie erhalten wir Informationen darüber, ob es sich bei dem Objekt um die Instanz einer „**verwandten**“ Klasse handelt?

In komplexeren Programmen wird es oftmals notwendig sein, ein beliebiges Objekt auf seine Klassenzugehörigkeit zu prüfen. Aus diesem Grund haben die Entwickler der Programmiersprache Java den **instanceof**-Operator geschaffen.

Dieser Operator besagt, ob ein bestimmtes Objekt eine bestimmte Klasse instantiiert.

Der Operator wird in der Form

**<Objektname> instanceof <Klassenname>**

verwendet und liefert einen boolschen Wert als Ergebnis.

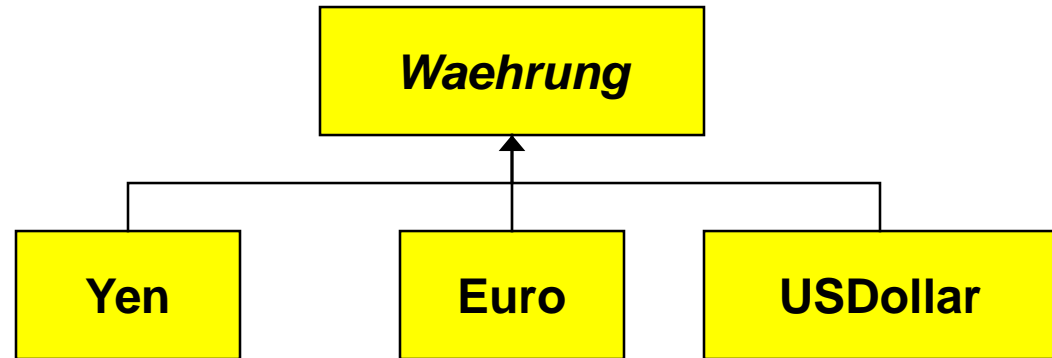
## Beispiel

Wir instantiieren ein **Euro**-Objekt durch die Zuweisung

```
Wahrung cent = new Euro(0.01);
```

und vergleichen dieses mit verschiedenen anderen Klassen aus unserer Klassenhierarchie.

cent : Euro



## Beispiel

Die Operation

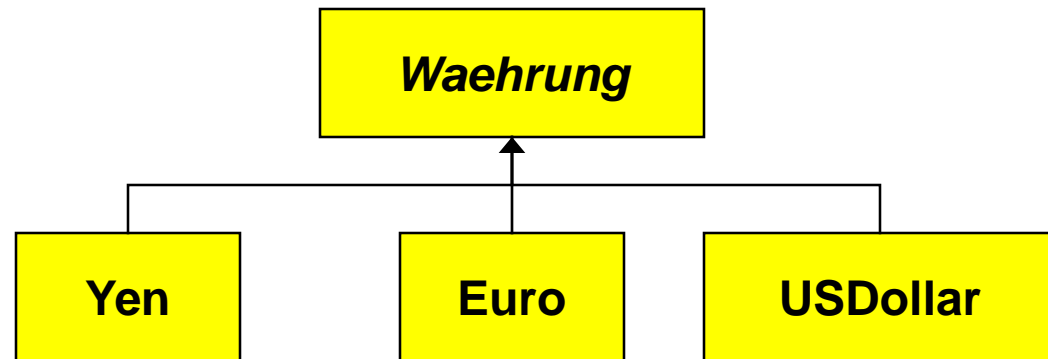
`cent instanceof Euro`

überprüft, ob das Objekt Instanz der Klasse **Euro** ist.

Diese Frage ist zu bejahen, denn obwohl wir das Objekt nur durch eine Variable vom Typ **Wahrung** referenzieren, handelt es sich dennoch um ein **Euro**-Objekt. Der Rückgabewert der Operation ist also **true**.

`cent : Euro`

`cent instanceof Euro` 



# Beispiel

Die Operation

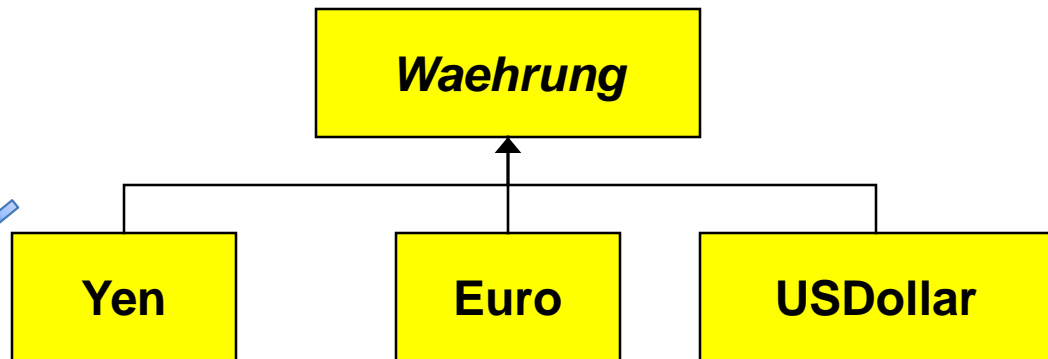
`cent instanceof Waehrung`

überprüft, ob das Objekt Instanz der Klasse `Waehrung` ist.

Auch diese Aussage trifft zu, denn `Waehrung` ist Superklasse von `Euro` - das Ergebnis ist also wiederum `true`.

`cent : Euro`

`cent instanceof Euro` ✓  
`cent instanceof Waehrung` ✓



## Beispiel

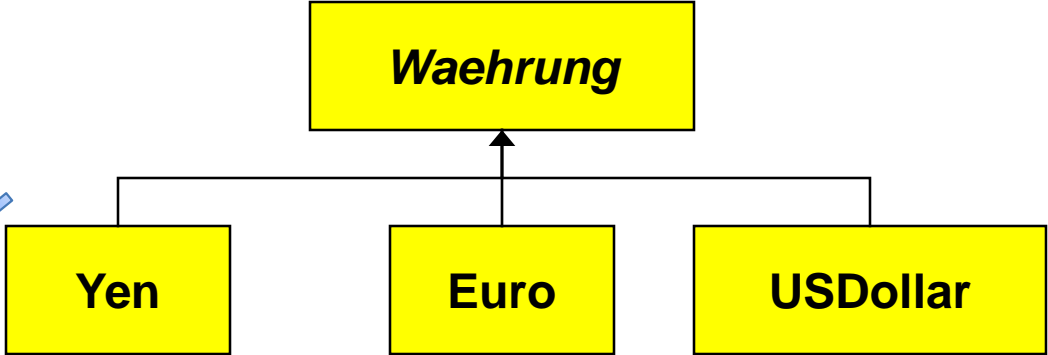
Die Operation

`cent instanceof Yen`

liefert im Gegensatz zu den anderen Vergleichen ein negatives Ergebnis. Die Klassen stehen in keiner direkten Beziehung zueinander!

`cent : Euro`

<code>cent instanceof Euro</code>	✓
<code>cent instanceof Waehrung</code>	✓
<code>cent instanceof Yen</code>	✗



## Beispiel

Zurück zu unserem ursprünglichen Problem:

```
public static void klaeffDichAus(Hund h) {  
    if (h instanceof Bernhardiner) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
}
```

Wir wollten das System überprüfen lassen, ob die Variable **h** eine Instanz der Klasse **Bernhardiner** referenziert. Mit Hilfe des **instanceof**-Operators ist dies nun kein Problem mehr.

## Hinweis

In den bisherigen Beispielen war es noch nicht notwendig gewesen, explizite Typumwandlung an Objekten anzuwenden. Die Programmzeilen

```
Wahrung eineWahrung = new Euro(13);  
Hund fifi = new Bernhardiner();
```

wurden fehlerfrei übersetzt, da das System von der `sub`- zur Superklasse implizit (also automatisch) konvertieren kann.

Die umgekehrte Richtung (**von der Super- zur Subklasse**) lässt sich **nicht** automatisch bewerkstelligen!



