

# **Algorithmen und Programmierung**

## **Prüfungsvorbereitung / Zusammenfassung von Materialien**

---

- 1. Dokumentinfos**
  - 1. Versionshistory**
  - 2. Kontaktmöglichkeiten**
- 2. Struktogramm**
  - 1. Blocktypen**
    - 1. Anweisung**
    - 2. Verzweigung**
    - 3. Schleife**
  - 2. Beispiel**
- 3. Datentypen**
  - 1. ganze Zahlen**
  - 2. Kommazahlen**
  - 3. Zeichen**
  - 4. andere Datentypen**
- 4. Arrays**
  - 1. 1-dimensionale Arrays**
  - 2. mehrdimensionale Arrays**
- 5. Funktion**
- 6. Rekursion**
- 7. Klassen / Objekte**
  - 1. Vererbung**
  - 2. Funktionen überschreiben**
  - 3. Funktionen überladen**
  - 4. Konstruktor**
- 8. Suchalgorithmen**
  - 1. Sequenzielle Suche**
  - 2. Binäre Suche**

# **1. Dokumentinfos**

## **Versionshistorie**

### **24/01/2012**

- erste Version erstellt
- Rechtschreibfehler korrigiert
- Struktogramm geändert

### **25/01/2012**

- Rechtschreibfehler korrigiert
- "Sequenzielle Suche" überarbeitet

## **Kontaktmöglichkeiten**

Fragen zum Dokument kann man einreichen unter: [david.pauli@tu-ilmenau.de](mailto:david.pauli@tu-ilmenau.de).

## 2. Struktogramm

Das Struktogramm (auch Nassi-Shneiderman-Diagramm<sup>1</sup>) dient zur Veranschaulichung eines Algorithmuses. Die Abläufe des Programmes werden in Blöcken anschaulich dargestellt.

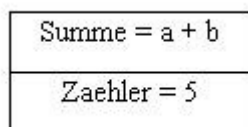
### Blocktypen

#### Anweisung

Eine Anweisung ist der einfachste und grundlegendste Teil eines Algorithmuses. In diesem wird eine grundlegende Operation beschrieben.

Im Beispiel wird auf **Summe** die Addition von **a** und **b** gespeichert. Weiterhin wird auf **Zaehler** der Wert 5 abgelegt.

Der programmierte Teil in Java sieht ähnlich aus. Lediglich ein Semikolon (;) wird nach jeder Anweisung als Abgrenzung angehängt.



```
Summe = a + b;
Zaehler = 5;
```

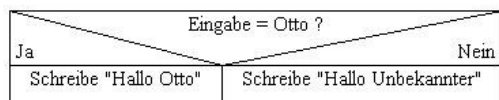
#### Verzweigung

Die Verzweigung bietet die Möglichkeit einer Fallunterscheidung. Somit kann man unterschiedliche Wegverzweigungen realisieren.

Im Beispiel wird gefragt, ob in der **Eingabe** der Wert **Otto** abgespeichert ist. Falls dies so ist, wird **Hallo Otto** ausgegeben. Ansonsten erscheint **Hallo Unbekannter**.

In Java wird es ähnlich gemacht. Die Anweisung **if** zeigt, dass danach die Anweisung folgt, die geprüft wird. In geschweiften Klammern (**{,}**) wird geschrieben, was passiert, wenn die Anweisung wahr ist. **System.out.print(...)** ist die Ausgabe-Funktion von Java. Dort wird geschrieben, was ausgegeben wird.

Zwischen die **else**-Klammern kommt die Ausgabe, falls die Anweisung falsch ist.



```
if( Eingabe == "Otto" )
{
    System.out.print("Hallo Otto");
}
else
{
    System.out.print("Hallo Unbekannter");
}
```

<sup>1</sup> <http://de.wikipedia.org/wiki/Nassi-Shneiderman-Diagramm>

## Schleife

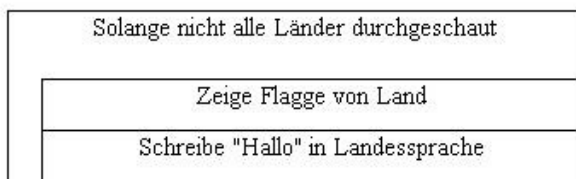
Schleifen dienen zur Wiederholung und damit zu Vereinfachung von wiederholten Abläufen. Der Vorteil ist, dass man vor dem Programmablauf nicht wissen muss, wie oft ein Programmteil wiederholt werden soll.

