

Objektorientiertes Programmieren

(Java-Einführung)



SoSe 2018

Oliver Wiese

Die Welt im Computer abbilden



Häuser in Python

```
wohnungen = {}  
hoehe = {}  
mietparteien = {}  
hausnummer = []
```

Wo sind die Mietparteien?

```
def neuesHaus(myHausnummer, myFarbe, myEckhaus, myHoehe, myFenster, myWohnungen):  
    fenster[myHausnummer] = myFenster  
    farbe[myHausnummer] = myFarbe  
    eckhaus[myHausnummer] = myEckhaus  
    hoehe[myHausnummer] = myHoehe  
    wohnungen[myHausnummer] = myWohnungen  
    mietparteien[myHausnummer] = 0
```

Wörterbuchchaos!

```
def einziehen(myHausnummer):  
    if mietParteien[myHausnummer] < wohnungen[myHausnummer]:  
        print("Willkommen!")  
    else:  
        print("Computer sagt nein.")
```

```
def ausziehen(myHausnummer):  
    if mietParteien[myHausnummer] > 0:  
        print("Auf Wiedersehen!")  
        mietParteien[myHausnummer] -= 1  
    else:  
        print("Computer sagt nein.")
```

```
def renovieren(myHausnummer, neueHoehe, neueFenster, neueWohnungen, neueFarbe):  
    print("Das dauert etwas.")
```

Können gleichzeitig Häuser renoviert werden?

Warum Objektorientierte Programmierung?

Hauptproblem bei prozeduraler Programmierung

Trennung zwischen Prozeduren (Funktionen) und Daten

- * welche Daten sollen lokal und welche global existieren?
- * Probleme, einen geeigneten Zugriffsschutz auf die globalen Daten zu finden
- * Probleme der realen Welt sind schwer *prozedural* zu simulieren
- * ungeeignet für große Softwaresysteme
- * schlechte Wiederverwendbarkeit der Software

Erste Objektorientierte Ideen

Ole-Johan Dahl und **Kristen Nygaard**

- 1965
- **Simula-I** Programmiersprache
 - erste objektorientierte Programmiersprache
 - speziell für diskrete ereignisorientierte Simulation.
- **Simula-67**
 - Neue Konzepte:
 - Objekte, Klassen, Subklassen, Nebenläufigkeit und *Garbage Collection*.



Erste Objektorientierte Ideen

```
Class Rectangle (Width, Height); Real Width, Height;  
Begin  
  Real Area, Perimeter;  
  
  Procedure Update;  
  Begin  
    Area := Width * Height;  
    Perimeter := 2*(Width + Height)  
  End of Update;  
  
  Boolean Procedure IsSquare;  
    IsSquare := Width=Height;  
  
  Update  
  OutText("Rectangle created: ");  
  OutFix(Width,2,6); OutFix(Height,2,6); OutImage  
End of Rectangle;
```

Simula

Erste Objektorientierte Ideen

Ivan Sutherland's **Sketchpad**, 1963



Erstes objektbasiertes
Malprogramm

Master- und Instanz-
Zeichnungen

draw a car

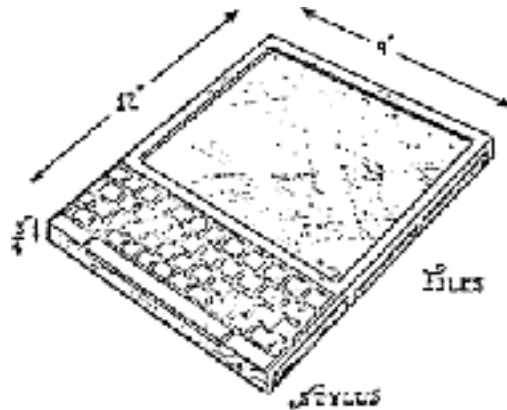
make two instances

Erste Objektorientierte Ideen

Alan Kay



- 1968. Xerox
- erste objektorientierte Konzepte für die Gestaltung von Benutzeroberflächen
- **Dynabook**-Konzept



- **Smalltalk**-Programmiersprache

„The best way to predict the future is to invent it.“

OO-Konzepte können mit anderen Programmiersprachen gelernt werden.