## Beispiel

Um unsere Methode **klaefffDichAus** etwas übersichtlicher zu gestalten, wollen wir das übergebene Objekt in einer anderen Variablen speichern. Sobald sicher ist, dass es sich um ein **Bernhardiner**-Objekt handelt, wollen wir dieses durch eine Variable namens **berni** referenzieren.

```
public static void klaefffDichAus(Hund h) {  
    if (h instanceof Bernhardiner) {  
        Bernhardiner berni=h;  
        berni.jaul();  
        berni.bell();  
        berni.jaul();  
        System.out.println(); // kurze Pause  
    }  
}
```

Der Versuch, das so veränderte Programm zu kompilieren, schlägt jedoch fehl. Statt des gewünschten Results erhalten wir eine Fehlermeldung der Form **Explicit cast needed to convert Hund to Bernhardiner.**

## Beispiel

```
public static void klaeffDichAus(Hund h) {  
    if (h instanceof Bernhardiner) {  
        Bernhardiner berni = (Bernhardiner) h;  
        berni.jaul();  
        berni.bell();  
        berni.jaul();  
        System.out.println(); // kurze Pause  
    }  
}
```

Obwohl wir dank der **if**-Abfrage garantieren können, dass es sich bei dem von **h** referenzierten Objekt um einen **Bernhardiner** handelt, kann das System dies nicht automatisch erkennen.

Wir müssen eine explizite Typumwandlung vornehmen, wie wir sie bereits von den einfachen Datentypen her kennen.

# Inhalt

- Überschreiben von Methoden
- Typumwandlungen innerhalb einer Objekthierarchie
- **Überschreiben von Variablen**
- Die Klasse `java.lang.Object`
- Interfaces

Wir haben uns bereits sehr intensiv mit der Frage beschäftigt, wie Objekthierarchien, das Prinzip des Polymorphismus und das Überschreiben von Methoden miteinander in Zusammenhang stehen. Hierbei haben wir festgestellt, dass durch das Überschreiben von Methoden eben jene „Spezialisierung“ von Subklassen möglich war, die eine große Stärke der objektorientierten Modellierung darstellt.



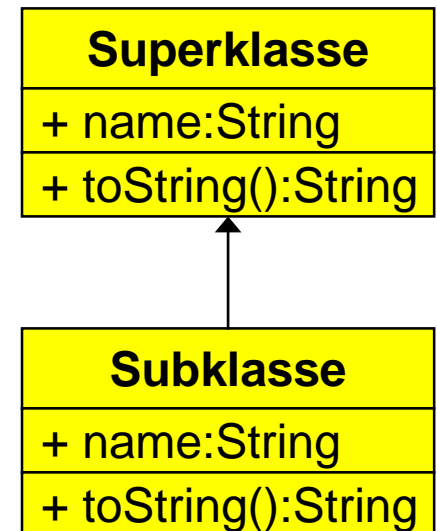
***Lässt sich dies auf Variablen übertragen?***

- Tatsächlich ist es in Java auch möglich, Variablen zu überschreiben.
- Jedoch gilt das Prinzip des **Polymorphismus** in Java **nicht für Variablen!**
- Auch wenn das Original durch eine Variable gleichen Namens verdeckt wird, unterscheidet das System klar zwischen der Variablen der Sub- und der Superklasse.

## Beispiel

Wir entwerfen zwei einfache Klassen, Sub- und Superklasse, die wir zweckmäßigerweise auch so nennen.

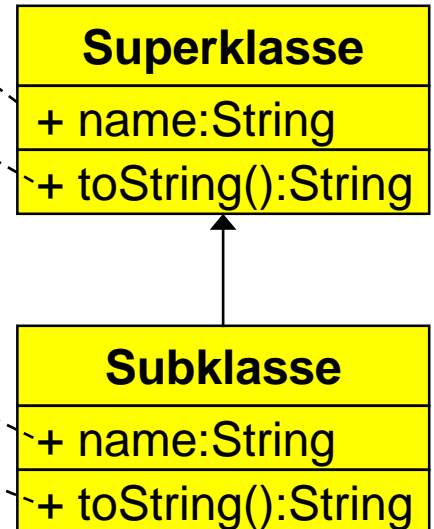
Beide verfügen über eine Instanzvariable „**name**“ und eine Methode **toString**. Diese Methode wird in der **Subklasse** neu definiert, das heißt sie **überschreibt** die entsprechende Methode ihrer **Superklasse**.



## Beispiel

Werfen wir zuerst einen Blick auf den erzeugenden Quelltext:

```
public class Superklasse {  
    public String name="Super";  
  
    public String toString(){  
        return this.name;  
    }  
}  
  
public class Subklasse extends Superklasse {  
    public String name="Sub";  
  
    public String toString() {  
        return this.name;  
    }  
}
```



## Beispiel

Wir vermerken diese Anfangsbelegung in Form von Kommentaren und werden nun mit unserem eigentlichen Testprogramm beginnen.

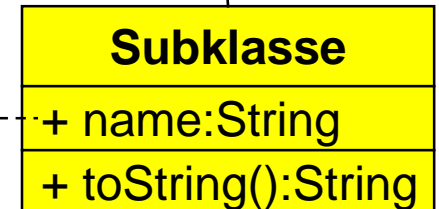
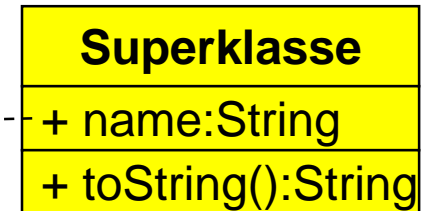
```
Superklasse vater      =new Superklasse();
Subklasse   sohn       =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;
```

```
System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);
```

```
System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```

name ist auf „Super“ gesetzt.

name ist auf „Sub“ gesetzt.





Wir erzeugen zwei neue Objekte **vater** und **sohn**, welche jeweils Super- und Subklasse instantiieren.

Ferner erzeugen wir eine Referenz **gewandelt**, in der wir das **sohn**-Objekt als Instanz der Superklasse referenzieren.

Wir werden nun den Inhalt der Instanzvariablen auf verschiedenem Wege auf dem Bildschirm auszugeben versuchen.

```
Superklasse vater    =new Superklasse();  
Subklasse   sohn     =new Subklasse();  
Superklasse gewandelt=(Superklasse)sohn;
```

```
System.out.println(vater);  
System.out.println(sohn);  
System.out.println(gewandelt);
```

```
System.out.println(vater.name);  
System.out.println(sohn.name);  
System.out.println(gewandelt.name);
```

vater:Superklasse  
name=„Super“

sohn:Subklasse  
name=„Sub“

**Superklasse**

+ name:String  
+ toString():String

**Subklasse**

+ name:String  
+ toString():String

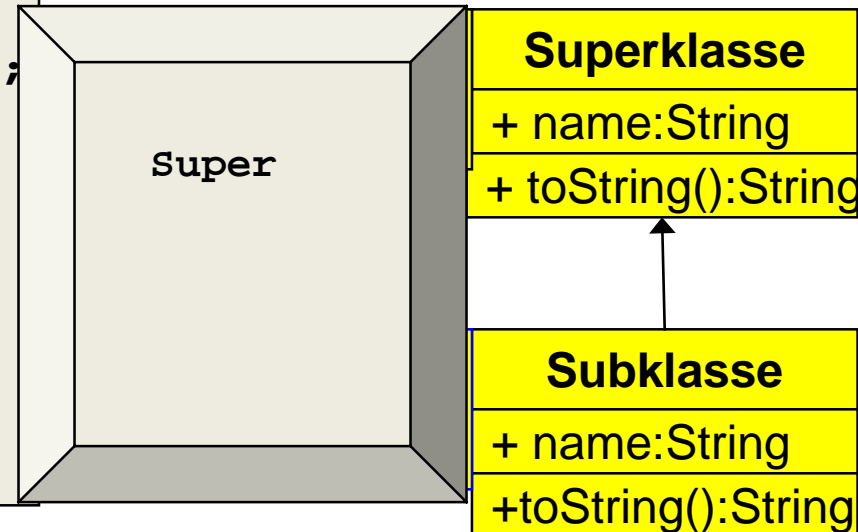


Wir versuchen zuerst, die durch **vater**, **sohn** und **gewandelt** referenzierten Objekte durch die Methode **toString** auf den Bildschirm zu bringen.

Im Falle des als vater bezeichneten Objektes ist das Ergebnis relativ einfach vorherzusehen.

Die Instanzmethode **toString** liefert den in der Variablen **name** gespeicherten Wert (also „Super“) zurück.