Im Beispiel wird der Programmcode so oft wiederholt, bis alle Länder durchgeschaut wurden. Wiederholt wird die Anzeige der Länderflagge sowie das Wort Hallo in der Landersprache.

In Java werden Schleifen mit **while** begonnen. Dahinter wird die Anweisung geschrieben, die wahr sein muss, damit die Schleife durchlaufen wird. Im Beispiel ist **Laender** ein Feld, was aus vielen Elementen besteht. (Siehe Datentypen / Felder). Mit **Laender.length** wird geschaut, aus wievielen Elementen das Feld besteht. In der Schleife wird der Wert **i** geführt, der die Anzahl der bereits getätigten Schleifendurchläufe zählt. Die Schleife wird sooft wiederholt (Wert **i**) wie viele Elemente in **Laender** vorhanden sind.

Zwischen den geschweiften Klammern (**{,}**) wird der Programmcode geschrieben, der wiederholt werden soll. Hier wird mit **System.out.print(...)** der Wert **Flagge** an der **[i]**ten Stelle ausgegeben. Dies kann das Bild der Flagge sein. Im Feld **Hallo** wird an der **[i]**ten Stelle die Begrüssung in der Landessprache gespeichert.

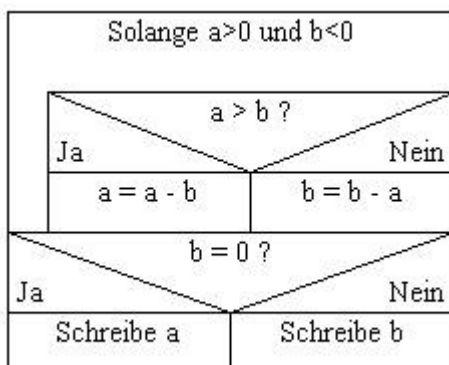
**i** muss erhöht werden, damit gezählt wird, dass die Schleife ein weiteres Mal durchgelaufen ist.



```
while ( i <= Laender.length )
{
    System.out.print( Flagge[i] );
    System.out.print( Hallo[i] );
    i = i+1;
}
```

## Beispiel

Das Beispiel zeigt, wie man mehrere Elemente ineinander verschachteln kann.



```
while ( a>0 && b<0 )
{
    if ( a>b )
    {
        a = a-b;
    }
    else
    {
        b = b-a;
    }
}
if ( b=0 )
{
    System.out.print(a);
}
else
{
    System.out.print(b);
}
```

## 3. Datentypen

Um Werte abzuspeichern, muss vorher definiert werden, welche Typen abgespeichert werden sollen. Dies wird zu Beginn eines Java-Programmes definiert. In jedem Java-Programm muss dies genau einmal, beim ersten Benutzen der Variable, passieren.

### ganze Zahlen (byte / short / int / long)

Der einfachste Datentyp ist eine **ganze Zahl**. Damit können Werte wie eine **5** oder **2** abgespeichert werden.

Der Unterschied der Datentypen **byte**, **short**, **int** und **long** ist ihre abspeicherbare Größe. So kann in einer Variable vom Typ **short** ein Wert zwischen **-32768** bis **32767** gespeichert werden<sup>2</sup>.

Im Beispiel wird die Zahl **5** auf **PersonenAnzahl** gespeichert. Weiterhin bekommt die Variable **Alter** den Wert **25**. Es wird die Variable **Stunden** als Ganzzahl definiert.

```
byte PersonenAnzahl = 5;
byte Alter = 25;
int Stunden;
```

### Kommazahlen (float / double)

Um Kommazahlen abzuspeichern, muss man den Datentyp einer **Fließkommazahl** definieren. Damit werden z.B. Werte wie **5,3** oder **3,141** gespeichert.

Der Unterschied der Datentypen **float** und **double** ist ihre abspeicherbare Größe<sup>3</sup>. Beim Schreiben im Java muss das Komma (,) als Punkt (.) geschrieben.

