



## Aufbau eines Java-Programms

```
public class Klassenname {
    public static void main(String[] args) {
        // Programmcode
    }
}
```

## Compiler

javac *Klassenname*.java übersetzt Programm in Bytecode  
 java *Klassenname* VM führt Bytecode aus

⚠ Zu obigen Rahmenprogramm, muss der Dateiname dem Klassennamen entsprechen!

## Kommentare

von // bis zum Ende der Zeile  
 von /\* bis \*/ eingeschlossener Block (mehrzeilig)

## Ausgabe

System.out.println(Ausgabe); mit Zeilenumbruch  
 System.out.print(Ausgabe); ohne Zeilenumbruch  
 z.B. Ausgabe: "Text1" + Variable + "Text2"

## Primitive Datentypen

- sind in die Sprache integriert  
 - Größe, Wertebereich und Operationen sind definiert

Name	Größe in Byte	Wertebereich
byte	1 (8 Bit)	$-2^7 - 2^7-1$
short	2 (16 Bit)	$-2^{15} - 2^{15}-1$
int	4 (32 Bit)	$-2^{31} - 2^{31}-1$
long	8 (64 Bit)	$-2^{63} - 2^{63}-1$
float	4 (32 Bit)	$\pm 3.4 \cdot 10^{38}$
double	8 (64 Bit)	$\pm 1.79 \cdot 10^{308}$
char	2 (16 Bit)	Unicode
boolean	1 (8 Bit)	true, false

## Automatische Typkonvertierung

byte → short → int → long → float → double  
 char ↗

## Konstanten

final Datentyp KONSTANTE = Wert;  
 z.B. final int JAHR = 2017;

## Regeln für Bezeichner

- Name für Variable, Methode, Datentyp
- Folge von Buchstaben und Ziffern
- beginnen mit einem Buchstaben
- \$ und \_ zählen als Buchstaben
- Groß- und Kleinschreibung wird unterschieden
- dürfen keine Schlüsselworte sein

## Variablendeklaration

Datentyp variablenname;

z.B. int zahl;  
 int a, b, c;

## Deklaration (Definition) mit Initialisierung

int zahl = 5;  
 String name = "Rudi";

## Daten einlesen

### Die Klasse Scanner

```
import java.util.Scanner;
public class Klassenname {
    public static void main(String[] args) {
        Scanner ein = new Scanner(System.in);
```

### ganze Zahlen einlesen (Rückgabotyp int)

z.B. izahl = ein.nextInt();

### gebrochene Zahlen einlesen (Rückgabotyp double)

z.B. dzahl = ein.nextDouble();

### Zeichenketten einlesen (Rückgabotyp String)

z.B. wort = ein.next();

### Zeile einlesen; inklusive Leerzeichen (Rückgabotyp String)

z.B. zeile = ein.nextLine();

### Die Klasse BufferedReader

```
import java.io.*;
public class Klassenname {
    public static void main(String[] args)
        throws IOException {
        BufferedReader ein = new BufferedReader
            (new InputStreamReader(System.in));
```

### Zeile einlesen; inklusive Leerzeichen (Rückgabotyp String)

z.B. zeile = ein.readLine();

### Zeichenketten (String) in Zahlen umwandeln

#### String in ganze Zahl umwandeln (int izahl)

z.B. izahl = Integer.parseInt(zeile);

#### String in gebrochene Zahl umwandeln (double dzahl)

z.B. dzahl = Double.parseDouble(zeile);

## Operatoren

### Assoziativitäten (Klammerung)

rechtsassoziativ: Präfixoperatoren, bedingter Ausdruck, Zuweisungsoperatoren

linksassoziativ: alle anderen Operatoren

### Prioritäten

Pri	Operatoren	Bemerkung
1	o.a. i++, a[], f(), .	Postfix-Operatoren
2	-x, !, ~, ++i	Präfix-Operatoren
3	new C(), (type) x	Objekt-Erzeugung, Cast
4	*, /, %	Multiplikation, etc.
5	+, -	Addition, Subtraktion
6	<<, >>, >>>	Shift
7	<, >, <=, >=	kleiner, größer, etc.
8	==, !=	gleich, verschieden
9	&	bitweise UND
10	^	bitweise XOR
11		bitweise ODER
12	&&	logisches UND
13		logisches ODER
14	?:	bedingter Ausdruck
15	=, +=, -=, *=, /=	Zuweisungen

