# Klassen in Java

**Was sind Klassen?**

Klassen sind eine Sammlung von Objekten. Jedes dieser Objekte hat bestimmte Eigenschaften und Fähigkeiten. Einzelne Objekte können wie erwähnt zu Objektklassen zusammengefasst werden.

Dies möchte ich am Beispiel Biologie verdeutlichen: Es gibt die Oberklasse der Lebewesen, diese haben z.B. alle einen Stoffwechsel, sind auf Fortpflanzung aus und bewegen sich; man kann sie jedoch z.B. auch noch in Amphibien, Säugetiere oder Spinnentiere einklassieren. Diese Objekte haben alle die gleichen Eigenschaften wie die Oberklasse Lebewesen. Jedoch noch spezifische Eigenschaften, so sind Säugetiere z.B. Plazentatiere (außer Schnabeltier).

Genauso ist es auch in Java. Klassen sind also Schablonen, ein exakt definierter Bauplan. Von diesem Bauplan werden Kopien erstellt. Diese Kopien haben alle Eigenschaften und Fähigkeiten der Klasse und können noch zusätzliche Eigenschaften und Fähigkeiten haben.

In Java bezeichnet man die Eigenschaften als Attribute und die Fähigkeiten als Methoden. Methoden verändern den Zustand eines Objekts durch Veränderung seiner Attribute. Also kann ein Tierobjekt folgende Attribute haben: Größe, Farbe, Gewicht. Durch die Methode Nahrungsaufnahme() kann das Attribut Gewicht verändert werden.

**Wie arbeitet man mit Klassen?**

Grundsätzlich könnte man mehrere Klassen in einer Datei speichern, jedoch darf nur eine davon mit *public class* gekennzeichnet werden. Jedoch ist das kein guter Stil, da dadurch die Übersichtlichkeit - gerade in größeren Projekten - verloren geht. Darum wird für jede neue Klasse auch eine neue Java-Datei angelegt, diese werden dann alle in dem gleichen Verzeichnis gespeichert.

Nun aber mal zum Aufbau:

```
public class Klassenname
{
  //Attribute
  public Datentyp Name;
}
```

Die Definition dieser öffentlichen (public) Klasse beginnt mit *public class* gefolgt von dem Klassennamen. Wie bei der Hauptklasse (die mit dem *main*-Block) wird der Rumpf durch *{* und *}* begrenzt. Klassennamen sollten immer mit einem Großbuchstaben beginnen, der Klassenname darf keine Leerzeichen enthalten und Umlaute sind auch nicht gestattet.

Die Attribute der Klasse werden in deren Rumpf vereinbart. Bei der Deklaration der Attribute wird der Modifier *public* vorangestellt, der die Zugriffsrechte auf die Attribute regelt. Man kann nämlich auch den Zugriff auf diese Attribute von außerhalb unterbinden (wird Kapselung genannt).

Will man nun Kopien der Klasse machen, so muss man eine Referenz erzeugen und den Konstruktor ausrufen. Das hört sich kompliziert an, ist jedoch nicht schwerer zu verstehen als die Deklaration von Variablen. Denn die Referenzerzeugung ist nichts anderes als die Vereinbarung der Objektnamen für die Kopie der Klasse. Mit dem Aufruf des Kontruktors wird dann der benötigte Speicherplatz reserviert. Das sieht dann so aus:

*Klassenname objektname;*
*objektname = new Klassenname();*

Danach kann auf die öffentlichen Attribute der Klasse über den Punktoperator zugegriffen werden. Das sieht dann so aus:
*Objektname.attributname = wert;*

**Methoden**

Eine Methode ist ein Programmteil (ähnlich einer Funktion), der in die Klassendefinition eingebettet ist. Er kann uneingeschränkt auf alle Daten des Objekts zugreifen. Methoden verringern den Programmieraufwand in dem Programm, da man Methoden nur einmal (eben in der Klassendefinition) programmieren muss und danach ganz einfach durch Aufruf ihre Funktion einsetzen kann.
Allgemein sehen Methoden so aus:

```
[Modifier] Typ Name([Parameter])
{
  Anweisungen
}
```