Im Beispiel wird die Zahl **3,141** auf **PI** gespeichert. Weiterhin wird die Variable **Gehalt** als Kommazahl definiert.

```
float PI = 3.141;
float Gehalt;
```

### Zeichen (char / String)

Um Namen oder ähnliches abzuspeichern, muss die Variable vom **Zeichentyp** definiert werden.

Bei Typ **char** kann lediglich ein Zeichen gespeichert werden. Für mehrere Zeichen gibt es den Typ **String**.

In dem Beispiel wird der erste Buchstabe vom **Vornamen** gespeichert. Weiterhin wird die Variable **Wohnort** als Typ **String** gesetzt.

```
char Vorname = "D";
String Wohnort;
```

<sup>2</sup> <http://www.teialehrbuch.de/Kostenlose-Kurse/JAVA/6559-byte-short-int-long.html>

<sup>3</sup> [http://de.wikipedia.org/wiki/Java-Syntax#Primitive\\_Datentypen](http://de.wikipedia.org/wiki/Java-Syntax#Primitive_Datentypen)

## andere Typen

### boolean

Der boolean-Datentyp definiert einen Wert, der nur 1 oder 0 (bzw wahr oder falsch) annehmen kann.

Im Beispiel wird abgespeichert, dass der Nutzer männlich ist. Weiterhin ist dieser Benutzer Nichtraucher.

```
boolean mann = true;  
boolean raucher = 0;
```

### array

Um mehrere Daten desselben Types abzuspeichern, nutzt man Arrays. Arrays sind Felder aus mehrererer Daten. Diese werden in 2 oder 3 Dimensionen abgespeichert. Mehr dazu im Kapitel "Arrays".

Ein Feld "Land" wird mit folgenden Ländern gespeichert: "Deutschland", "England" und "Brasilien".

```
String Land[] = {"Deutschland",  
"England", "Brasilien"};
```

## 4. Arrays

Um Daten des gleichen Types komfortabler abzuspeichern, sollten Arrays benutzt werden. Mit Arrays (oder auch Feldern) werden solche Elemente in einer Reihenfolge aufsortiert.

### 1-dimensionale Arrays

Ein eindimensionales Array sieht beispielsweise folgendermaßen aus:

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Wert	12	9	22	33	9	3	7	22	63	31	10	50	34	9	15	18	27	12	99	25

Das Array besteht aus 20 Elementen (**i von 0 bis 19**). Das erste Element eines Arrays ist immer das Element **0**.

Im Beispiel könnte das Alter von Umfrageteilnehmern abgespeichert worden sein. So ist der Teilnehmer 1 (**i = 0**) 12 Jahre alt. Der 8. (**i = 7**) Teilnehmer ist 22 Jahre alt.

Folgendermaßen wird dieses Feld gespeichert.

```
int Alter[] = {"12", "9", "22", "33",
              "9", "3", "7", "22", "63", "31", "10",
              "50", "34", "9", "15", "18", "27", "12",
              "99", "25"};
```

**Alter** ist der Name des Arrays. Davor kommt der Datentyp (*siehe Datentypen*) der einzelnen Elemente. Da es sich um Zahlen ohne Komma handelt, wird **int** geschrieben. An den **[]** nach dem Namen des Arrays wird in Java definiert, dass es sich um ein Feld handelt. Nach dem Gleichheitszeichen folgen die Werte in geschweiften Klammern (**{}**). Jedes Element wird zwischen Anführungsstriche ("" ) geschrieben.

Alternativ kann auch ein leeres Array definiert werden. Dann sind noch keine Inhalte abgespeichert.

Anlegen eines leeren Feldes.

```
int Alter[];
```

Das Alter des 12. Umfrageteilnehmers (**i = 11**) wird mit **23** (über)schrieben.

```
Alter[11] = "23";
```

Beim Setzen bzw. Überschreiben eines bestimmten Elementes wird der Array-Name (**Alter**) gefolgt von dem zu ändernden Element in eckigen Klammern (**[11]**) geschrieben. Hinter dem Gleichheitszeichen wird der Wert zwischen den Anführungszeichen ("" ) geschrieben, der gesetzt werden soll.

Der Vorteil von Arrays sind die unzähligen Funktionen, mit Ihnen umzugehen. Es gibt Funktionen um diese Elemente zu sortieren o.ä. Ebenso kann man Schleifen konstruieren, um mit jedem Element die gleiche Operation auszuführen.