```
Superklasse vater      =new Superklasse();  
Subklasse   sohn      =new Subklasse();  
Superklasse gewandelt=(Superklasse)sohn;  
  
System.out.println(vater);  
System.out.println(sohn);  
System.out.println(gewandelt);  
  
System.out.println(vater.name);  
System.out.println(sohn.name);  
System.out.println(gewandelt.name);
```



Auch die Ausgabe für **sohn** ist einfach nachzuvollziehen. Die Methode **toString** liefert den Inhalt von **this.name** zurück.

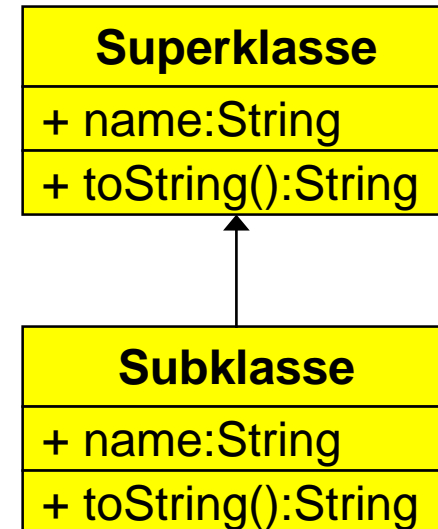
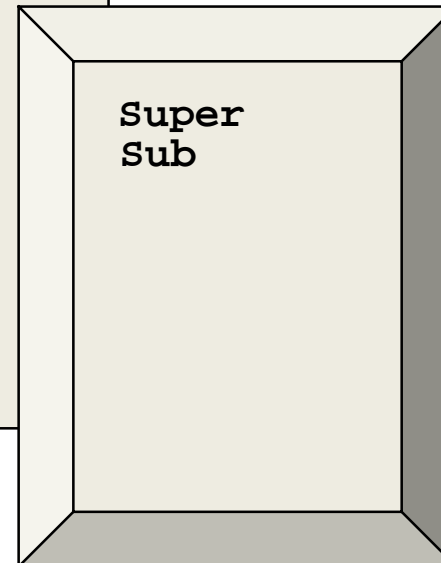
Hiermit ist natürlich eben jene Variable gemeint, die in derselben Klasse wie die Methode definiert wurde - also die der Subklasse.

Auf dem Bildschirm wird „Sub“ statt „Super“ ausgegeben.

```
Superklasse vater      =new Superklasse();  
Subklasse   sohn       =new Subklasse();  
Superklasse gewandelt=(Superklasse)sohn;
```

```
System.out.println(vater);  
System.out.println(sohn);  
System.out.println(gewandelt);
```

```
System.out.println(vater.name);  
System.out.println(sohn.name);  
System.out.println(gewandelt.name);
```



Das Ergebnis des Methodenaufrufes von `gewandelt.toString()` ist wieder auf das Prinzip des Polymorphismus zurückzuführen.

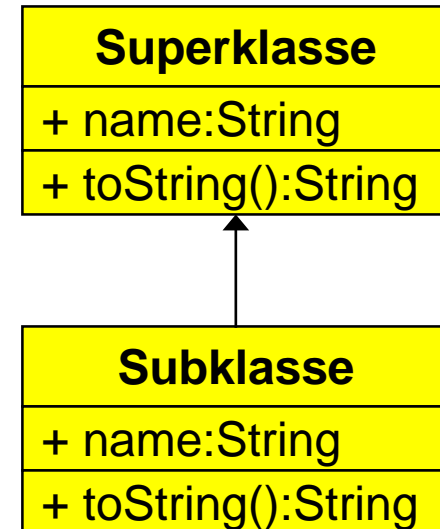
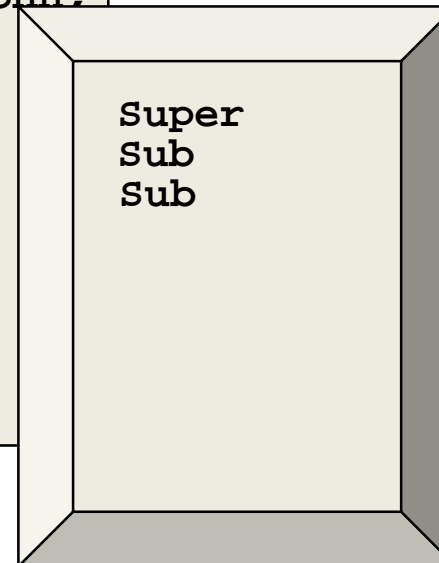
Obwohl das Objekt als Instanz der Superklasse angesehen wird, wird natürlich auch hier die **Methode der Subklasse aufgerufen**, welche die **originale `toString`-Methode überschreibt**.

Diese liefert dann erneut „Sub“ als Resultat.

```
Superklasse vater      =new Superklasse();  
Subklasse   sohn       =new Subklasse();  
Superklasse gewandelt=(Superklasse)sohn;
```

```
System.out.println(vater);  
System.out.println(sohn);  
System.out.println(gewandelt);
```

```
System.out.println(vater.name);  
System.out.println(sohn.name);  
System.out.println(gewandelt.name);
```



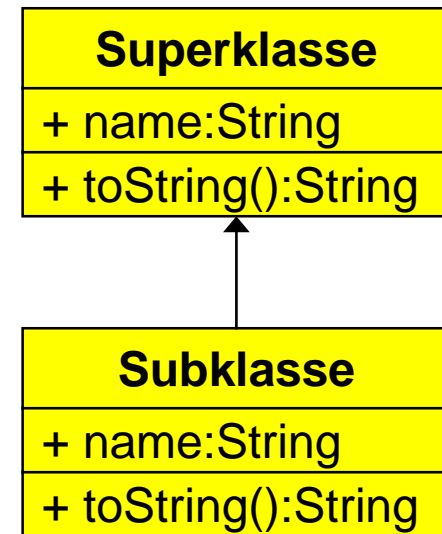
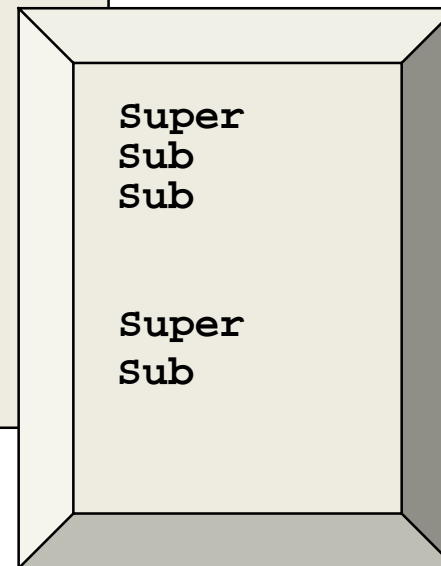
Wir werden nun den Inhalt der Variablen **name** direkt auf dem Bildschirm ausgeben - also ohne den Umweg über die **toString**-Methode.

Weder bei der Ausgabe von **vater.name** noch bei **sohn.name** erleben wir eine Überraschung. Der Mechanismus des Überschreibens scheint wie bei Methoden zu funktionieren.

```
Superklasse vater      =new Superklasse();  
Subklasse   sohn       =new Subklasse();  
Superklasse gewandelt=(Superklasse)sohn;
```

```
System.out.println(vater);  
System.out.println(sohn);  
System.out.println(gewandelt);
```

```
System.out.println(vater.name);  
System.out.println(sohn.name);  
System.out.println(gewandelt.name);
```



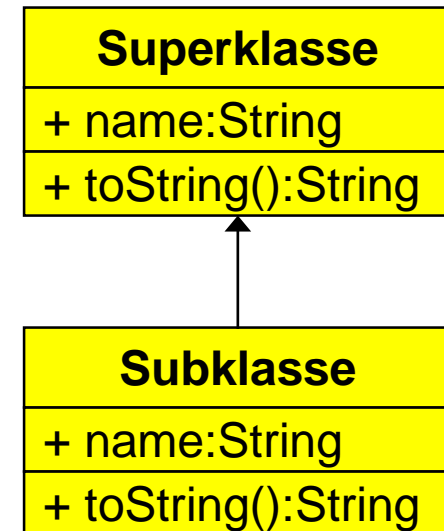
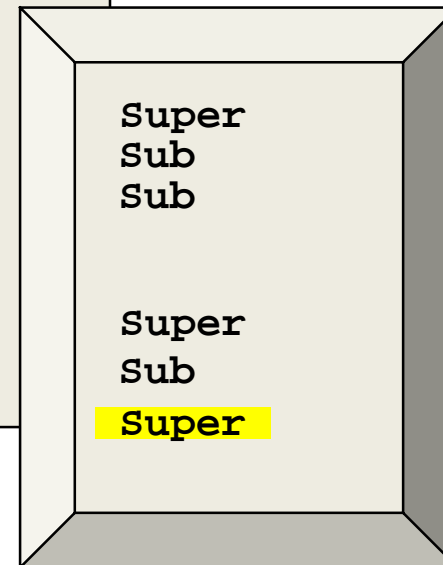
In der letzten Zeile jedoch erleben wir eine Überraschung: anstelle von „Sub“ wird „Super“ auf dem Bildschirm ausgegeben.

Der Grund hierfür liegt darin, dass Polymorphismus bei Variablen nicht anwendbar ist. Das **Objekt wird als Instanz der Superklasse betrachtet** - entsprechend erfolgt auch der Zugriff auf die Instanzvariable der Superklasse. Und in dieser ist nun einmal „Super“ abgespeichert...

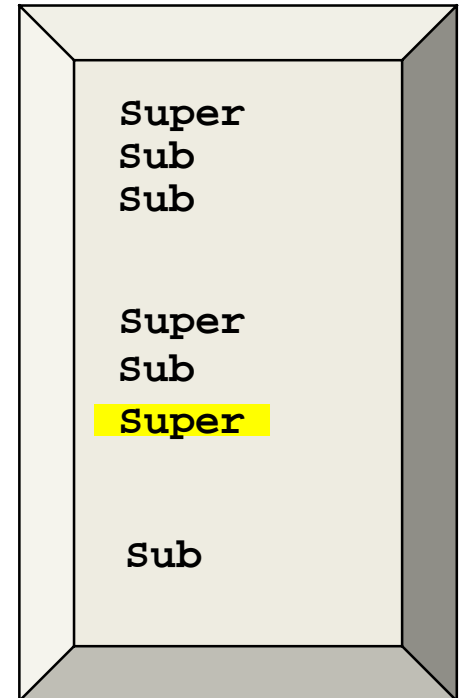
```
Superklasse vater      =new Superklasse();  
Subklasse   sohn       =new Subklasse();  
Superklasse gewandelt=(Superklasse)sohn;
```