## Kontrollstrukturen

### Bedingte Anweisung (Falls ... sonst ...)

```
if (Bedingung) {
    // Anweisungen werden ausgeführt,
    // falls Bedingung wahr ist
}
else {
    // Anweisungen werden ausgeführt,
    // falls Bedingung nicht wahr ist
}
```

## Fallunterscheidung (Auswahl)

```
switch (Variable) {
    case Wert1: // Einstiegspunkt für den Fall
                // Variable == Wert1
                break;
    case Wert2: // Einstiegspunkt für den Fall
                // Variable == Wert2
                break;
    ...
    default: // Anweisungen, falls die
            // Variable keinen der zur
            // Auswahl stehenden Werte
            // enthält
}
```

⚠ Der Datentyp von Variable darf nur ein primitiver Datentyp, String oder eine Wrapperklasse sein. Wert1, Wert2, ... sind **Konstanten!**

## Wiederholungen

### while – Schleife (solange ...)

```
while (Bedingung) {
    // Anweisungen, welche ausgeführt werden,
    // solange die Bedingung erfüllt ist
}
```

### do while – Schleife (mindestens einmal aber solange ...)

```
do {
    // Anweisungen, werden mindestens einmal
    // ausgeführt, und dann so oft wiederholt,
    // solange die Bedingung erfüllt ist
} while (Bedingung);
```

### for – Schleife (Spezialfall: Zählschleife)

```
for (Initialisierung; Bedingung; Schrittweite) {
    // Anweisungen
}
```

Beispiel: Schleife wird 10 mal durchlaufen

```
for (int i = 0; i < 10; i++) {
    // Anweisungen
}
```

### for each – Schleife für Arrays

```
double[] a = new double[10];
```

```
for (double x : a) {
    System.out.println(x);
}
```

## vorzeitiges Beenden einer Schleife

## continue

- springt zum Ende des Schleifenrumpfes

## break (Abbruch)

- Schleife / switch wird an dieser Stelle verlassen und nächste Anweisung hinter der Schleife / dem switch wird ausgeführt

## Exceptions

- `ArrayIndexOutOfBoundsException` (ungültiger Index)  
- `StringIndexOutOfBoundsException` (ungültige Position)  
- `NullPointerException` (Zugriff auf null – Referenz)  
- `NumberFormatException` (String enthält keine Zahl)  
- `ArithmeticException` (z.B. Division durch 0)

Exceptionbehandlung:

```
try {
    // Anweisungsblock
}
catch (Ausnahme1) {
    // Anweisungen zur Fehlerbehandlung
}
catch (Ausnahme2) {
    // Anweisungen zur Fehlerbehandlung
}
...
finally {
    // wird, wenn vorhanden, immer ausgeführt
}
```

## Methoden überladen

- überladene Methoden = Methoden mit gleichem Namen aber mit **unterschiedlichen Parameterlisten** innerhalb einer Klasse

1. Variante: **unterschiedliche Anzahl der Parameter**
2. Variante: **mindestens ein Parameter hat anderen Datentypen**

- Zweck: semantisch ähnliche Aktionen für unterschiedliche Datentypen (Siehe Methode `System.out.println(...)`)

```
public void println()
public void println(String s)
public void println(int x)
...
```

## Referenztypen

### Arrays

#### eindimensionale Arrays

Eigenschaften: - Elemente liegen im Speicher hintereinander  
- Elemente sind alle vom gleichen Datentyp

```
// Deklaration der Referenzvariablen
int [] zahlenfeld;
// Speicherplatz für Array anlegen
zahlenfeld = new int[Länge];
```

oder

```
int [] zahlenfeld = new int[Länge];
```

```
// Deklaration + Initialisierung
int [] gerade = { 2, 4, 6, 8, 10 };
```

#### Länge des Arrays

```
zahlenfeld.length
```

#### Zugriff auf ein Element des Arrays

- Zugriff über Index

- kleinster Index: **0**; größter Index: **Länge-1**

z.B. `zahlenfeld[index] = 7;`

#### mehrdimensionale Arrays

⚠ In Java ist ein zweidimensionales Array eigentlich ein eindimensionales Array, das selbst wieder Referenzen auf eindimensionale Arrays als Elemente enthält.