Jedes Element wird durchlaufen und ausgegeben.

```
int i = 0;
while ( i < Alter.length)
{
    System.out.print(Alter[i]);
    i = i+1;
}
```

Die Schleife dient zur Ausgabe des Alter jeder Umfrageteilnehmer. Zuerst wird eine Laufvariable **i** definiert und auf **0** gesetzt. Die Schleife wiederholt sich so lange, wie **i** kleiner als die Anzahl der Elemente in **Alter** ist. **Alter.length** gibt die Anzahl der Elemente in **Alter** aus. Dies nennt man auch die Größe des Arrays **Alter**. Im Beispiel ist die Größe des Arrays **20**. Da sich **i** bei jedem Durchlauf um eins vergrößert (**i = i + 1**), bei dem Wert 0 startet (**int i=0**) und die Länge des Arrays 20 ist (**Alter.length**) wird die Schleife 20 mal durchlaufen. Bei jedem Durchlauf wird das **ite** Element ausgegeben (**System.out.print**).

Die Schleife gibt somit die Elemente **Alter[0], Alter[1], Alter[2] ... Alter[19]** aus. Sobald **i** den Wert 20 erreicht, wird die Schleife unterbrochen.

## mehrdimensionale Arrays

Das Schöne an Arrays ist, dass sie nicht nur aus einer Dimension bestehen müssen. In 2-dimensionalen Arrays lässt sich z.B. eine Karte realisieren. 3-dimensionale Arrays sind praktisch bei Karten mit Höhenprofilen.

	i	0	1	2	3	4	5	6	7
j	Bedeutung	<i>Sieben</i>	<i>Acht</i>	<i>Neun</i>	<i>Zehn</i>	<i>Bube</i>	<i>Dame</i>	<i>König</i>	<i>Ass</i>
0	Herz	<u>Keiner</u>	Hans	Max	Hans	Dieter	Dieter	Max	Dieter
1	Pik	Max	Dieter	Dieter	Hans	Dieter	Max	Hans	Hans
2	Karo	Hans	Max	Hans	Max	<u>Keiner</u>	Hans	Dieter	Max
3	Kreuz	Max	Hans	Dieter	Dieter	Max	Hans	Max	Dieter

Im Beispiel wird in einem 2-dimensionalen Array abgespeichert, wer beim Skat welche Karte hat. Mitspieler sind Max, Hans und Dieter. Weiterhin sind 2 Karten im Besitz von niemandem.

Das Feld **Skat** wird definiert (aus Platzgründen jedoch nur auszugsweise beschrieben)

```
String Skat[][] = {"Keiner", "Max",
                  "Hans", "Max"}, {"Hans", "Dieter",
                  "Max", "Hans"}, {"Max", "Dieter",
                  "Hans", "Dieter"}, {"Hans", "Hans",
                  "Max", "Dieter"} ..... ;
```

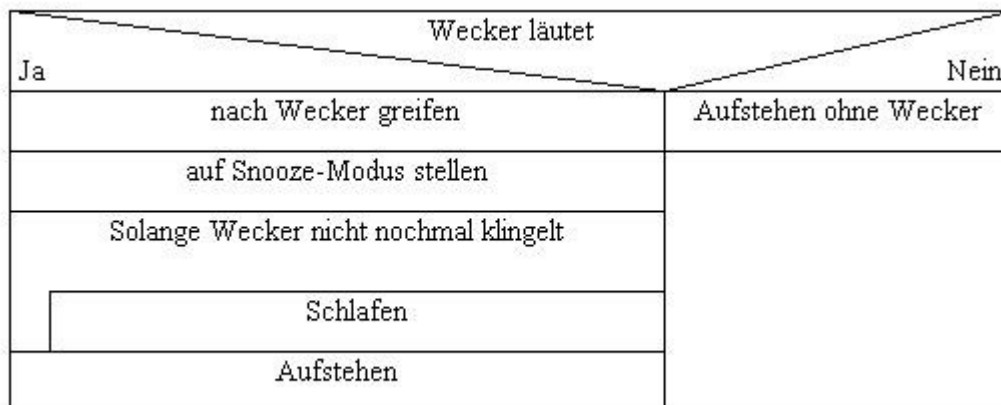
Hans bekommt die Pik-Dame.

```
Skat[1][5] = "Hans";
```

## 5. Funktion

Um bestimmte Befehle auszulagern und sie damit oft zu wiederholen, gibt es Funktionen. Funktionen beschreiben einen bestimmten Ablauf, der immer gleich ist.

