- die Klasse **Ware** hat nur drei Attribute:

```
public class Ware
{
  private int warenNummer;
  private String warenName;
  private Preis preis;

  // Konstruktoren
  // Methoden
}
```

  - die Klasse **Preis** stammt aus unserem früheren Beispiel
- für verschiedene Situationen sehen wir drei Konstruktoren vor





- dem ersten Konstruktor ist ein Objekt vom Typ **Preis** zu übergeben

```
public Ware(int dieWarenNummer,
            String derWarenName,
            Preis derPreis)
{
  boolean inputOk = (0 <= dieWarenNummer &&
                     dieWarenNummer <= Integer.MAX_VALUE) &&
                    (derWarenName != null) &&
                    (derPreis != null);
  if (inputOk)
  {
    warenNummer = dieWarenNummer;
    warenName = derWarenName;
    preis = derPreis;
  }
  else
    throw new IllegalArgumentException();
}
```





- der zweite Konstruktor zeigt erneut, wie ein Konstruktor einer Klasse mit Hilfe von **this** einen anderen Konstruktor derselben Klasse aktiviert

```
public Ware(int dieWarenNummer,
            String derWarenName,
            double derPreis)
{
  this(dieWarenNummer, derWarenName, new Preis(derPreis));
}
```





- beim dritten Konstruktor handelt es sich um einen so genannten copy constructor
  - er dient dazu, eine tiefe Kopie zu erzeugen
  - dazu benutzt er u.a. eine (noch zu deklarierende) Methode der Klasse

```
public Ware(Ware eineWare)
{
  if (eineWare != null)
  {
    warenNummer = eineWare.warenNummer;
    warenName = eineWare.warenName;
    preis = new Preis(eineWare.lieferePreis());
  }
  else
    throw new IllegalArgumentException();
}
```

- das API der Klasse **Ware** besteht aus acht Methoden
  - zwei davon liefern auf kanonische Weise den Warennamen und die Warennummer

```
public int liefereNummer()
{
  return warenNummer;
}

public String liefereNamen()
{
  return warenName;
}
```

  - drei weitere zeigen, wie Experten vom Typ **Ware** Aufgaben an Experten vom Typ **Preis** delegieren





```
public double liefere Preis()
{
  return preis.liefereBruttoPreis();
}

public double liefereNettoPreis()
{
  return preis.liefereNettoPreis();
}

public double liefereMehrwertSteuerAnteil()
{
  return preis.liefereMehrwertSteuerAnteil();
}
```

- **Hinweis:** die Delegation von Aufgaben zählt zu den wichtigsten Prinzipien objektorientierter Programmierung





  - eine Methode gestattet es, den Preis nachträglich zu ändern

```
public void aenderePreis(double derNeuePreis)
{
  if (derNeuePreis >= 0.0)
    preis = new Preis(derNeuePreis);
}
```

  - zwei weitere Methoden sind eher untypisch:
    - wir wollen Informationen über die Ware in einem tabellenähnlichen Format ausgeben
    - dabei soll die (Feld)Breite der Spalte, die die Informationen enthält, variabel festlegbar sein
    - die folgende Methode liefert die Informationen als geeignet formatierte Zeichenkette





```
public String liefereWarenInfo(int feldbreite)
{
  String info = String.format("%-10s%2s%" + feldbreite + "s\n",
                              "Name", ":", warenName);
  info += String.format("%-10s%2s%" + feldbreite + "d\n",
                              "Nummer", ":",  warenNummer);
  info += String.format("%-10s%2s%" + feldbreite + ".2f\n",
                              "Preis", ":", liefere Preis());
  return info;
}
```

- passend dazu liefert die letzte Methode die minimale (Feld)Breite, die erforderlich ist, um die Attribute aufzunehmen

```
public int liefereAttributTextLaenge()
{
  int laenge = Math.max(("" + warenNummer).length(),
                        warenName.length());
  return Math.max(laenge, ("" + liefere Preis()).length());
}
```