Der Modifier ist nicht erforderlich und beinhaltet Schlüsselwörter wie *public*. Die Namen von Methoden müssen immer mit einem Kleinbuchstaben beginnen.
Werden mit der Methode Attribute verändert, so sollte der Methodenname mit *set* (z.B. ***set****SeiteL*) beginnen.
Sollen Methoden keinen Rückgabewert haben, d.h. hat die Berechnung kein Ergebnis, dann ist das Schlüsselwort void einzusetzen. Somit würde die erste Zeile folgendermaßen aussehen:
*public* ***void*** *Name()*

**Beispielprojekt: Klassen**

Da das Thema nun nicht so einfach zu verstehen ist, möchte ich nun an dieser Stelle mal schnell ein kleines Beispielprojekt einschieben, an Hand dessen man sich die komplizierten Fakten etwas besser zu Gemüte führen kann.
Ich möchte ein Programm schreiben, das die Flächen zweier rechteckigen Grundstücke berechnet. Die Berechnung soll in der Klasse 'Grund' gemacht werden. Diese Klasse soll so aussehen:

```
public class Grund
{
  public double seiteL;
  public double seiteB;

  public void showFlaeche()
  {
    System.out.println("Die Fläche des Grundstücks beträgt: "+(seiteL * seiteB)+"
    qm");
  }
}
```

Die Klasse 'Haupt' ist die Hauptklasse und soll so aussehen:

```
import java.io.*;
public class Haupt
{
  public static void main(String argv[]) throws IOException
  {
    BufferedReader in = new BufferedReader (new InputStream-
    Reader(System.in));
    Grund g1,g2;
    g1 = new Grund();
    g2 = new Grund();
    System.out.println(" - Grundstücksvergleich -");
    System.out.println(" -------------");
    System.out.print("Bitte Länge von Grundstück1 eingeben:");
    g1.seiteL = Integer.parseInt(in.readLine());
    System.out.print("Bitte Breite von Grundstück1 eingeben:");
    g1.seiteB = Integer.parseInt(in.readLine());
    System.out.print("Bitte Länge von Grundstück2 eingeben:");
    g2.seiteL = Integer.parseInt(in.readLine());
    System.out.print("Bitte Breite von Grundstück2 eingeben:");
    g2.seiteB = Integer.parseInt(in.readLine());
    System.out.println("Grundstück1 " +g1.seiteL+"X"+g1.seiteB);
    g1.showFlaeche();
    System.out.println("Grundstück1 " +g2.seiteL+"X"+g2.seiteB);
    g2.showFlaeche();
  }
}
```

In der Klasse Grund sieht man nun die Methode showFlaeche, die die Fläche des Grundstücks ausrechnet und ausgibt. In der Haupt-Klasse werden zwei Klassenobjekte g1 und g2 erstellt. Den Attributen dieser beiden Instanzen werden nun die Eingaben des Benutzers zugewiesen. Mit dem Aufruf der Methoden werden nun die Fläche berechnet.
So sieht die Arbeit mit Klassen in Java im einfachsten Fall aus. Man könnte die Anzahl der zu vergleichenden Grundstücke mit sechs Zeilen Code beliebig erweitern!

**Methoden mit Parameter**

Nach dem Beispiel sollte nun jeder den grundsätzlichen Gebrauch von Klassen verstanden haben. Nun wollen wir diese Kenntnisse noch ausbauen. Mit Parametern sind Methoden noch variabler einsetzbar. Bei Parametern handelt es sich um Informationen die an die Methode übergeben werden. Mit diesen Informationen arbeitet die Methode.
Diese Übergabe erfolgt über die Parameterliste. In der Klassendeklaration habe ich die Parameterliste schon erwähnt. Hier noch mal zur Erinnerung:
*[Modifier] Typ Name****([Parameter])***

Im einfachsten Fall war die Parameterliste leer (wie in unserem Beispiel). Bei der Methode main in der Hauptklasse sieht man jedoch schon einmal eine Parameterliste (diese ist zwingend vorgeschrieben!).
Da der Parametername oft gleichlautend mit dem Attributname gewählt wird, wird

zur besseren Übersichtlichkeit das Schlüsselwort *this* eingesetzt, das folgendermaßen eingesetzt wird:
***this.****Parametername = Wert;*

Hier wird dem Parameter ein Wert zugewiesen und nicht dem Originalattribut.
Müssen mehrere Parameter übergeben werden, so wird bei der Definition der Parameterliste jeder Parameter einzeln definiert (Typ und Name) und getrennt durch ein Komma aufgelistet. Außerdem ist darauf zu achten, dass der übergebene Wert kompatibel zum Datentyp des Parameters ist, sonst kommt es zu einem Fehler!