Im Alltag ist eine Funktion z.B. der Ablauf des Aufwachens. Man könnte ihn folgendermaßen Beschreiben.



Dies sind Programmteile eines Struktogramms. Das Besondere ist, dass wir den kompletten Vorgang als **Aufwachen** bezeichnen. Wenn wir dies so definieren, reicht es, einmal das **Aufwachen** aufzurufen. Dann werden immer dieselben Befehle ausgeführt.

Die Funktion **Aufwachen** wird definiert. Sie schreibt "Wecker klingelt".

```
public void Aufwachen()
{
    System.out.print("Wecker klingelt");
}
```

Vor dem Namen der Funktion **Aufwachen** werden Eigenschaften geschrieben. Die Eigenschaft **public** beschreibt die Möglichkeit, von wo aus die Funktion **Aufwachen** definiert ist. Welche Parameter es gibt, liest man [hier](#). Generell wird aber nur die Eigenschaft **public** benötigt. Mit der Eigenschaft **void** wird der Datentyp des Rückgabewertes definiert. Hier sind also auch andere Datentypen möglich (*siehe Datentypen*). **void** bedeutet, dass kein Wert zurückgegeben wird. Funktionen ohne Rückgabewert heißen **Methoden**.

Die Funktion **PI** wird definiert. Sie gibt PI mit drei Stellen nach dem Komma **aus**.

```
public void PI()
{
    System.out.print("3.141");
}

PI();
```

In der Funktion **PI** wird die Ausgabe des Wertes **3.141** geregelt. Später ruft der Aufruf **PI();** die Funktion an dieser Stelle auf.

Die Funktion **PI** wird definiert. Sie gibt PI mit drei Stellen nach dem Komma **zurück**.

```
public double PI()
{
    return 3.141
}

wertPI = PI();
System.out.print(wertPI);
```

Der zweite Quellcode realisiert dasselbe wie der Quellcode 1. Jedoch wird dabei die Zahl PI mit 3 Stellen nach dem Komma nicht direkt in der Funktion ausgegeben, sondern zurückgegeben. Dies erkennt man an der Zeile **return 3.141** sowie am Typus **double** vor dem Funktionsnamen **PI()**. Mit **double** wird eine Kommazahl als Rückgabewert definiert. Der Befehl **return 3.141** schreibt den Wert **3.141** an diese Stelle, wo die Funktion mit **PI();** aufgerufen wird. Diese wird dann auf **wertPI** gespeichert und ausgegeben.

Funktionen können nicht nur Werte zurückgeben. Ebenso können Werte an eine Funktion übergeben werden. Dann kann mit den Werten etwas gemacht werden.

Die Funktion **PI** wird definiert. Sie erwartet die Anzahl der Stellen als Übergabewert. Sie gibt PI mit der Anzahl der Stellen aus, die übergeben wurden.

```
public void PI(int stellen)
{
    if(stellen=="1")
    {
        System.out.print("3.1");
    }
    if(stellen=="2")
    {
        System.out.print("3.14");
    }
    if(stellen=="3")
    {
        System.out.print("3.141");
    }
}

PI(2);
```

Mit **int stellen** wird definiert, dass eine Ganzzahl übergeben werden muss, die in der Funktion **stellen** genannt wird. Weitere Übergabewerte werden mit einem Komma abgetrennt. Die Funktion mit mit **PI(2);** aufgerufen. Damit hat **stellen** den Wert **2**.

## 6. Rekursion

Ein rekursiver Algorithmus ist eine Funktion (*siehe Funktion*), die sich selbst aufruft. Damit lassen sich Schleifen o.ä. realisieren.