- wir testen die Klasse **`Ware`** im folgenden Programm

```
public class WareMain
{
  public static void main(String[] args)
  {
    Ware[] warenkorb =
                  { new Ware(781503, "LCD-Fernseher", 1789.23),
                    new Ware(100212, "Ananas", 2.99) };
    schreibe(warenkorb);
  }

  private static void schreibe(Ware[] warenkorb)
  {
    int feldbreite = ermitteleFeldbreite(warenkorb) + 2;
    for (Ware ware : warenkorb)
      System.out.println(ware.liefereWarenInfo(feldbreite));
  }
```





```
  private static int ermitteleFeldbreite(Ware[] warenkorb)
  {
    int feldbreite = warenkorb[0].liefereAttributTextLaenge();
    for (Ware ware : warenkorb)
      if (ware.liefereAttributTextLaenge() > feldbreite)
        feldbreite = ware.liefereAttributTextLaenge();
    return feldbreite;
  }
}
```

**es führt zu dieser Ausgabe:**

```
Name        :  LCD-Fernseher
Nummer      :        781503
Preis       :       1789,23

Name        :        Ananas
Nummer      :        100212
Preis       :          2,99
```





- Waren, die durch einen Hersteller produziert werden, sind spezielle Waren
  - sie haben alle Eigenschaften, die eine allgemeine Ware hat
  - sie können an allen Stellen (zum Beispiel eines Warenwirtschaftssystems) benutzt werden, wo allgemeine Waren auftreten können
  - daher wird die Klasse **`ProduzierteWare`** von der Klasse **`Ware`** abgeleitet
    - sie erhält (der Einfachheit halber nur) ein zusätzliches Attribut **`herstellerName`**
    - damit von außen auf dieses Attribut zugegriffen werden kann, wird eine neue Methode **`liefereHersteller`** hinzugefügt





```
public class ProduzierteWare extends Ware
{
  private String herstellerName;

  public ProduzierteWare(Ware dieWare, String derHersteller)
  {
    super(dieWare);
    if (derHersteller != null)
      herstellerName = derHersteller;
    else
      throw new IllegalArgumentException();
  }

  public ProduzierteWare(int dieWarenNummer, String derWarenName,
                         double derPreis, String derHersteller)
  {
    super(dieWarenNummer, derWarenName, derPreis);
    if (derHersteller != null)
      herstellerName = derHersteller;
    else
      throw new IllegalArgumentException();
  }
```

```
  public String liefereHersteller()
  {
    return herstellerName;
  }

  public String liefereWarenInfo(int feldbreite)
  {
    return super.liefereWarenInfo(feldbreite) +
           String.format("%-10s%2s%" + feldbreite + "s\n",
                         "Hersteller", ":",  herstellerName);
  }

  public int liefereAttributTextLaenge()
  {
    return Math.max(super.liefereAttributTextLaenge(),
                    herstellerName.length());
  }
}
```





- die Methoden **`liefereWarenInfo`** und **`liefereAttributTextLaenge`** waren schon in der Basisklasse vorhanden
  - weil aber bei einer produzierten Ware mehr Informationen anfallen als bei einer allgemeinen Ware, wurden sie in der abgeleiteten Klasse neu geschrieben, um sie der neuen Situation anzupassen
- das folgende Testprogramm zeigt, wie die zusätzliche Information ausgegeben wird:





```
public class ProduzierteWareMain
{
  public static void main(String[] args)
  {
    ProduzierteWare[] warenkorb =
          { new ProduzierteWare(238934, "Mac Pro One",
                                2179.50, "Apple"),
            new ProduzierteWare(new Ware(781503,
                                         "LED-Fernseher",
                                         1059.00),
                                "Panasonic")};
    schreibe(warenkorb);
  }

  private static void schreibe(ProduzierteWare[] warenkorb)
  {
    int feldbreite = ermitteleFeldbreite(warenkorb) + 2;
    for (Ware ware : warenkorb)
      System.out.println(ware.liefereWarenInfo(feldbreite));
  }
```





```
  private static int ermitteleFeldbreite(ProduzierteWare[] korb)
  {
    int feldbreite = korb[0].liefereAttributTextLaenge();
    for (Ware ware : korb)
      if (ware.liefereAttributTextLaenge() > feldbreite)
        feldbreite = ware.liefereAttributTextLaenge();
    return feldbreite;
  }
}
```

**es führt zu dieser Ausgabe:**

```
Name        :    Mac Pro One
Nummer      :         238934
Preis       :        2179,50
Hersteller :          Apple

Name        :  LED-Fernseher
Nummer      :         781503
Preis       :        1059,00
Hersteller :      Panasonic
```