**Methoden überladen**

Überladen von Methoden bedeutet, dass es eine Methode mehrfach mit gleichem Namen, aber mit unterschiedlicher Parameterliste gibt. Auf diese Weise können benutzerfreundliche Klassen erstellt werden, die z.B. sowohl Zahlen als auch Strings akzeptieren.
So könnte man folgende Methodenüberladung für unser Beispiel vornehmen:

```
public void showFlaeche(double seiteL)
{
}
public void showFlaeche(String seiteL)
{
  this.seiteL(Double.parseDouble(seiteL));
}
```

Es findet also abhängig vom Datentyp eine automatische Konvertierung des Datentyps statt. Damit lässt sich ebenfalls einiges an Programmieraufwand einsparen. So könnte man z.B. die Umwandlung der Eingaben vom InputStreamReader über diese Methode laufen lassen.

**Kapselung**

Bis jetzt war es immer möglich direkt auf alle Attribute der Klasse zuzugreifen. Jedoch ist es nicht immer erwünscht, dass jeder von außerhalb auf die Attribute der Klasse zugreifen darf. Mit dem Modifier den ich schon ein paar Mal erwähnt habe, kann man den Zugriff beschränken. Bis jetzt war der Modifier immer *public* (=öffentlich), will man den Zugriff begrenzen, so setzt man den Modifier *private* (=nicht-öffentlich) ein. Ist als Modifier *private* gewählt, dann kann nur noch innerhalb der Klasse auf das private Attribute zugegriffen werden, von der Hauptklasse (oder jeder anderen Klasse) ist aber der Zugriff auf dieses Attribut nun nicht mehr möglich.
Zur Erinnerung hier noch mal die Syntax der Attributdefinition: ***[public/private]*** *Datentyp Name;*

Will man Zugriff von außerhalb, so muss man Methoden aus der Klasse der mit den privaten Attributen verwenden. Mit diesen Methoden kann man Fehleingaben abfangen (durch Methodenüberladung).
Das Verfahren, dass man nur noch über Methoden auf die privaten Attribute zugreifen kann nennt man *Kapselung*.
Das ganze funktioniert natürlich auch bei Methoden. Normalerweise sind aber Attribute einer Klasse *private* und Methoden *public*, damit die Attribute geschützt sind und die einmal definierten Methoden mehrfach verwendet werden können. Intern ge-

nutzte Methoden können jedoch auch auf *private* gesetzt werden.

**Methoden mit Rückgabewert**

Eingangs habe ich schon erwähnt, dass man bei Klassen die keinen Rückgabewert haben sollen das Schlüsselwort void einsetzen muss. Zur Erinnerung: *public* ***void*** *setSeiteL(seiteL)*

Im Umkehrschluss heißt dass, das alle Methoden für gewöhnlich einen Rückgabewert haben. Rückgabewert heißt, dass das Ergebnis, dass nach der Abarbeitung der Methode erzeilt wird, anstelle des Aufrufs steht. Also z.B.

*summe = getSumme(int Sum1, Sum2)*

Anstelle von getSumme steht dann da das Ergebnis. Hieran sieht man auch schon, dass Methoden, die einen Rückgabewert zurückgeben sollen mit *get* (***get****Summe*) beginnen.

Die Synatx der Methodendefinition mit Rückgabetyp sieht so aus:

```
[Modifier] Typ Name([Parameter])
{
  Anweisungen
  return wert;
}
```

Eine Methode mit Rückgabewert muss immer mit der Anweisung *return wert;* enden. Nach dieser Anweisung darf keine weitere Anweisung folgen, da *return wert;* zum sofortigen Ende der Methode führt. Der Typ des Rückgabewerts muss zum Datentyp der Methode passen!