Die Funktion **Summe** soll die Summe aus einer Zahl und allen kleineren ganzen Zahlen berechnen. Somit soll sie bei der Eingabe von **5** insgesamt **5+4+3+2+1+0** berechnen.

```
public int Summe(int summand)
{
    if (summand > 0)
    {
        return Summe(summand-1) + summand;
    }
    else
    {
        return 0;
    }
}

ergebnis = Summe(5);
System.out.print(ergebnis);
```

Mit **Summe(5)** wird der Startwert **5** an die Funktion **Summe** übergeben. Dort wird geschaut, ob der Wert größer als 0 ist. Wenn dies der Fall sein sollte, wird **Summe(summand 1)+summand** aufgerufen. Sollte der Wert **summand** also **5** haben, dann wird **Summe(4)+5** zurückgegeben. **Summe(4)** wiederum gibt **Summe(3)+4** zurück. Dies wird bis zum Ende (bis der **summand** den Wert **0** hat) fortgeführt. Folgende Funktionen werden aufgerufen.

Rückgabewert	summand
Summe(5)	5
Summe(4) + 5	4
Summe(3) + 4 + 5	3
Summe(2) + 3 + 4 + 5	2
Summe(1) + 2 + 3 + 4 + 5	1
Summe(0) + 1 + 2 + 3 + 4 + 5	0
0 + 1 + 2 + 3 + 4 + 5	-

Zu Beginn ist der Wert, der ausgegeben wird, **Summe(5)**. Da dies eine Funktion ist, wird sie aufgerufen. Diese wird gibt jedoch **Summe(4)+1** zurück. Jedoch ist **Summe(4)** wieder eine Funktion und kein fester Wert. Deswegen wird die Funktion **Summe()** mit dem Wert **4** aufgerufen. Das **+5** ist jedoch fest und wird nicht verändert. **Summe(4)** wird mit **Summe(3) + 4** ersetzt. **Summe(3)** ist wiederum **Summe(2) + 3**. Dies wird weiter fortgesetzt.

## 7. Klassen / Objekte

Klassen bzw. Objekte dienen zur Gruppierung (Klassifizierung) von Funktionen. Dadurch stehen bei einer Vielzahl von Funktionen alle Funktionen in einem Zusammenhang.

Im Alltag klassifizieren wir ebenso. So wird z.B. nach der Art der Lebewesen klassifiziert. So gibt es die Art der **Säugetiere** oder **Vögel**. Jede Gruppe hat spezifische **Merkmale** und **Funktionen**.

**Säugetiere** haben ein **Gebiss**, ein **Zwerchfell** sowie **Gliedmaßen**. Weiterhin können sie **säugen** und **laufen**.

**Vögel** wiederum haben einen **Schnabel** und ebenso **Gliedmaßen**. Sie können **fliegen** und **laufen**.

Die Klassen **Säugetiere** sowie **Vögel** werden mit den **Eigenschaften** und **Funktionen** definiert.

```
public class Säugetiere
{
    boolean Gebiss = true;
    boolean Zwerchfell = true;
    int Gliedmaßen = 4;
    String Name;

    public static void säugen()
    {
    }
    public static void laufen()
    {
    }
}

public class Vögel
{
    boolean Schnabel = true;
    int Gliedmaßen = 2;
    String Name;

    public static void fliegen()
    {
    }
    public static void laufen()
    {
    }
}
```

Beim Definieren der Klassen werden vor dem Klassennamen die Eigenschaften dieser Klasse geschrieben. Mit **public** wird eine Klasse definiert, die überall einbindbar ist. Mit **class** wird definiert, dass es sich um eine Klasse handelt.

In der Klasse gibt es **Eigenschaften** sowie **Funktionen**.

Das **Pferd** wird in die Klasse der **Säugetiere** einsortiert.

```
Säugetiere Pferd = new Säugetiere();  
Pferd.Name = "Kleiner Onkel";  
Pferd.laufen();
```

Das **Pferd** wird in die Klasse der **Säugetiere** einsortiert. Damit hat das Pferd ein **Objekt** der **Klasse** Säugetiere. Objekte sind die benutzten Klassen. Weiterhin wird der **Name** des Pferdes auf "Kleiner Onkel" gesetzt. Weiterhin beginnt das Pferd zu **laufen**. Die Ausführungen in der Funktion **laufen()** werden ausgeführt.

## Vererbung

Vererbung bietet die Möglichkeit Klassen weiter zu deklarieren. So ist z.B. das **Säugetier** ein **Lebewesen**. Das **Säugetier** hat also alle **Eigenschaften** und **Funktionen** der **Lebewesen**.