```
System.out.println(vater);  
System.out.println(sohn);  
System.out.println(gewandelt);
```

```
System.out.println(vater.name);  
System.out.println(sohn.name);  
System.out.println(gewandelt.name);
```



```
Superklasse vater      = new Superklasse();  
Subklasse   sohn       = new Subklasse();  
Superklasse gewandelt = (Superklasse) sohn;  
  
System.out.println(vater);  
System.out.println(sohn);  
System.out.println(gewandelt);  
  
System.out.println(vater.name);  
System.out.println(sohn.name);  
System.out.println(gewandelt.name);  
  
Subklasse sohn2 = (Subklasse) gewandelt;  
System.out.println(sohn2.name);
```



## Fazit

**Auch Variablen können in Java überschrieben werden.**

**Das Prinzip des Polymorphismus ist auf diesen Fall allerdings nicht anwendbar!**

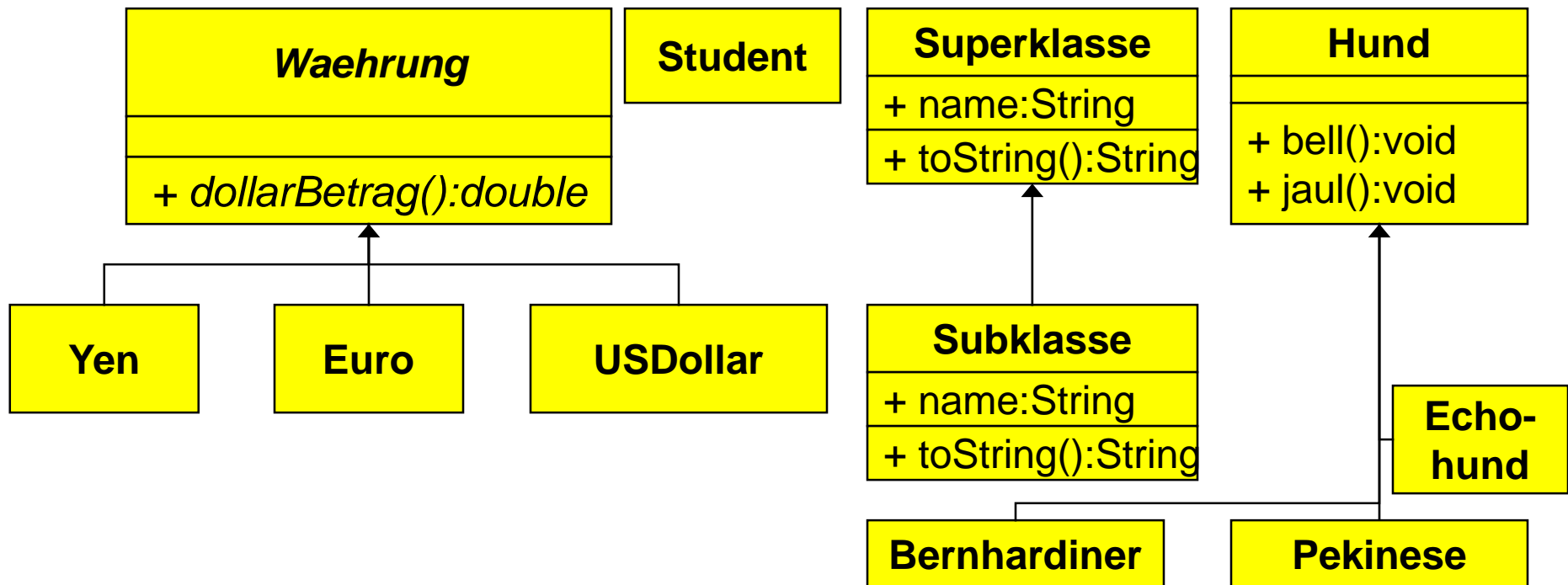
Ein Programmierer, der von dieser Möglichkeit also Gebrauch machen möchte, sollte sich immer des erhöhten Risikos bewusst sein, unbeabsichtigt Fehler in seinem Code zu produzieren.

Die Verwendung von Methoden zum Zugriff ([Datenkapselung](#)) ist immer noch die sicherste Methode!

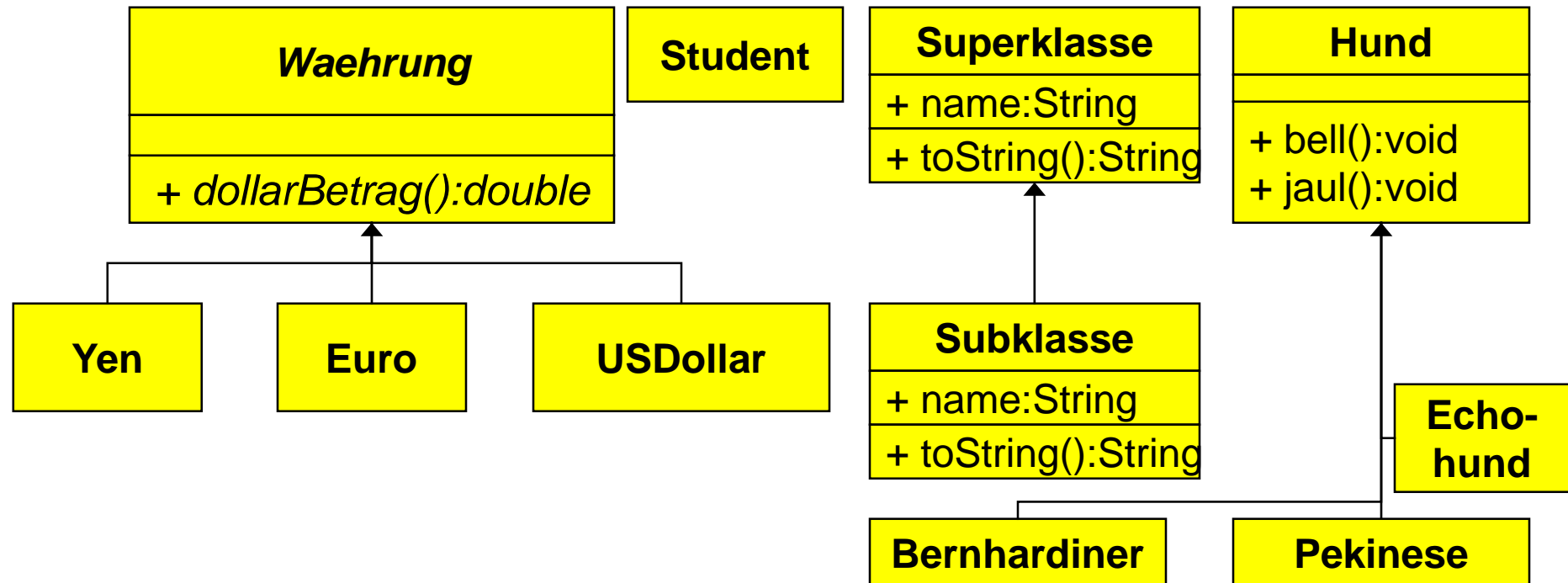


- Überschreiben von Methoden
- Typumwandlungen innerhalb einer Objekthierarchie
- Überschreiben von Variablen
- **Die Klasse `java.lang.Object`**
- Interfaces

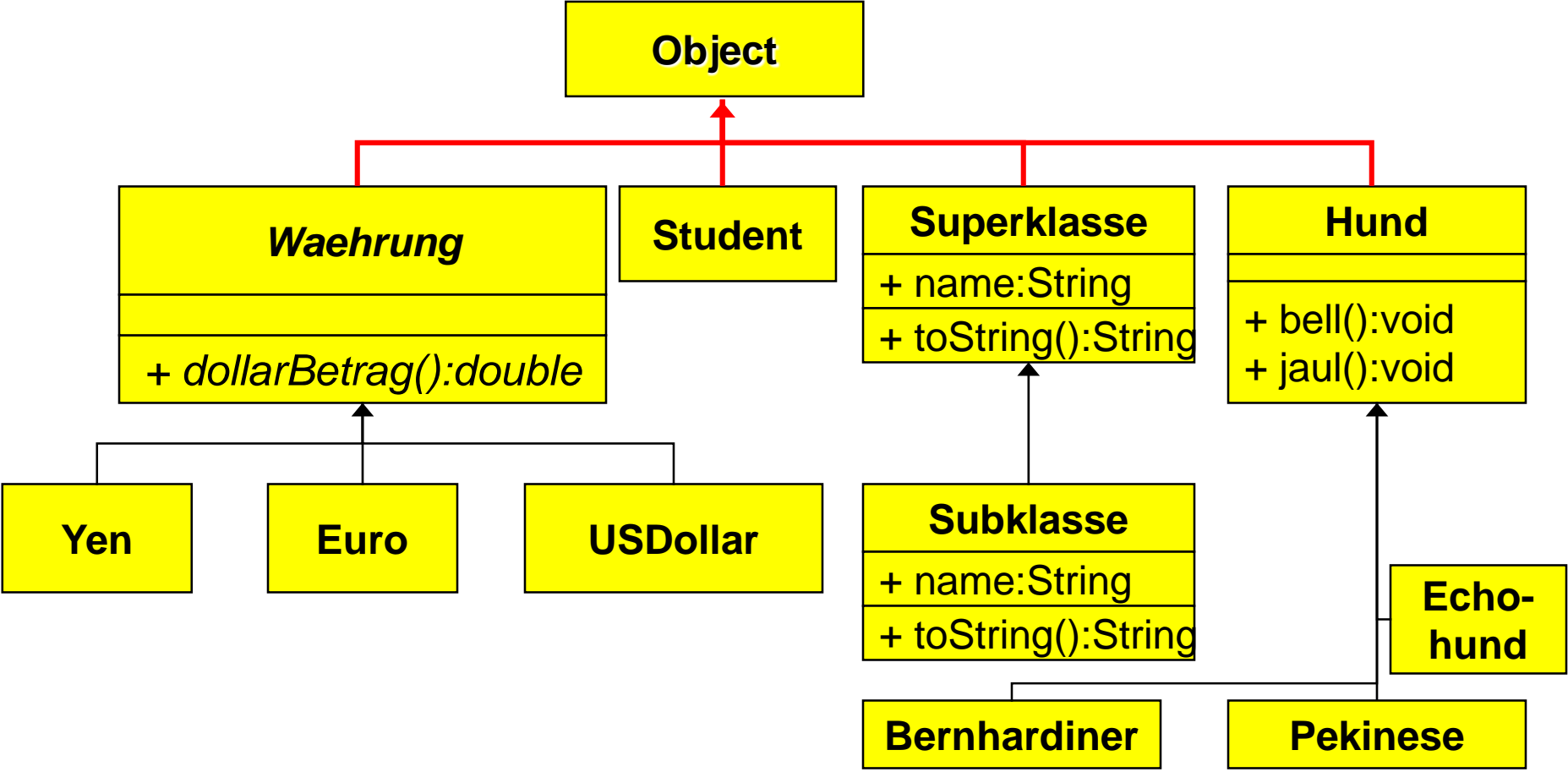
- Wir haben im Laufe der letzten Lektionen eine Vielzahl von verschiedenen Klassen geschaffen.
- Es stellt sich die Frage, ob diese verschiedenen Klassenhierarchien miteinander in Verbindung stehen.



- Es fällt an dieser Stelle natürlich schwer, eine Möglichkeit der Abstraktion zu finden. Welche Gemeinsamkeiten sollen etwa Studenten, Hunde und Zahlungsmittel aller Art besitzen?