**Statische Methoden und Variablen**

Wird eine Instanz von einer Klasse erstellt, so wird die Schablone der Klasse angewendet, d.h. alle Methoden und Attribute werden kopiert und es werden individuelle Daten in die Methoden und Attribute der Instanz eingefügt. Manchmal gibt es aber Attribute und Methoden, die eigentlich nichts mit den Daten zu tun haben, sie werden nicht mit individuellen Daten der Instanz gefüllt. Darum macht es eigentlich auch keinen Sinn, diese in die einzelnen Instanzen zu kopieren, sie bekommen dort ja keine individuellen Daten. Deshalb belässt man diese Methoden und Attribute in der Ursprungsklasse. Diese Methoden und Attribute werden statisch genannt.

Um dieses statische Verhalten umzusetzen, wird vor das Attribut oder die Methode das Schlüsselwort *static* gesetzt. Das sieht so aus: *public* ***static*** *int ausgabeFormat;*

Um auf ein solches Attribut zuzugreifen, müssen wir einfach den Klassennamen benutzen.

**Beispielprojekt: Statische Methoden und Variablen**

Theoretisch mag man den Sachverhalt nun verstanden haben, jedoch möchte ich das aus noch praktisch verdeutlichen. Ich möchte einen Umrechner für kW in PS-Beträge entwerfen. Der Benutzer soll eine Zahl eingeben und dann wählen dürfen, ob er von kW in PS oder anders herum konvertieren möchte. Eine statische Methode soll dann prüfen, ob die korrekte Zahl eingegeben wurde. Eine andere Methode soll dann die Berechnung vornehmen, das ganze wird dann schön formatiert ausgegeben. Der Quelltext dazu soll so aussehen:

```
import java.lang.Math;
public class Rechner
```

```
{
private int zahl;
public static int auswahl;

public static void setAuswahl(int w)
{
  if (w==1 || w ==2)
  auswahl =w;
  else auswahl=1;
}
public void setZahl(int zahl)
{
  if(zahl >=0) this.zahl=zahl;
  else zahl=1;
}
public void showErgebnis()
{
int erg;
  if(auswahl==1)
  {
     erg = (int)Math.floor((zahl * 1.361) + 0.5d);
     System.out.println(zahl+"kW entsprechen "+ erg+"PS!");
  }
  else
  {
     erg = (int)Math.floor((zahl * 0.735) + 0.5d);
     System.out.println(zahl+"PS entsprechen "+ erg+"kW!");
  }
}
```

So sieht die Hauptklasse aus:

```
import java.io.*;
public class Haupt
{
   public static void main(String argv[]) throws IOException
   {
      BufferedReader in = new BufferedReader (new InputStream-
      Reader(System.in));
      Rechner r1;
      r1 = new Rechner();
      System.out.println(" - kW-PS Umrechner -");
      System.out.println(" ---------------------------");
      System.out.print("Für Umrechnung kW->PS (1), für PS->kW (2) einge-
      ben:");
      Rechner.setAuswahl(Integer.parseInt(in.readLine()));
      System.out.print("Bitte Betrag eingeben:");
```

```
            r1.setZahl(Integer.parseInt(in.readLine()));
            r1.showErgebnis();
        }
    }
```

**Konstruktor**

Ich möchte noch etwas zum Konstruktor loswerden. Zur Erinnerung: Der Konstruktor ist bei der Erzeugung von Instanzen einer Klasse für die Reservierung von Speicherplatz für die Instanz zuständig.
Bisher haben wir den Konstruktor (z.B. r1 = new Rechner();) immer nur aufgerufen. Mittlerweile sollte jedem klar sein, dass in Java immer alles definiert sein muss, bevor man es ausrufen kann. So ist es - wie sollte es auch anders sein - auch bei den Konstruktoren. Eigentlich braucht man sich in Java nicht um den Konstruktor zu kümmern, da der Compiler (der Übersetzer des Quellcodes in Bytecode) so freundlich ist und - sollte keiner definiert sein - diesen in der Klasse durch den Standartkonstruktor ergänzt.
Es hat jedoch einen Grund, warum ich trotzdem auf den Konstruktor eingehe: Der Standartkonstruktor hat keine Parameter! Manchmal ist es jedoch sinnvoll mit Parametern zu arbeiten. Schließlich kann man diese überladen. Da haben wir es schon! Durch Parameter kann man Überladen und durch Überladen wird das Ganze flexibler und benutzerfreundlicher. Das ist ein sehr guter Grund die wenigen Zeilen der Definition des Konstruktors selber zu übernehmen.
Der Konstruktor ist eine Art "Methode ohne Rückgabewert" und trägt immer denselben Namen wir die Klasse in der er steht. Der Konstruktor hat KEINEN Rückgabewert (aus diesem Grund ist NICHT mit void zu arbeiten!). Bei der Klasse "Rechner" sieht der Konstruktor also so aus:

```
public Rechner()
{
}
```

Die Konstruktoren können in der Klassendefinition zwar an einer beliebigen Stelle stehen, jedoch ist es guter Stil folgende Reihenfolge einzuhalten: Attribute, Konstruktoren, Methoden.