- das Vorgehen, eine Methode der Basisklasse in der abgeleiteten Klasse neu zu schreiben, um sie der speziellen Situation anzupassen, nennt man Überschreiben der Methode („overriding“)
  - die Klasse `ProduzierteWare` überschreibt zwei Methoden der Klasse `Ware` und fügt ihr außerdem eine neue Methode hinzu
  - diese Situation ist bei Vererbung typisch
- **Achtung:** ein Überschreiben liegt nur dann vor, wenn die Signatur der Methoden in der abgeleiteten Klasse und der Basisklasse übereinstimmen





- zusätzlich muss berücksichtigt werden:
  - wenn die überschriebene Methode der Basisklasse keinen Rückgabewert hat, darf auch die überschreibende Methode der abgeleiteten Klasse keinen Rückgabewert haben
  - wenn der Datentyp des Rückgabewertes der überschriebenen Methode ein primitiver Datentyp ist, muss der Rückgabewert der überschreibenden Methode denselben Datentyp haben
  - wenn der Datentyp des Rückgabewertes der überschriebenen Methode ein Referenzdatentyp ist, reicht es, wenn der Datentyp des Rückgabewertes der überschreibenden Methode zuweisungskompatibel zu diesem ist
- ansonsten kommt es bereits beim Übersetzen zu einem Fehler





- eine Methode der Basisklasse kann nur dann überschrieben werden, wenn sie `public`, `protected` oder „friendly“ ist
  - Methoden, die `private` sind, stehen ja nur innerhalb der Basisklasse zur Verfügung
- beim Überschreiben einer Methode kann ihr Zugriffsschutz nie verschärft, aber wie folgt verringert werden

| Methode (Basisklasse) | Methode (abgeleitete Klasse) |
|---|---|
| „friendly“ | „friendly“, `protected`, `public` |
| `protected` | `protected`, `public` |
| `public` | `public` |





- eine Methode der Basisklasse, die `static` ist, kann nur durch eine Methode überschrieben werden, die ebenfalls `static` ist
- wenn eine überschreibende Methode `static` ist, muss umgekehrt auch die überschriebene Methode `static` sein
- bezüglich der sonstigen Methodenmodifizierer `synchronized`, `native` und `strictfp` gibt es keine Einschränkungen
- das trifft auch für Annotations zu

- jetzt können wir auch erklären, wozu der sonstige Methodenmodifizierer **`final`** dient:
  - eine als **`final`** deklarierte Methode kann nicht überschrieben werden
  - dadurch ist es möglich, einzelne Methoden, die als „endgültig“ angesehen werden, vor dem Überschreiben zu schützen
- selbstverständlich kann eine überschreibende Methode als **`final`** deklariert werden, obwohl die überschriebene Methode nicht **`final`** sein kann





- die **`throws`**-Klausel einer überschreibenden Methode kann verschieden sein von der **`throws`**-Klausel der überschriebenen Methode
  - darauf gehen wir später genauer ein
- außerdem ist beim Überschreiben von Methoden noch zu beachten:
  - die Parameter der überschriebenen und der überschreibenden Methode können völlig unabhängig voneinander als **`final`** deklariert werden
  - wenn die überschriebene Methode einen Parameter mit Ellipse hat, kann er in der überschreibenden Methode ersetzt werden durch einen Parameter mit passendem Arraydatentyp





- wir betrachten noch einmal folgende Zeilen unseres Beispiels:

```
for (Ware ware : warenkorb)
  System.out.println(ware.liefereWarenInfo(feldbreite));
```

- die Ausgabe hat gezeigt, dass die überschreibende Methode der abgeleiteten Klasse ausgeführt wird
- die überschriebene Methode der Basisklasse ist zwar im Objekt vorhanden, aber nach außen nicht sichtbar
  - es handelt sich um einen Namenskonflikt, der wie üblich dadurch gelöst wird, dass das Lokale (hier: Spezielle) das Globale (hier: Allgemeinere) verdeckt





- innerhalb der abgeleiteten Klasse kann aber mit Hilfe von **`super`** auf die verdeckte Methode zugegriffen werden

```
public String liefereWarenInfo(int feldbreite)
{
  return super.liefereWarenInfo(feldbreite) +
         String.format("%-10s%2s%" + feldbreite + "s\n",
                       "Hersteller", ":",  herstellerName);
}
```