- Tatsächlich gibt es jedoch eine Eigenschaft, die alle hier aufgeführten Klassen miteinander teilen: **es lassen sich **Objekte** aus ihnen bilden.**



- Diesem Umstand haben die Entwickler von Java Rechnung getragen und eine Klasse namens `Object` entwickelt. Jede wie auch immer gestaltete Klasse leitet sich von dieser ab.



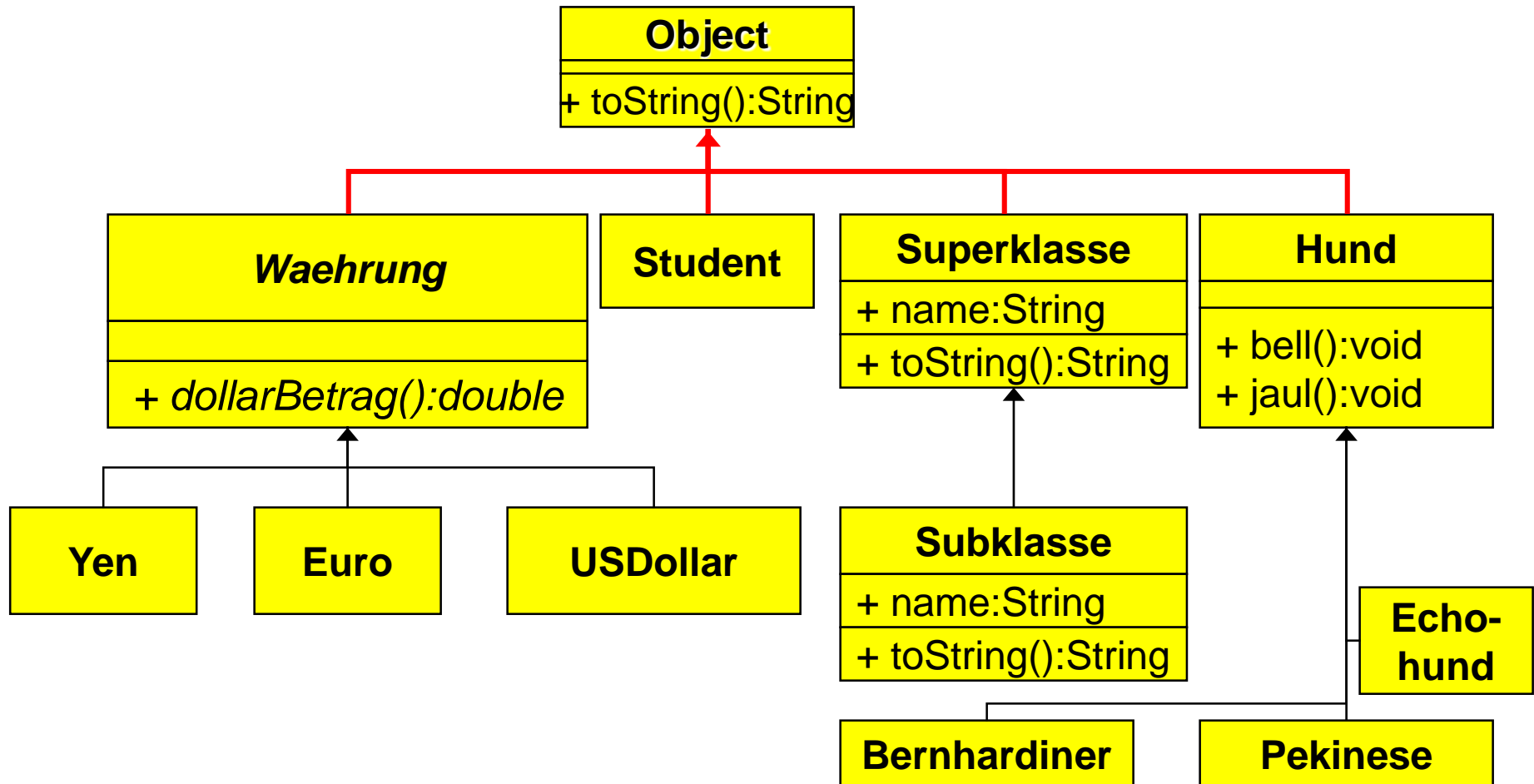
Welchen praktischen Sinn hat sie jedoch, die „**Mutter aller Klassen**“?

Um diesen Punkt zu verstehen, müssen wir begreifen, dass es gewisse Eigenschaften gibt, die alle Objekte miteinander teilen sollen. Hierbei handelt es sich hauptsächlich um die Existenz von gewissen Variablen und Methoden, die das System (aber auch der Benutzer) zum täglichen Umgang mit Objekten benötigt.

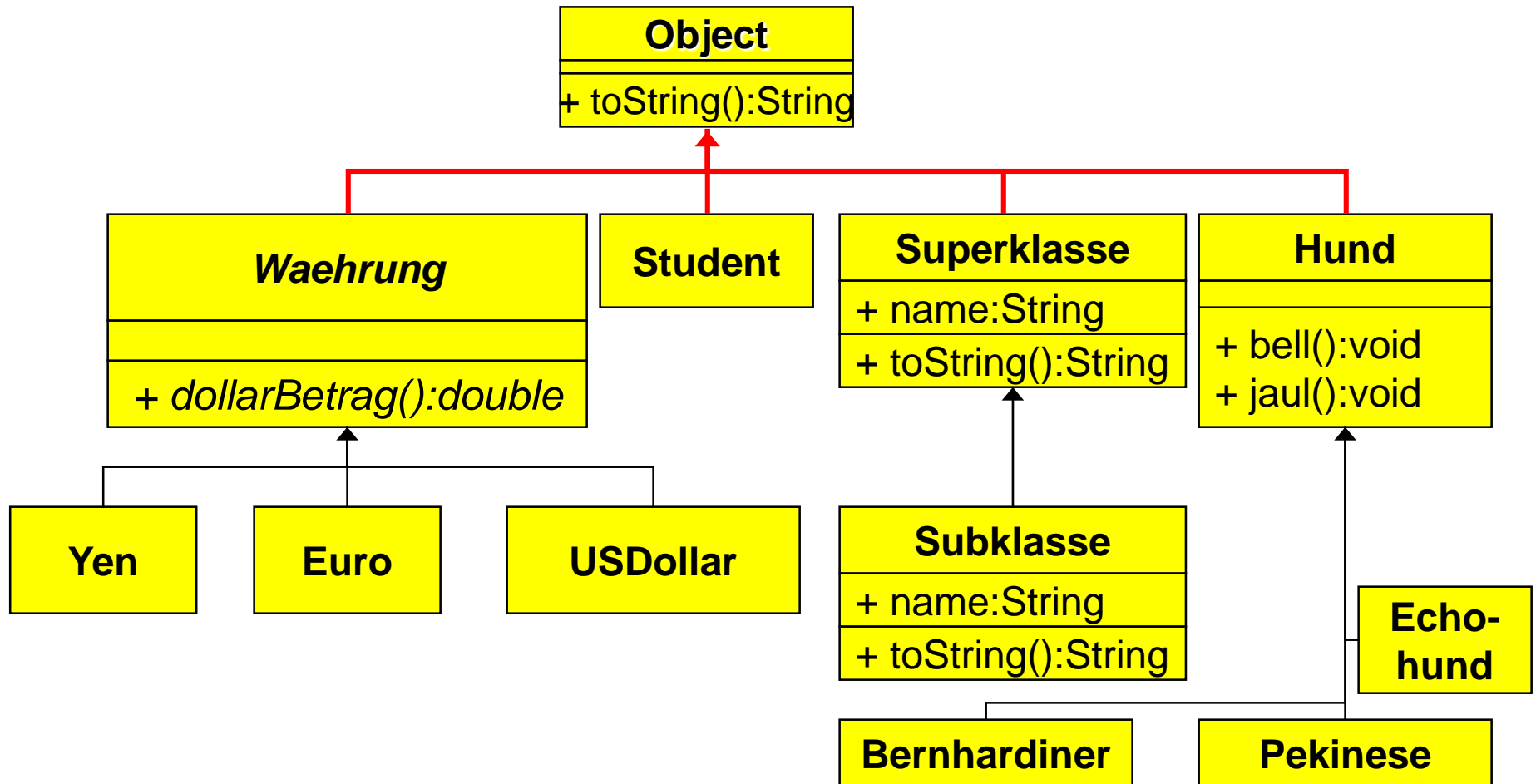
Ein gutes Beispiel hierfür ist die `toString`-Methode, mit der wir textuelle Darstellungen unserer Objekte erzeugen (etwa für unsere Bildschirmausgabe). Wenn wir Objekte mit dem Befehl `System.out.println` auf der Konsole ausgegeben haben, wurde von uns zu diesem Zweck immer eine `toString`-Methode geschaffen. Wir haben es als „Konvention“ betrachtet, die für die Sprache festgelegt wurde.

Der wahre Grund liegt jedoch in eben dieser Klasse **Object**.

Die Methode `toString` wurde bereits in der Klasse `Object` definiert. Aufgrund der Vererbung besitzt somit **jede** Klasse automatisch eine `toString`-Methode.



Wenn wir in einer Klasse also eine eigene Methode definieren, überschreiben wir hiermit lediglich die `toString`-Methode der Klasse `Object`.

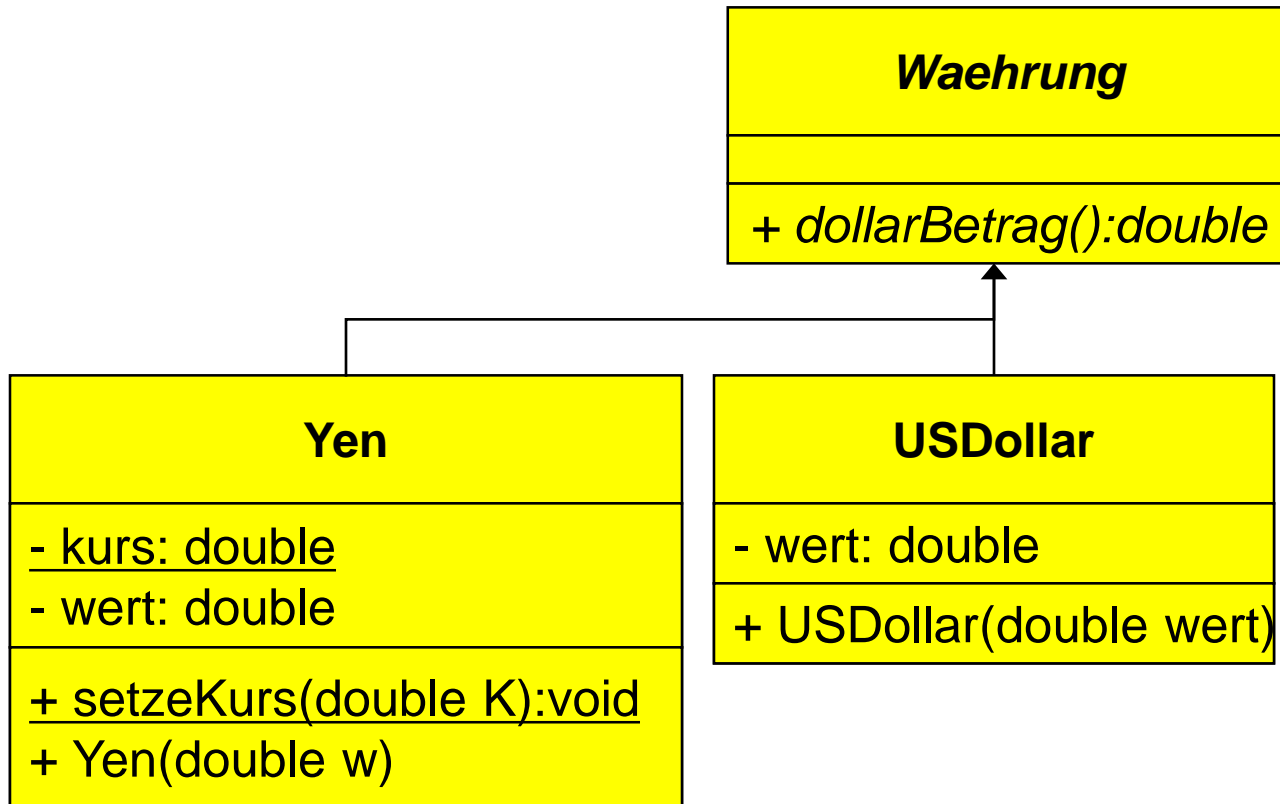


- Weitere Methoden, etwa zum „**Klonen**“ von **Objekten**, sind ebenfalls in der Klasse `java.lang.Object` definiert. Es empfiehlt sich, bei Bedarf einen Blick in die Dokumentation zu werfen.