**Vererbung**

Die Vererbung ist eines der wichtigsten objektorientieren Prinzipien der Klassen. Die Vererbung in Java ist im Grunde genommen der des Menschen ganz ähnlich. Der Vorfahr - in Java die Ober- oder Superklasse - vererbt an den Nachfahr - in Java die Unter- oder Subklasse - alle Eigenschaften, jedoch kann der Nachfahr noch einige Eigenschaften zusätzlich haben. In der Praxis sieht das so aus: Es wird eine allgemeine Klasse erstellt, z.B. Mensch, danach wird eine weitere spezifischere Klasse, z.B. Arbeiter, erstellt, bei der wird Vererbung eingesetzt, d.h. alle Eigenschaften werden automatisch übernommen und in der zweiten Klasse werden zu den Eigenschaften die die Klasse schon von der ersten geerbt hat noch weitere spezifische Eigenschaften hinzugefügt. So braucht man die allgemeinen Eigenschaften nur einmal definieren und kann diese in den speziellen Klassen um spezielle Eigenschaften erweitern.
In Java wird die Vererbung mit folgender Syntax vollzogen:

```
public class Klasse1
{ ... }
```

*public class Klasse2 extends Klasse1*
*{ ... }*

Klasse1 ist die Oberklasse und Klasse2 die Unterklasse. Man sieht, dass die eigentliche Vererbung mit *extends Oberklasse* von Statten geht. Nun kann man auch auf die geerbeten Eigenschaften von Klasse2 zugreifen als wären es die eigenen Eigenschaften. Ist z.B. in Klasse1 das Attribut *name* vereinbart, so ist es möglich auf dieses Attribut der Klasse2 mittels *Klasse2.name* zuzugreifen, obwohl doch eigentlich in Klasse2 kein solches Attribut vereinbart wurde; die Vererbung machts möglich!
Wir haben uns einige Abschnitte zuvor mit Modifiern befasst. Diese haben auch Einfluss auf die Vererbung! Ist ein Attribut oder eine Methode mit *private* gekennzeichnet, so wird diese NICHT mitvererbt, sondern bleibt gemäß der Definition in der Oberklasse. Zusätzlich zu dem Modifier *private* kommt bei der Vererbung noch der Modifier *protected* ins Spiel.
Dieser Modifier bewirkt in der Oberklasse das gleiche wie der Modifier *private* (der Zugriff von außen wird verhindert), jedoch können Attribute oder Methoden, die mit *protected* gekennzeichnet sind mitvererbt werden. So haben diese Attribute oder Methoden auch in der Unterklasse die zugriffbeschränkenden Eigenschaften.

**Methoden überschreiben**
Bei der Vererbung werden Eigenschaften aus der Oberklasse an die Unterklasse vererbt und um Methoden und Attribute ergänzt. Wenn aber in der Unterklasse eine Methode mit dem gleichen Namen, der gleichen Parameterliste und dem gleichen Rückgabewert definiert wird, so nennt man das Überschreiben einer Methode.
Beim Überschreiben der Methode gibt es zwei Möglichkeiten, entweder man will eine ganz neue Definition der Methode durchführen, oder man will die Methode nur erweitern. Wenn man die Methode nur erweitern will, so kann man mit der Referenz *super* auf die Eigenschaft der Oberklasse verweisen. Folgendes Beispiel soll das verdeutlichen:

```
public class Klasse1
{
 public void meth()
 {
  System.out.println("Sehr geehrte Damen,");
 }
}

 public class Klasse2 extends Klasse1
 {
  public void meth()
  {
   super.meth();
   System.out.println("liebe Herren");
  }
 }

        public class Haupt
        {
         public static void main (String
```

```
argv[])
{
 Klasse2 k = new Klasse2();
 k.meth();
}
}
```

Die abgeleitete Klasse sorgt dafür, dass nicht nur '*Sehr geehrte Damen,*' sondern auch noch eine Zeile tiefer '*liebe Herren*' ausgegeben wird.
Hinter *super* muss immer eine Methode oder ein Attribut stehen!