- eine Methode **`machWas`** der Basisklasse wird sehr oft nach diesem Muster überschrieben:

```
public void machWas()
{
  super.machWas();   // mache es für das Teilobjekt der Basisklasse
  // mache das Zusätzliche für die abgeleitete Klasse
}
```

- soll die abgeleitete Klasse sowohl die überschreibende Methode als auch die überschriebene Methode nach außen zur Verfügung stellen, kann für eine zusätzliche Methode der folgenden Art gesorgt werden

```
public void machDasAllgemeine()
{
  super.machWas();   // mache es für das Teilobjekt der Basisklasse
}

public void machWas()
{
  super.machWas();   // mache es für das Teilobjekt der Basisklasse
  // mache das Zusätzliche für die abgeleitete Klasse
}
```





- **Achtung:** das Überschreiben von Methoden („overriding“) darf nicht mit dem Überladen von Methoden verwechselt werden („overloading“)
  - wenn eine Methode einer abgeleiteten Klasse im Namen mit einer Methode der Basisklasse übereinstimmt, sich aber in den Parametern unterscheidet (weniger, mehr, andere Datentypen), handelt es sich um das Überladen von Methoden
  - dadurch wird in der abgeleiteten Klasse eine zusätzliche Variante der Methode der Basisklasse deklariert
  - die abgeleitete Klasse stellt dann beide Varianten zur Verfügung





- bei der Deklaration der Attribute einer abgeleiteten Klasse kommt es selten zu Namenskonflikten, da die Attribute der Basisklasse in der Regel **`private`** sind
  - dennoch müssen auch für den seltenen Fall Vorkehrungen getroffen werden, weil zum Beispiel folgende Situation denkbar ist:
    - eine wichtige Klasse **`A`** befindet sich seit Jahren in einer viel genutzten Klassenbibliothek
    - sie wurde in Tausenden von Programmen als Basisklasse benutzt
    - der Klasse **`A`** soll nachträglich ein Attribut hinzugefügt werden, das **`public`** und **`final`** ist (Konstante)





  - dadurch kann es in diversen abgeleiteten Klassen zu einem Namenskonflikt kommen!
- wenn es bei Attributen zu einem Namenskonflikt kommt, wird das Attribut der Basisklasse durch das gleichnamige Attribut der abgeleiteten Klasse verdeckt
  - dabei können die Attribute sogar unterschiedliche Datentypen haben
  - auch Attributmodifizierer spielen keine Rolle
- um das zu illustrieren, betrachten wir das folgende, auf das Wesentliche reduzierte Beispiel

```
public class TestA                  // völlig untypische Klasse !
{
  public final int a = -1;
  public final double x = 3.14;
}
```

```
public class TestB extends TestA    // völlig untypische Klasse !
{
  public String a = "Hallo";                // Namenskonflikt !
}
```

die Anweisungen

```
TestB testObjekt = new TestB();
System.out.println(testObjekt.a);
System.out.println(testObjekt.x); // ok, weil geerbt und public
```

führen zu der Ausgabe:

```
Hallo
3.14
```





- innerhalb der abgeleiteten Klasse kann auf das überschriebene Attribut mit **`super`** zugegriffen werden
  - aber nicht von außen
  - um das zu zeigen, fügen wir der Klasse **`TestB`** eine Methode hinzu

```
public class TestB extends TestA  // völlig untypische Klasse !
{
  public String a = "Hallo";

  public void schreibeInfo()
  {
    System.out.println("TestB:         a --->  " + a);
    System.out.println("TestB:   super.a --->  " + super.a);
  }
}
```





die Anweisungen

```
TestB testObjekt = new TestB();
System.out.println(testObjekt.a);
System.out.println(testObjekt.x); // ok, weil geerbt und public
System.out.println();
testObjekt.schreibeInfo();
```

führen jetzt zu der Ausgabe:

```
Hallo
3.14

TestB:         a --->  Hallo
TestB:   super.a --->  -1
```





- weil jedes Objekt einer von **`A`** abgeleiteten Klasse **`B`** nach dem Liskov'schen Substitutionsprinzip uneingeschränkt in allen Situationen eingesetzt werden kann, in denen ein Objekt der Klasse **`A`** eingesetzt werden kann, ist es sinnvoll, Wertzuweisungen der folgenden Art zuzulassen

```
A testObjekt = new B();
```

  - das heißt, es ist sinnvoll, **`B`** als zuweisungskompatibel zu **`A`** anzusehen

