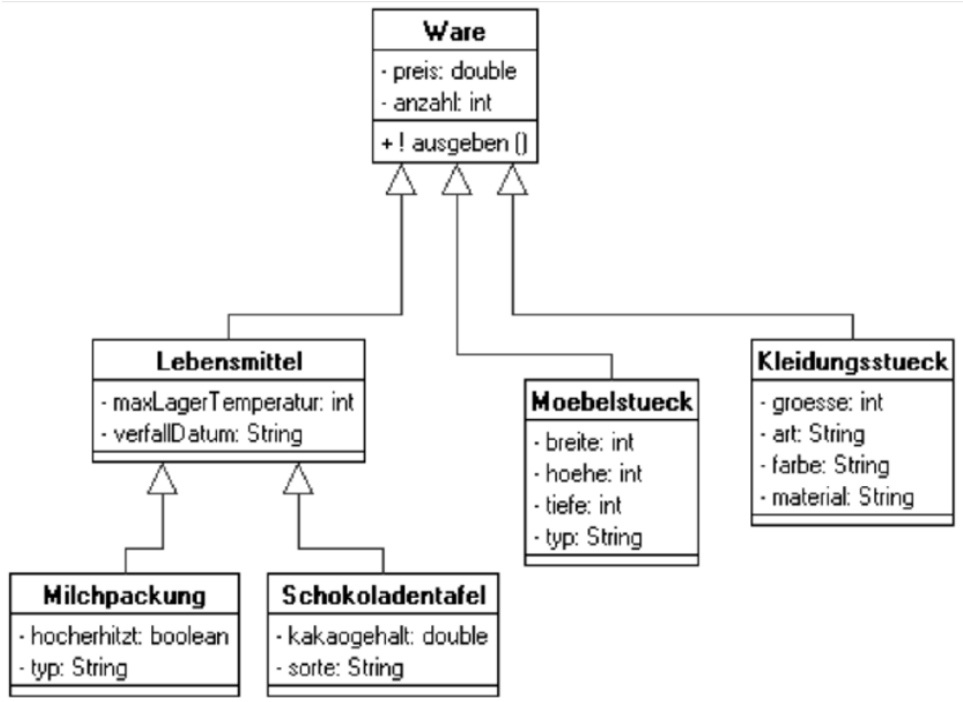
Polymorphie

3. Prinzip der OOP

(nach Kapselung und Vererbung)!

Variablen vom Typ einer Oberklasse, die Objekte verschiedenen Typs – sowohl der Klasse selber als auch jeder beliebigen Unterklasse – aufnehmen können, werden in der objektorientierten Programmierung als **polymorph** (deutsch: **vielgestaltig**) bezeichnet.



Beispiel:

Gegeben seien die Klassen „Ware“ und „Moebelstueck“, wobei „Moebelstueck“ als Unterklasse von „Ware“ definiert sei (siehe Abbildung).

In dem Zusammenhang deklarieren wir die folgenden beiden Anweisungen:

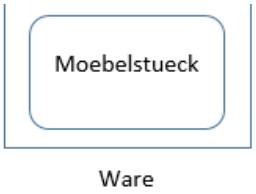
```
Ware ding ;
Moebelstueck moeb = new Moebelstueck ( ) ;
```

Die Variable „ding“ ist vom Typ der Klasse „Ware“, in der Variable „moeb“ ist ein Objekt der Klasse „Moebelstueck“ enthalten. Dieser Umstand ermöglicht uns nun die folgende Zuweisung:

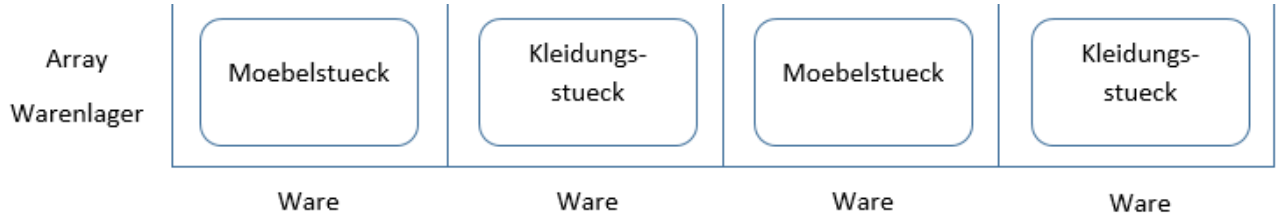
```
ding = moeb;
```

Wir sehen, dass die Variable „ding“ auch ein Objekt vom Typ „Moebelstueck“ aufnehmen kann – dies funktioniert, weil jedes Möbelstück eine Ware ist (und in Java jede Kindsklasse auch alle Methoden und Attribute ihrer Elternklassen implementieren muss).

Die Variable „ding“ ist wie ein Behälter, in der die Ware Möbelstück hineingelegt werden kann. Ganz allgemein kann „ding“ alle Objekte aufnehmen, die vom Typ einer Unterklasse von „Ware“ sind, z.B. auch Objekte einer (hier noch unbekannten) Klasse „Kleidungsstueck“.



In Anwendung kann man nun z.B. einen Array vom Typ „Ware“ anlegen, welcher nicht nur Objekte der Klasse „Ware“ aufnehmen kann, sondern zusätzlich alle möglichen Objekte vom Typ einer beliebigen Unterklasse von „Ware“ – wie z.B. „Moebelstueck“ oder „Kleidungsstueck“. Ein solcher Array ist wie eine Aneinanderreihung von Ware-Behältern, in denen sämtliche Waren hineingelegt werden können.



Besitzen in diesem Zusammenhang zwei Klassen (Ober- und Unterklasse) Methoden mit identischen Namen, jedoch unterschiedlicher Implementierung, so heißen diese Methoden ebenfalls **polymorph**. Man sagt auch, dass die Methode der Oberklasse „überschrieben“ wird.

Kommen wir nun zu einem anderen, sehr einfachen Beispiel (in Java):

```
abstract class Stift {
   String schreibfarbe;

   public void SchreibWas() {
      System.out.println("Ich bin ein Stift. ");
   }

   public abstract void gibSchreibfarbeAus();
}

class Kugelschreiber extends Stift {
   public Kugelschreiber (String e) {
      schreibfarbe = e;
   }

   public void SchreibWas() {
      System.out.println("Ich bin ein Kugelschreiber.");
   }

   public void gibSchreibfarbeAus() {
      System.out.println("Die Schreibfarbe des
      Kugelschreibers ist " + schreibfarbe + ".");
   }
}

class Buntstift extends Stift {
   public Buntstift (String e) {
      schreibfarbe = e;
   }

   public void gibSchreibfarbeAus() {
      System.out.println("Mein Buntstift hat die
      Schreibfarbe " + schreibfarbe + ".");
   }
}

public class StiftTester {
   public static void main ( String[] args ) {
      Stift irgendeinStift = new Kugelschreiber("blau");
      irgendeinStift.SchreibWas();
      irgendeinStift.gibSchreibfarbeAus();
      irgendeinStift = new Buntstift ("rot");
      irgendeinStift.SchreibWas();
      irgendeinStift.gibSchreibfarbeAus();
   }
}
```

Hier werden einer Variablen, die auf ein Objekt der Elternklasse "Stift" zeigt, Objekte der Unterklassen "Kugelschreiber" und "Buntstift" zugewiesen. Da alle Klassen die Methoden „SchreibWas“ bzw. "gibSchreibfarbeAus" besitzen, ist der Aufruf dieser Methoden zu keinem Zeitpunkt ein Problem. Die JVM weiß, welche Klasse sich hinter dem Objekt der Variablen "irgendeinStift" verbirgt – so wird stets die entsprechende Methode der jeweiligen (Unter-)Klasse aufgerufen.
(Mit dem Aufruf von "irgendeinStift.getClass()" kann man sich im Übrigen die Klasse des Objektes „anschauen“, auf welches die Variable "irgendeinStift" zeigt!)