**Konstruktoren und Vererbung**

Konstruktoren werden nicht vererbt. Deswegen muss man bei Bedarf in den Unterklassen ganz neue Konstruktoren definieren. Wenn ein Objekt einer Unterklasse erzeugt wird, ruft der Konstruktor der Unterklasse automatisch den Standart-Konstruktor der Oberklasse auf.
Wie schon soeben beim Überschreiben der Methoden, kommt auch jetzt wieder das Schlüsselwort *super* zum Einsatz. Diesmal jedoch gelten andere Regel für die Benutzung! Denn nun darf keine Anweisung vor dem Aufruf des Konstruktors stehen.
Der Kontsruktor wird jedoch erst dann interessant, wenn man Parameter einsetzt; und das sieht so aus:

```
public class KlasseA
{
   private int x;
   public KlasseA(int wert)
   {
     x = wert;
   }
}

public class KlasseB extends KlasseA
{
   public KlasseB(int zahl)
   {
     super(zahl);
   }
}
```

In KlasseB wird der Konstruktor mit der Parameterliste (also mit int-Werten) der Oberklasse ausgerufen.

**Polymorphie**

Polymorphie (= Vielgestaltigkeit) ist eine weitere objektorientierte Methode von Java. Sie ermöglicht dynamisches Binden, d.h. der Compiler entscheidet zur Laufzeit dynamisch welche Methode er aufruft.
Das klingt jetzt komisch und bevor ich mit der eigentliche Polymorphie beginne muss ich noch eine weitere Technik einführen: die Typanpassung. Instanzen werden ja durch folgende Anweisung erzeugt: *Klasse instanz = new Klasse();*
Es ist jedoch auch möglich ein Objekt einer Unterklasse einem Objekt der Oberklasse

zuzuweisen. Das sieht dann so aus:
*Erbe instanz1 = new Erbe();*
*Ahn instanz2 = instanz1;*

Man erzeugt also zuerst das Objekt *instanz1* - das auf herkömmlichen Wege erstellt wurde - danach erzeugt man eine Referenz *instanz2* der Klasse *Ahn* und lassen diese auf das Objekt *instanz1* zeigen. Da das *Erbe*-Objekt ein spezialisiertes Ahn-Objekt ist, funktioniert diese Zuweisung. Das erscheint auf den ersten Blick blödsinnig! Jedoch übernimmt *instanz2* alle Attribute und Methoden die Ober- und Unterklasse gemeinsam haben. Alle Attribute und Methoden die nach der Vererbung in die Unterklasse *Erbe* eingefügt wurden, werden also nicht in *instanz2* übernommen.
**Achtung!** Es lässt sich immer nur eine Referenz von Oberklasse auf die Unterklasse legen. Es ist also nicht möglich eine Referenz vom Typ Erbe auf Ahn zu legen (folgende Anweisung ist also nicht möglich: Erbe instant1 = instanz2;). Hier gilt, dass die Typen unvereinbar sind.

Das war die Vorarbeit. Nun geht es an die eigentliche Polymorphie. Wir haben soeben definiert, dass bei der Typanpassung in das Oberklasseobjekt alle Attribute und Methoden übernommen werden, die Ober- und Unterklasse gemeinsam haben. Was aber ist, wenn alle Klassen die gleichen Attribute und Methoden besitzen und in den Unterklassen nur die Methoden überschrieben werden?
Was also macht man bei folgender Codierung:
*Ahn a1 = new Erbe1();*
*Ahn a2 = new Erbe2();*
*System.out.println(a1.text());*
*System.out.println(a2.text());*

Wird nun in beiden Fällen die Methode *text()* aus der Ahn-Klasse aufgerufen? NEIN! Der Compiler registriert zwar dass es sich um die *Ahn*-Klasse handelt, er erkennt jedoch auch dass ein *Erbe*-Objekt erzeugt wird.
Wie eingangs erwähnt entscheidet sich der Compiler dynamisch welche Methode auszurufen ist. Die Polymorphie wurde also eingesetzt.