- allgemeiner gibt es bezüglich der Zuweisungskompatibilität bei Referenzdatentypen folgende Regeln
  - `null` ist zuweisungskompatibel zu allen Referenzdatentypen (Klassen, Arrays, `enums`, Interfaces)
  - wenn `A` und `B` Klassen sind, ist `B` zuweisungskompatibel zu `A`, wenn `A` identisch ist mit `B` oder `B` von `A` abgeleitet ist (auch über mehrere Stufen)
  - der Arraydatentyp `B[]` ist zuweisungskompatibel zu `A[]`, wenn `A` und `B` identische primitive Datentypen sind oder `A` und `B` Referenzdatentypen sind und `B` zuweisungskompatibel zu `A` ist (Rekursion!)
  - für höherdimensionale Arrays gilt Analoges

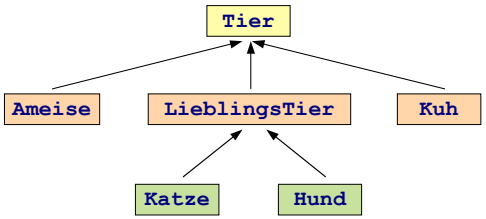




- über diese Regeln hinaus gibt es weitere, die wir zu gegebener Zeit kennen lernen werden
  - z.B. für Interfaces
- wegen dieser (natürlichen) Regeln kommt es zu einem Phänomen, das es bei primitiven Datentypen nicht gibt
  - wir erklären es an folgender Klassenhierarchie








- die beteiligten Klassen haben eine sehr einfache Struktur:





```
public class Tier
{
  private String art;

  public Tier(String dieArt)
  {
    art = dieArt;
  }

  public void sprich()
  {
  }

  public String liefereInfo()
  {
    return art + ". ";
  }
}
```

- die Methode `sprich` ist zum Überschreiben vorgesehen (nicht alle Tiere können „sprechen“)

```
public class Ameise extends Tier
{
  public Ameise()
  {
    super("Ameise");
  }

  public String liefereInfo()
  {
    return "Ich bin eine " + super.liefereInfo();
  }
}
```

- da Ameisen nicht „sprechen“ können, wird **sprich** nicht überschrieben





```
public class Kuh extends Tier
{
  public Kuh()
  {
    super("Kuh");
  }

  public void sprich()
  {
    System.out.print("Muuuh!");
  }

  public String liefereInfo()
  {
    return "Ich bin eine " + super.liefereInfo();
  }
}
```

- eine Kuh kann aber charakteristische Laute von sich geben





```
public class LieblingsTier extends Tier
{
  private String name;

  public LieblingsTier(String dieArt, String derName)
  {
    super(dieArt);
    name = derName;
  }

  public String liefereInfo()
  {
    return super.liefereInfo() + "Ich heisse " + name + ". ";
  }
}
```

- Objekte vom Typ **LieblingsTier** zeichnen sich dadurch aus, dass sie einen Namen haben
- für unseren Zweck nehmen wir an, dass alle Hunde und Katzen einen Namen haben (fragwürdig!)





```
public class Hund extends LieblingsTier
{
  private String rasse;

  public Hund(String  derName, String dieRasse)
  {
    super("Hund", derName);
    rasse = dieRasse;
  }

  public void sprich()
  {
    System.out.print("Wau, wau!");
  }

  public String liefereInfo()
  {
    return "Ich bin ein " + super.liefereInfo() +
           "Ich bin ein " + rasse + ". ";
  }
}
```

```
public class Katze extends LieblingsTier
{
  private String rasse;

  public Katze(String  derName, String dieRasse)
  {
    super("Katze", derName);
    rasse = dieRasse;
  }

  public void sprich()
  {
    System.out.print("Miau, miau.");
  }

  public String liefereInfo()
  {
    return "Ich bin eine " + super.liefereInfo() +
           "Ich bin eine " + rasse + ". ";
  }
}
```





wir betrachten nun folgende Anweisungen:

```
Tier testTier = new Ameise();      // ok! zuweisungskompatibel
System.out.print(testTier.liefereInfo());              // (*)
testTier.sprich();                                     // (#)
System.out.println();

testTier = new Hund("Lumpi", "Dackel");     // ok! siehe oben!
System.out.print(testTier.liefereInfo());              // (*)
testTier.sprich();                                     // (#)
System.out.println();
```

sie führen auf der Konsole zu dieser Ausgabe:

```
Ich bin eine Ameise.
Ich bin ein Hund. Ich heisse Lumpi. Ich bin ein Dackel. Wau, wau!
```