- Anzahl der Klammerpaare bestimmen die Dimension

```
int [][] matrix; // Deklaration 2 dim.
int [][][] quader; // Deklaration 3 dim.
```

```
matrix = new int[5][10]; // 2 dim.
Quader = new double[5][10][2]; // 3 dim.
```

#### Länge des Arrays

```
matrix.length // Größe der 1. Dimension
matrix[2].length // Länge der 2. Zeile
                // der 2. Dimension
```

#### Zugriff auf ein Element des Arrays

```
matrix[0][0] = 7; // 2 dim
quader[0][0][0] = 7; // 3 dim
```

## Kommandozeilenparameter

- werden beim Start des Programms an dieses übergeben  
- können über das Array `args` der `main`-Methode abgerufen werden

```
java Programmname param1 param2 ...
```

```
public static void main(String[] args) {
    System.out.println(args[0]); → param1
    System.out.println(args[1]); → param2
    ...
}
```

## Konzepte der Objektorientierten Programmierung

### 1. Konzept: Kapselung von Daten und deren verarbeitenden Funktionen (Klassen)

#### Aufbau einer Klasse

```
class NamederKlasse {  
    // Attribute (Variablen/Eigenschaften/  
    //           Instanzvariablen)  
    // Methoden (Verarbeiten der Attribute)  
}
```

Klasse ist ein Datentyp (gibt vor, wie der Speicher für eine Instanz (Objekt) aussehen soll)

**Klasse = eine Schablone für viele Objekte!**

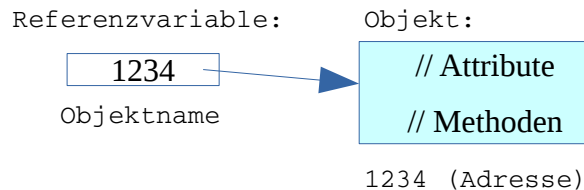
#### Objekte / Instanzen einer Klasse

##### Referenzvariable deklarieren

```
NamederKlasse objektname;
```

##### Objekt erzeugen und der Referenzvariablen zuweisen:

```
objektname = new NamederKlasse();
```



Operator new: Reservieren des Speicherplatzes

#### Methoden

- dienen hauptsächlich zur Kommunikation mit dem Objekt (z.B. Werte an das Objekt übergeben [Zustand des Objektes ändern] oder Attribute aus dem Objekt auslesen [Zustand des Objektes abfragen]) → bestimmen das Verhalten

```
Rückgabetyyp Methodenname (Parameterliste) {  
    // Anweisungen  
  
    // bei Bedarf Rückgabe eines Wertes  
    // return Ausdruck;  
}
```

Methode gibt **keinen Wert** zurück

→ Rückgabetyyp = **void**

return-Anweisung wertet den Ausdruck aus, beendet die Methode und gibt den Wert des Ausdrucks zurück

## Zugriff auf Komponenten eines Objektes

- Zugriff erfolgt über den Operator .

```
Objektname.Attribut      bzw.  
Objektname.Methode (Parameterliste)
```

#### Die Klasse Object

- Basisklasse für alle Klassen des Programms

- enthält folgende wichtigen Methoden:

**Kopie eines Objektes erzeugen**

```
protected Object clone()
```

**Vergleich zweier Objekte**

```
boolean equals (Object o)
```

@Override

```
public boolean equals (Object o) {  
    if (o == null) {return false;}  
    if (! o instanceof NamederKlasse)  
        {return false;}  
    if (o == this) {return true;}  
  
    // Typumwandlung von o mit (cast)  
    NamederKlasse oneu = (NamederKlasse) o;  
  
    // Eigener Vergleich mit oneu  
    ...  
}
```