Die Klasse **Lebewesen** hat die Funktion **leben**. Das **Säugetier** hat die Klasse **Lebewesen** als Mutterklasse. Weiterhin ist die Funktion **laufen** definiert. Der **Löwe** wird als **Säugetier** deklariert.

```
public class Lebewesen  
{  
    public static void leben()  
    {  
    }  
}  
public class Säugetier extends Lebewesen  
{  
    public static void laufen()  
    {  
    }  
}  
Säugetier Löwe = new Säugetier();  
Löwe.leben();
```

Im Codebeispiel bekommt die Klasse **Lebewesen** die Eigenschaft **leben()**. Jedes Lebewesen lebt. Das **Säugetier** hat als Mutterklasse die **Lebewesen**. Dies wird durch das Schlüsselwort **extends** definiert. Weiterhin hat es selber eine Funktion (**laufen()**). Der Löwe wird als **Säugetier** deklariert. Jetzt hat er ebenso die Funktion **leben()**, die mit **Löwe.leben()** aufgerufen wird.

## Funktionen überschreiben

Bei der Vererbung der Klassen können auch Funktionen überschrieben werden. Damit definiert man eine bereits vorhandene Funktion der Elternklasse neu.

Die Klasse **Lebewesen** hat die Funktion **Gattung**. Das **Säugetier** hat ebenso die Funktion **Gattung**. Der **Löwe** wird als **Säugetier** deklariert.

```
public class Lebewesen
{
    public static void Gattung()
    {
        System.out.print("keine Gattung bekannt");
    }
}
public class Säugetier extends Lebewesen
{
    public static void Gattung()
    {
        System.out.print("ich bin ein Säugetier");
    }
}
Säugetier Löwe = new Säugetier();
Löwe.Gattung();
```

Die Funktion **Gattung()** der Klasse **Lebewesen** gibt die Art des Lebewesens aus. Zu Beginn wird "**keine Gattung bekannt**" ausgegeben. Es wurde noch keine Gattung definiert. Die Funktion **Gattung** wird in der Klasse **Säugetier** überschrieben. Jetzt gibt sie "**Ich bin ein Säugetier**" aus.

## Funktionen überladen

Es ist in Java auch möglich gleiche Funktionsnamen für verschiedene Funktionen doppelt zu vergeben.

Die Klasse **Figur** hat die Funktion **Name()** sowie 2 weitere Funktionen **Name**, die zwar gleich heißen aber sich durch die Übergabewerte unterscheiden.

```
public class Figur
{
    public static void Name()
    {
        System.out.print("unbekannte Figur!");
    }
    public static void Name(int ecken)
    {
        if(ecken=="4")
        {
            System.out.print("Viereck!");
        }
        if(ecken=="3")
        {
            System.out.print("Dreieck!");
        }
    }
    public static void Name(String name)
    {
        if(name=="Viereck")
        {
            System.out.print("Viereck!");
        }
        if(ecken=="Dreieck")
        {
            System.out.print("Dreieck!");
        }
    }
}
Figur unbekannt = new Figur();
Figur.Name();

Figur Dreieck = new Figur();
Figur.Name(3);

Figur Viereck = new Figur();
Figur.Name(Viereck);
```

Die Klasse **Figur** hat 3 Möglichkeiten, den Namen der Figur auszugeben. Die Funktion **Name()** wird benutzt, wenn keine Eigenschaften bekannt sind. Die Funktion **Name(int ecken)** wird mit einer Zahl als Weitergabewert aufgerufen. Sobald eine Zeichenkette weitergegeben wird, hat die Funktion **Name(String name)** Vorrang.

## Konstruktor

Konstruktoren bieten die Möglichkeit sofort etwas auszuführen, sobald die Klasse gestartet wird bzw. das Objekt aus der Klasse erstellt wird. Ein Konstruktor ist eine Funktion mit dem selben Namen wie die Klasse.