**obwohl die durch `(*)` bzw. `(#)` gekennzeichneten Anweisungen völlig identisch sind, haben sie unterschiedliche Auswirkungen!**





- das Verhalten hängt von der Situation ab !
- dieses Phänomen nennt man Polymorphismus
  - vom Griechischen „polymorph“ (= vielgestaltig)
- diese Form des Polymorphismus kann nur bei Vererbung in Verbindung mit dem Überschreiben von Methoden auftreten, die nicht **`static`** sind
  - er zählt (zusammen mit der Datenkapselung und der Vererbung) zu den charakteristischen Eigenschaften der Objektorientierung





- genauer betrachtet zeigt sich, dass das Verhalten der Referenz davon abhängt, von welchem Typ das Objekt ist, auf das sie aktuell zeigt
  - man sagt, dass „das Objekt selbst weiß, was es kann und wie es sich verhält“
    - eine Ameise weiß, dass sie nicht „sprechen“ kann
    - ein Hund weiß, dass er bellen kann
  - das Programm kann sich darauf verlassen und braucht die Situation zum Beispiel nicht kompliziert zu analysieren:
    - „wenn das Tier eine Ameise ist, dann ...“
    - „wenn aber das Tier ein Hund ist, dann ...“

- richtig eingesetzt ist Polymorphismus ein mächtiges Mittel zur Gestaltung eleganter Programme
  - dies mögen folgende Methode und deren Anwendung verdeutlichen

```
private void schreibeTierInfo(Tier einTier)
{
  System.out.print(einTier.liefereInfo());
  einTier.sprich();
  System.out.println();
}
```

  - da der Parameter von **`schreibeTierInfo`** vom Typ **`Tier`** ist, kann die Methode jede Art von Tier als Input erhalten





- weil das übergebene Tier(objekt) selbst weiß, was es kann, gibt **`schreibeTierInfo`** für jedes Tier genau die passende Information aus:

```
Tier[] viecher = { new Hund("Lumpi", "Dackel"),
                   new Katze("Pussy", "Siamkatze"),
                   new Kuh(),
                   new Ameise(),
                   new Katze("Mieze", "Perserkatze"),
                   new Hund("Tarzan", "Pinscher"),
                   new Hund("Lu", "Wolfsspitz")
                 };
for (Tier kreatur : viecher)
  schreibeTierInfo(kreatur);
```

```
Ich bin ein Hund. Ich heisse Lumpi. Ich bin ein Dackel. Wau, wau!
Ich bin eine Katze. Ich heisse Pussy. Ich bin eine Siamkatze. Miau, miau.
Ich bin eine Kuh. Muuuh!
Ich bin eine Ameise.
Ich bin eine Katze. Ich heisse Mieze. Ich bin eine Perserkatze. Miau, miau.
Ich bin ein Hund. Ich heisse Tarzan. Ich bin ein Pinscher. Wau, wau!
Ich bin ein Hund. Ich heisse Lu. Ich bin ein Wolfsspitz. Wau, wau!
```





- Polymorphismus ist auch aus technischer Sicht interessant
  - bei einer überladenen Methode muss der Compiler den Bytecode aller möglichen Versionen der Methode in der Method Area ablegen
  - erst die JVM kann zur Laufzeit (in Abhängigkeit von der Situation) entscheiden, welche der Versionen tatsächlich ausgeführt wird
- man bezeichnet dies als „late binding“ (oder auch „dynamic binding“, im Gegensatz zu dem bei prozeduralen Sprachen üblichen „static binding“)





- **Achtung:** Polymorphismus gibt es nicht bei Methoden, die **`static`** sind
  - Methoden, die **`static`** sind, können zwar in abgeleiteten Klassen überschrieben werden
  - die verschiedenen Versionen können auch durch Voranstellen des jeweiligen Klassennamens aktiviert werden
    - zum Beispiel **`A.machWas()`** oder **`B.machWas()`**
  - werden sie jedoch über Objektreferenzen aktiviert (was möglich, aber nicht sinnvoll ist), wird immer nur die Methode der Basisklasse ausgeführt
    - weil bei ihnen (wie bei prozeduralen Sprachen) „static binding“ vorgenommen wird (daher der Name!)