**Liefert String-Darstellung eines Objektes**

```
String toString()
```

@Override

```
public String toString() {  
    return "Zeichenkette";  
}
```

#### Konstruktoren

- spezielle Methoden

- dienen zum Initialisieren eines Objektes (der Attribute bzw. des Speicherplatzes)

- werden direkt beim Erzeugen des Objektes aufgerufen

- Klasse kann mehrere Konstruktoren enthalten

Eigenschaften:

1. besitzen keinen Rückgabetyyp
2. haben den gleichen Namen wie die Klasse

```
// Konstruktor  
NamederKlasse (Parameterliste) {  
    ...  
}
```

#### Zugriffsrechte

- stehen direkt (private, public) / oder indirekt (package default) **vor** jedem Attribut

**private**

- erlaubt den Zugriff **nur** für Objekte dieser Klasse

**package default**

- erlaubt den Zugriff für alle Objekte aus dem gleichen Paket

**public**

- erlaubt den Zugriff für Objekte aus einem beliebigen Paket

**this**

- enthält eine Referenz auf das Objekt selbst  
Über diese kann auf alle Komponenten des Objektes zugegriffen werden.

```
this.Attribut      bzw.  
this.Methode (Parameterliste)
```

#### Klassenattribute/Klassenmethoden (static)

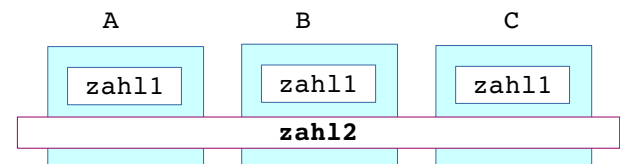
- existieren nur einmal für alle Objekte einer Klasse

- existieren auch dann, wenn es kein einziges Objekt der Klasse gibt

**!** static-Methoden können nur auf andere static-Komponenten zugreifen!

```
class NamederKlasse {  
    int zahl1;  
    static int zahl2;  
    ...  
}
```

```
A = new NamederKlasse();  
B = new NamederKlasse();  
C = new NamederKlasse();
```



Zugriffsmöglichkeiten auf das gemeinsame Klassenattribut:

A.zahl2

B.zahl2

C.zahl2

**NamederKlasse.zahl2**

## Vererbung / Erweitern

### 2. Konzept: Übertragen von Attributen und Methoden auf eine andere Klasse

#### Zugriffsrechte – Fortsetzung

##### protected

- erlaubt den **Zugriff** für Objekte aus dem gleichen Paket und **für Objekte der Subklassen** aus anderen Paketen

#### Basisklasse (Oberklasse)

```
class BasisKlasse {
```

private Komponenten

protected Komponenten

package default Komponenten

public Komponenten

```
}
```

#### Subklasse (Unterklasse)

Eine Subklasse erweitert die Basisklasse um zusätzliche Attribute und Methoden.

```
class SubKlasse extends BasisKlasse {
```

zusätzliche Attribute und Methoden

Zugriff auf Komponenten der Basisklasse

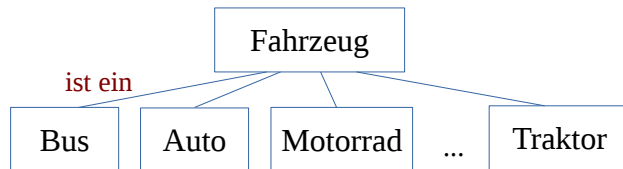
protected Komponenten

package default Komponenten

public Komponenten

```
}
```

#### Beispiel: Realisierung von Hierarchien



Ein Bus **ist ein** Fahrzeug.