- Die Existenz der Klasse `Object` wird in vielen Klassen ausgenutzt, die Problemlösungen für allgemeine Klassen zur Verfügung stellen. Hierzu zählen etwa Datenspeicher (Hashtabelle, Kellerspeicher, ...) und andere vielbenutzte Klassen des Pakets `java.util`.
- Wir werden aus Zeitgründen auf diesen Bereich jedoch hier nicht näher eingehen...

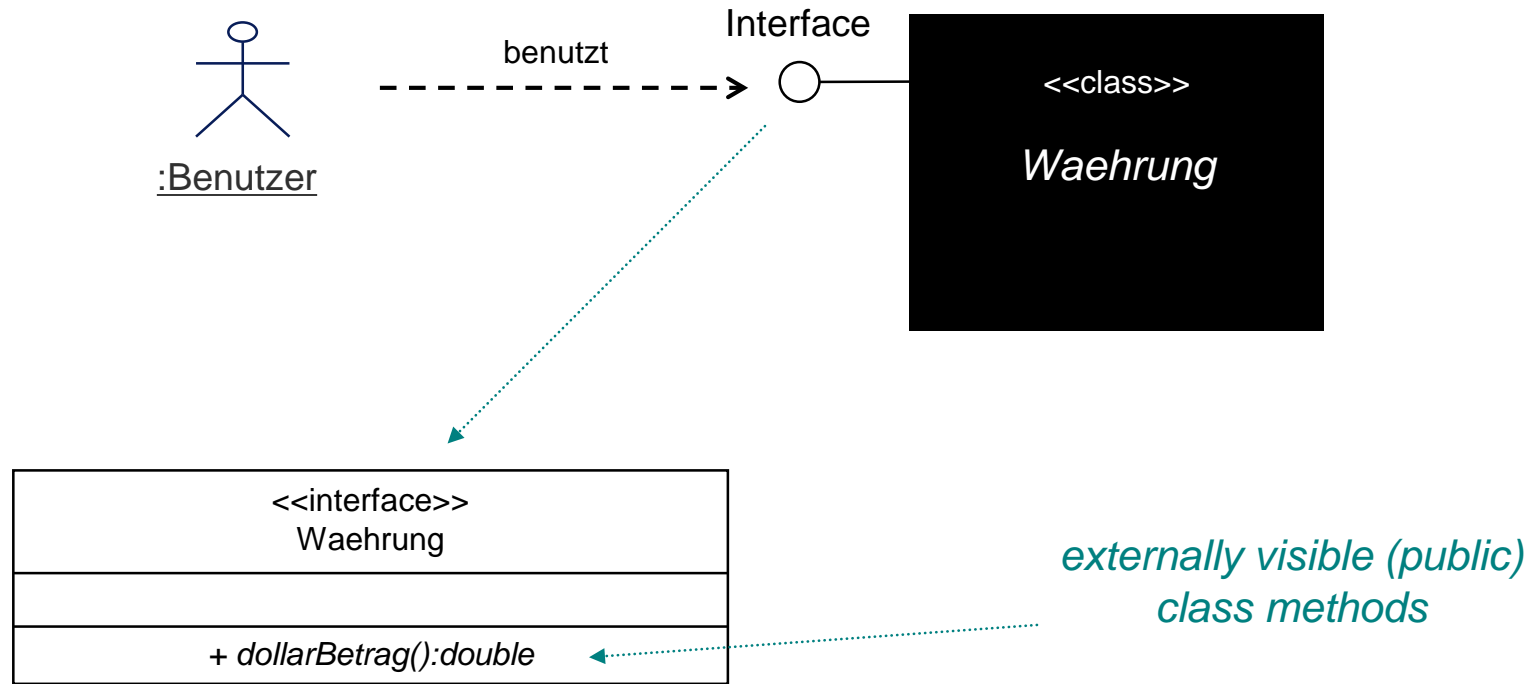


- Überschreiben von Methoden
- Typumwandlungen innerhalb einer Objekthierarchie
- Überschreiben von Variablen
- Die Klasse `java.lang.Object`
- **Interfaces**

# Beispiel von letzter Vorlesung



# Interface als allgemeines Konzept



- Alle konkreten Subklassen von *Waehrung* müssen eine Implementierung der Methode *dollarBetrag()* liefern
  - sie unterstützen (implementieren) damit die gleiche Schnittstelle
  - die entsprechende unterstützte Funktionalität kann einheitlich von außen aufgerufen werden ohne die internen Details der Klasse zu kennen

- Kann man in Java von mehreren Klassen erben?
- **Mehrfachvererbung gibt es leider nicht in Java!**
- Eine eingeschränkte Form existiert jedoch durch die Benutzung von Interfaces

## Beispiel

Neben dem Barvermögen gehören unserer Hotelkette noch verschiedene andere Wertgegenstände, z.B. Grundstücke, Aktien, Firmenbeteiligungen etc.

Jeder dieser Wertgegenstände besitzt **völlig unterschiedliche Eigenschaften**.

Deshalb ist es **schwer für alle eine allgemeine Klasse zu definieren**.

Trotzdem soll der Gegenwert eines Wertgegenstands in Dollar (oder Yen, Euro,...) bestimmt werden können.

Dazu wollen wir unserer Klasse eine Schnittstelle vorgeben.

Alle unsere Klassen sollen eine Methode `wert()` besitzen, die den aktuellen Wert (z.B. unserer Immobilie) in einer beliebigen Währung zurückgibt.

Wir formulieren unsere Anforderung als ein sogenanntes **Interface**:

```
/** Ein beliebiger Wertgegenstand */  
public interface Wertgegenstand {  
    /** Gibt den Wert des Objekts als Waehrung zurueck */  
    public Waehrung wert();  
}
```

Alle unsere Klassen sollen eine Methode `wert()` besitzen, die den aktuellen Wert (z.B. unserer Immobilie) in einer beliebigen Währung zurückgibt.

Wir formulieren unsere Anforderung als ein sogenanntes **Interface**:

```
/** Ein beliebiger Wertgegenstand */  
public interface Wertgegenstand {  
    /** Gibt den Wert des Objekts als Waehrung zurueck */  
    public Waehrung wert();  
}
```

Interfaces sind im Gegensatz zu abstrakten Klassen keine Klassen im eigentlichen Sinne. Es existieren keine Kindklassen, die ein Interface mit Hilfe des Schlüsselwortes **extends** beerben. Das entsprechende Wort für Interfaces heißt **implements**.

```
/** Ein Goldbarren (=Wertgegenstand) */
public class Goldbarren implements Wertgegenstand {
    /**Wie viel ist Gold heutzutage eigentlich wert? */
    public static double preisProGrammInDollar=60;

    /**Das Gewicht des Barrens */
    private double gewicht;

    /** Konstruktor - das Gewicht ist in Gramm anzugeben*/
    public Goldbarren(double gewichtInGramm){
        gewicht = gewichtInGramm;
    }

    /**Implementierung des Interfaces*/
    public Waehrung wert() {
        return new USDollar(gewicht*preisProGrammInDollar);
    }
}
```

**Beispiel**



## Beispiel

Die Klasse `Goldbarren` setzt das Interface `Wertgegenstand` in einer Klasse um und macht dies dem Compiler durch die Worte `implements Wertgegenstand` klar.

Um nun dabei eine gültige Klassendefinition zu erzeugen **müssen** wir eine entsprechende Methode `wert` definieren (sonst erhalten wir eine Fehlermeldung beim Übersetzen).

Hierbei errechnen wir den Wert unseres Barrens in Dollar aus dem Gewicht und geben diesen als `USDollar`-Objekt zurück.

Dieses Vorgehen ist erlaubt, da `USDollar` Subklasse von `Waehrung` ist.

## Beispiel