Die Klasse **Tier** hat eine Funktion **Tier()**. Diese ist der Konstruktor dieser Klasse. Später wird der **Vogel** als **Tier** definiert.

```
public class Tier
{
    public static void Tier()
    {
        System.out.print("Tier erstellt");
    }
}
Tier Vogel = new Tier();
```

Wenn eine Klasse dieselbe Funktion wie ihr Name hat, ist diese Funktion ihr Konstruktor. Diese Funktion (hier **Tier()**) wird aufgerufen, sobald ein Objekt auf der Klasse **Tier** erzeugt wird (**Tier Vogel = new Tier();**). Somit erscheint "**Tier erstellt**" auf dem Bildschirm.

Sinn ergibt ein Konstruktor dann, wenn Werte definiert werden.

Die Klasse **Tier** hat eine Funktion **Tier()** sowie die Variable **Gliedmaßen**. Später wird der **Vogel** als **Tier** mit dem Wert **2** definiert.

```
public class Tier
{
    int Gliedmaßen;

    public static void Tier(int beine)
    {
        Gliedmaßen = beine;
    }
}
Tier Vogel = new Tier(2);
```

Bei der Erzeugung des Objektes **Tier Vogel = new Tier(2);** wird **2** übergeben. Damit wird die Variable **beine** auf 2 gesetzt. Später wird dies auf die **Gliedmaßen** gespeichert. Alternativ lässt sich auch folgendes schreiben: **Vogel.Gliedmaßen = 2;**

## 8. Suchalgorithmen

### Sequenzielle Suche

Die sequenzielle Suche ist der einfachste, aber auch aufwendigste Suchalgorithmus. Er benötigt keine sortierte Zahlenkette. Jedes einzelne Element wird von vorn nach hinten durchgeschaut und mit dem gesuchten Wert verglichen.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position													

Die Zahl **148** wird gesucht. Der Algorithmus der **sequenziellen Suche** beginnt von links zu vergleichen, ob die gesuchte Zahl gleich, größer oder kleiner als die gegebene Zahl ist.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position	X												

Im ersten Feld steht die Zahl **0**. Diese ist kleiner als die gesuchte Zahl. Deswegen wird im nächsten Feld weiter gesucht.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position	X	X											

Auch dort ist eine Zahl (**2**) die kleiner ist, als der gesuchte Wert. Genauso wird mit den nächsten Elementen fortgefahren.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position	X	X	X	X	X	X	X	X	X	X	X	X	X

Am Ende wird die erste Zahl erreicht. Es wurden 13 Vergleiche (in diesem Beispiel) durchgeführt. Die Suche ist abgeschlossen.

In diesem Beispiel wird 6x verglichen.

## Binäre Suche

Die Binäre Suche ist ein optimiertes Suchverfahren. Dieser Algorithmus setzt ebenso eine sortierte Suchkette voraus.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position													

Die Zahl **148** wird gesucht. Der Algorithmus der **binären Suche** beginnt den Vergleich der gesuchten Zahl mit dem mittleren Element. Dies ist hier das Element 7.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position							X						

Die Zahl **163** ist größer als die gesuchte Zahl **148**. Bei einer sortierten Kette ist bekannt, dass rechts dieses Elementes die Zahlen größer als **163** sind. Deswegen muss die Zahl **148** links vom Element 7 gesucht werden. Nur die Elemente 1-6 sind jetzt noch relevant. Das mittlere Element kann die 3 oder 4 sein. Wir legen das Element 3 fest.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position			X				X						

Dort ist die Zahl **25**, die kleiner als die gesuchte **148** ist. Jetzt muss nur noch zwischen den Elementen 3 und 7 gesucht werden. Das mittlere Element dieser 3 restlichen Elemente (4, 5, 6) ist die Nummer 5.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position			X		X		X						

Der Wert im Element 5 ist kleiner als die gesuchte Zahl (**147 < 148**) damit muss jetzt mit einem Element hinter Element 5 gesucht werden. Deswegen wird der Wert auf Element 6 geprüft.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13
Wert	0	2	25	75	147	150	163	183	260	320	552	627	889
Position			X		X	X	X						

Das Element 6 hat den Wert **150**. Dies ist nicht der gesuchte Wert. Jetzt ist bekannt, dass der gesuchte Wert **148** nicht vorhanden ist. Die Suche ist beendet.

Im Gegensatz zur sequenziellen Suche, werden bei der binären Suche in dem Beispiel nur 4 Vergleiche benötigt.