Jede Instanz der Subklasse ist auch eine Instanz der Basisklasse  
Fahrzeug F1 = new Auto(); // **Polymorphie**

## Konstruktoren

- Konstruktoren der Basisklasse sind in der Subklasse **nicht sichtbar**, obwohl sie enthalten und `public` sind
- Es werden immer die Konstruktoren der Basisklasse (bzw. der gesamten Hierarchie) aufgerufen
- Reihenfolge: Konstruktor der Basisklasse zuerst, dann die Konstruktoren entlang der Hierarchie
- expliziter Aufruf eines Konstruktors der Basisklasse erfolgt mit `super(Parameterliste);`
- Der explizite Aufruf des Konstruktors der Basisklasse ist immer die erste Anweisung im Konstruktor der Subklasse

#### Referenz **super**

- erlaubt in einem Objekt einer Subklasse den Zugriff auf die geerbten Komponenten der unmittelbar übergeordneten Basisklasse
- erlaubt somit den **Zugriff auf Konstruktoren, überschriebene Methoden und verdeckte Attribute**

#### verdeckte Attribute

- Attribute der Subklasse, welche den gleichen Namen haben, wie Attribute der Basisklasse, verdecken **in den Methoden der Subklasse** die geerbten Attribute der Basisklasse

```
class B {
    public int a;
    public B() {a = 5;}
}
```

```
class S extends B {
    public int a;
    public int basis_a;
    public S() {
        a = 10;
        basis_a = super.a;
    }
}
```

```
S testS = new S();
System.out.println(testS.a);           → 10
System.out.println(testS.basis_a);    → 5
```

#### // **Polymorphie**

```
B testBS = new S();
System.out.println("\n " + testBS.a); → 5
```

#### **final** - Modifikator

- `final` definierte Methoden können in der abgeleiteten Klasse nicht mehr überschrieben werden
- `final` definierte Klassen können nicht vererbt werden

## Überschreiben von Methoden

- Anpassen der Methode an die Anforderungen der Subklassen
- Methode muss in der Subklasse die gleiche Signatur (gleicher Name, Parameterliste) und einen kompatiblen Rückgabtyp besitzen
- Überschriebene Methoden der Basisklasse sind nur noch aus den Methoden der Subklasse und dort über die explizite Angabe von `super` erreichbar

```
class BM {
    public int zb;
    public BM() {zb = 2;}
    @Override public String toString () {
        return "zb = " + zb; }
}
class SM extends BM {
    public int zs;
    public SM() {zs = 20;}
    @Override public String toString () {
        return super.toString() +
            "\n zs = " + zs; }
}
```

```
SM testSM = new SM();
System.out.println(testSM);
```

zb = 2  
zs = 20

#### // **Polymorphie**

```
BM testBMSM = new SM();
System.out.println(testBMSM);
```

#### abstrakte Basisklassen

- Definieren von Grundeigenschaften in einer Basisklasse
- von einer abstrakten Klasse können keine Instanzen erzeugt werden
- abstrakte Basisklasse kann **abstrakte Methoden = Methoden ohne Implementierung** enthalten

```
abstract class Abasis {
    int z;
    abstract public void print (int a);
    public int get_z () {return z;}
}
```

- Klasse mit abstrakten Methoden, muss selbst als `abstract` definiert werden
- eine abstrakte Methode kann nicht `private` sein

## generische Klassen

- Schablone für eine Menge von Klassen
- Klassen der Menge unterscheiden sich nur im Datentyp einiger Attribute (generische Attribute)

```
class A {
    int att;
    ...
}
class B {
    String att;
    ...
}
class C {
    Person att;
    ...
}
```

Instanz erzeugen:

```
A aobj = new A();
B bobj = new B();
C cobj = new C();
```

- in generischer Klasse wird ein Platzhalter für einen Datentyp an betroffenen Stellen eingefügt

```
class<Platzhalter> ABCgen {
    Platzhalter att;
    ...
    public Platzhalter getAtt () {return att;}
    public void setAtt (Platzhalter w) {
        att = w; }
    @Override public boolean equals(Object o) {
        if(o instanceof ABCgen<?>) {
            ABCgen<?> p = (ABCgen<?>) o;
            ...
        }
    }
}
```