```
/** Berechne den Gesamtwert einer Menge von Wertgegenstaenden */  
public static Waehrung gesamtwert(Wertgegenstand[] objekte){  
    double summe=0;  
    for (Wertgegenstand w : objekte){  
        summe += w.wert().dollarBetrag();  
    }  
    return new USDollar(summe);  
}
```

In der Schleife werden die verschiedenen Geldbeträge über die Methode `wert()` ausgelesen.

Weil das Resultat dieser Methode jeweils ein Währungsobjekt ist, muss der Dollarbetrag noch über die Methode `dollarBetrag()` ausgelesen werden.

# Interfaces

- Sofern die Objekte das Interface implementieren, können sie als Wertgegenstand aufgefasst werden.
- **Unterschied** zu abstrakten Klassen: **Die Methoden im Interface müssen alle abstrakt sein.**
- Damit bieten sich Interfaces als Schnittstellenvorgabe an.
- Im Gegensatz zu normalen Klassen **erlauben Interfaces** jedoch die Möglichkeit von **Mehrfachvererbung**.
- **Eine Klasse darf nämlich zwar nur eine Superklasse besitzen; sie darf aber beliebig viele Interfaces implementieren!**

## Beispiel

```
/** Das beruehmte goldene Zahlungsmittel*/  
public class Kruegerrand extends Waehrung implements Wertgegenstand  
{  
    /**Ein Kruegerrand ist soviel Dollar wert */  
    private static double kurs;  
  
    /** Instanzvariable: Wert in Kruegerrand */  
    private double wert;  
  
    /** Konstruktor */  
    public Kruegerrand(double wert) {  
        this.wert=wert;  
    }  
    ...  
}
```

# Beispiel

...

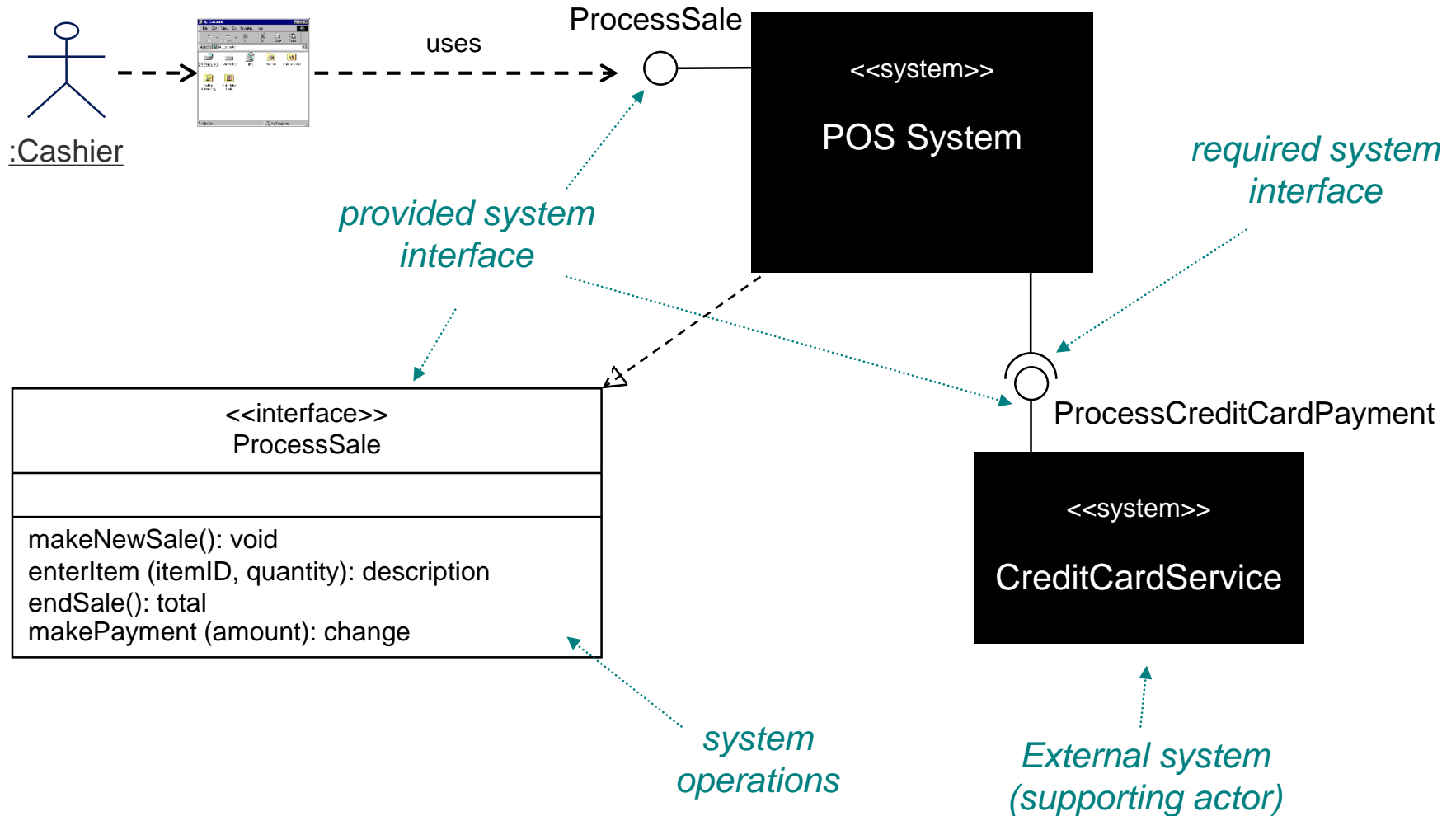
```
/** Deklaration der sonst abstrakten Methode dollarBetrag */
public double dollarBetrag(){
    return wert*kurs;
}

/**Zugriff auf die private Klassenvariable */
public static void setKurs(double k){
    kurs = k;
}

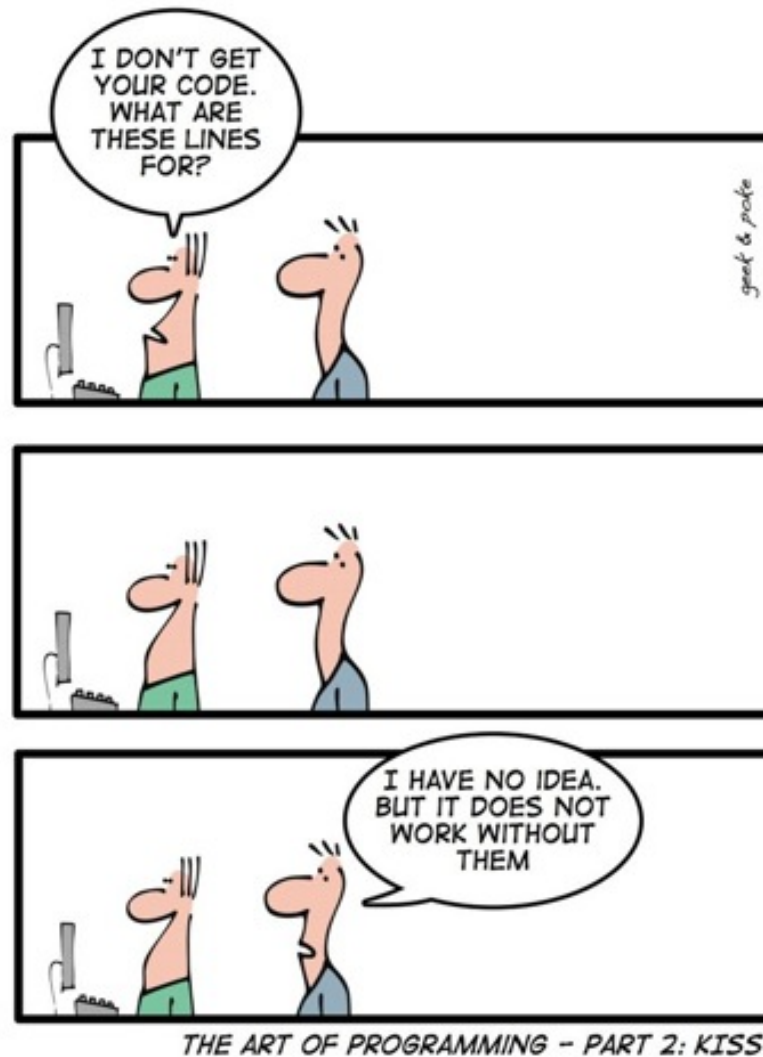
/**Implementierung des Interfaces:
das Objekt ist selbst schon Waehrung*/
public Waehrung wert() {
    return this;
}

}
```

# System Interface Example



# Fragen?



# Danksagung

- Vorlesungsmaterialien von Prof. Dr. Detlef Seese wurden als Basis verwendet
- Unterstützung bei der technischen und inhaltlichen Gestaltung des Vorlesungsmaterials leisteten:

Jóakim v. Kistowski

Dietmar Ratz, Joachim Melcher, Roland Küstermann, Jana Weiner, Hagen Buchwald, Matthes Elstermann, Oliver Schöll, Niklas Kühl, Tobias Diederich