- generische Klasse kann nur eingeschränkt Funktionalitäten auf generischen Attributen bereitstellen
- bei Erzeugung einer Instanz von einer generischen Klasse muss der Datentyp, für welchen der Platzhalter steht, angegeben werden
- Datentyp muss selbst ein Referenztyp sein

```
ABCgen<Integer> aobj =new ABCgen<Integer>();
ABCgen<String> bobj = new ABCgen<String>();
ABCgen<Person> cobj = new ABCgen<Person>();
```

## ArrayList

- speichert Elemente in einem internen Array
- Größe im Vergleich zu Arrays veränderbar

```
import java.util.ArrayList;
...
ArrayList<Datentyp> aliste;
aliste = new ArrayList<Datentyp>();
```

## Methoden der Klasse ArrayList

Element an einer bestimmten Position ausgeben

```
aliste.get(Position)
```

Element am Ende der Liste einfügen

```
aliste.add(Element)
```

Element mit Positionsangabe einfügen

```
aliste.add(Position, Element)
```

Alle Elemente einer anderen Liste einfügen

```
aliste.addAll(Position, liste2)
```

Element an Position verändern

```
aliste.set(Position, Element)
```

Element löschen / Liste leeren

```
aliste.remove(Element) / aliste.clear()
```

Element bestimmter Position löschen

```
aliste.remove(Position)
```

Gibt Größe des Arrays zurück

```
aliste.size()
```

Prüft, ob bestimmtes Element enthalten ist (liefert true, false)

```
aliste.contains(Element)
```

Liefert den Index eines Elementes

```
aliste.indexOf(Element)
```

## Interfaces / Schnittstellen

- Interface schreibt einer Klasse vor, welche Methoden die Klasse implementieren muss
- Methoden sind immer öffentlich und abstrakt (ohne Schlüsselworte public, abstract)
- Methoden können nicht static sein
- deklarierte Attribute sind implizit static, final und public und müssen Initialisiert werden (→ Konstanten)
- Interfaces können auch an andere Interfaces vererbt werden
- durch Interfaces wird Mehrfachvererbung möglich
- Subklassen können **nur eine Basisklasse erweitern, aber mehrere Interfaces implementieren!**

Beispiel für generisches Interface:

Vergleich zweier Objekte:

```
interface Comparable<T> {
    int compareTo (T obj2);
}
```

Rückgabewert: 0 bei Gleichheit  
< 0 falls Objekt selbst kleiner als obj2  
> 0 falls Objekt selbst größer als obj2


```
class Kl implements Comparable<Kl>{
    private int wert;
    ...
    @Override
    public int compareTo(Kl obj2) {
        ...
    }
}
```

Verwendung z.B. Sortieren:

```
import java.util.Collections;
...
ArrayList<Kl> klliste;
klliste = new ArrayList<Kl>();
...
Collections.sort(klliste);
...
```

## Klassen für Zeichenketten

### String

 **Inhalt kann nicht mehr verändert werden!**  
(Methoden replace, toLowerCase ... erzeugen neue Strings)

```
String satz = "Viel Spaß!";
```

Zugriff auf ein Zeichen / Länge des Strings

```
char charAt(int index) / int length()
```

Liefert 1. Position, von ch (Zeichen) / st (String) im String

```
int indexOf(int ch) bzw. (String st)
```

true, falls die Zeichenkette st im String enthalten ist

```
boolean contains(String st)
```

Vergleich zweier Strings; true, falls beide Strings gleich

```
boolean equals (Object anObject)
```

Liefert Teilstring (ohne das Zeichen an Position end)

```
String substring(int begin, int end)
```

Ersetzen eines Zeichens

```
String replace(char old, char new)
```

Umwandeln in Kleinbuchstaben / Großbuchstaben

```
String toLowerCase() / toUpperCase()
```

## StringBuilder und StringBuffer

- veränderbare Zeichenketten

```
import java.lang.StringBuilder;
...
StringBuilder satz;
satz = new StringBuilder("Viel Spaß!");
```