- * Eigentlich ist egal welche Sprache.
- * Die wichtigsten Konzepte und Algorithmen sind sprachunabhängig.
- * Java ist keine perfekte Sprache und auch nicht die beste.
- * Es kommen mit Sicherheit bessere Programmiersprachen.

Smalltalk-Programmiersprache

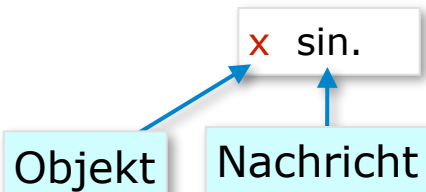
- * erste rein objektorientierte Programmiersprache
- * für das Betriebssystem des **Dynabook**
- * sehr einfach und elegant

"Alles ist ein Objekt"

- * Dynamisches Typsystem
- * Objekte bekommen Nachrichten und werden damit aktiviert.
- * die wesentliche objektorientierte Konzepte sind da
- * eigenwillige Syntax
- * automatische Speicherbereinigung (Garbage Collection)
- * Virtuelle Maschine

Smalltalk-Programmiersprache

'Hello World' out.



Zuweisung

```
c ← Counter new.  
c := Counter new.
```

return

a < b

```
ifTrue: [^a]  
ifFalse: [^b]
```

Anweisungsblock

Schleife

```
10 timesRepeat: [  
    Transcript show:'hi'.  
    Transcript cr.  
].
```

for-Schleife

```
1 to: 10 do:[ :k |  
    Transcript show:k.  
    Transcript show:' '.  
    Transcript show:k sqt.  
    Transcript cr.  
].
```

while-Schleife

```
|i|  
i:=5.  
[i > 0] whileTrue:[  
    Transcript show: ((i*2) asString) ; cr.  
    i:=i-1.  
].
```

Objektorientierte Programmiersprachen

1962 - **Simula I**

1967 - **Simula 67**

1971 - **Smalltalk -71**

1980 - **Smalltalk-80**

1983 - **Objective-C**

1983 - **C++**

1985 - **Eiffel**

1991 - **Python**

1991 - **Java**

1993 - **Ruby**

1994 - **CLOS**

1995 - **Delphi (Object Pascal)**

„I invented the term Object-Oriented, and I can tell you I did not have C++ in mind.“

Alan Kay

Objektorientierte Programmiersprachen

1995 - Ada (Objektorientiert)

2001 - **C#**

2002 - COBOL (Objektorientiert)

2003 - Fortran (Objektorientiert)

2003 - **Scala**

2004 - PHP5 (Objektorientiert)

2005 - F#

2007 - D

2008 - MATLAB

2011 - Ceylon

2011 - Dart

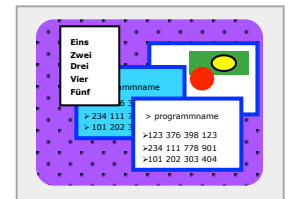
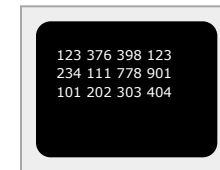
2011 - Opa

2012 - Perl

...

Warum objektorientiertes Programmieren?

- Simulationsprobleme
 - in der Welt läuft alles parallel
- **Wiederverwendbarkeit** von Software
 - saubere Modularisierung der Software
 - mit klaren Schnittstellen
- Graphische Benutzeroberflächen
 - nicht sequentielle, interaktive Steuerung von Anwendungsprogrammen
- Programmierung verteilter Anwendungen
 - Parallelität und Kommunikation auf natürliche Weise



Was ist ein Objekt?

Ein Objekt vereint Daten und Prozeduren (Funktionen oder Methoden), die auf diesen Daten operieren, in einem Wert (Verbundvariable).

Was ist ein OO-Programm?

Ein **OO**-Programm ist ein System kooperierender Objekte.

Was ist ein Objekt?

Objekte sind verwandt mit:

C/C++ **struct**-Datentyp

```
struct Student {  
    int matrikelnummer;  
    int alter;  
    int semester;  
};
```

Pascal Records

```
TYPE datum = RECORD  
    tag, monat, jahr : INTEGER;  
END;
```

Mit dem **wesentlichen** Unterschied, dass Objekte immer einen zusätzlichen Zeiger auf die Tabelle der Klasse haben, nach deren Vorgabe diesen erzeugt wurden.

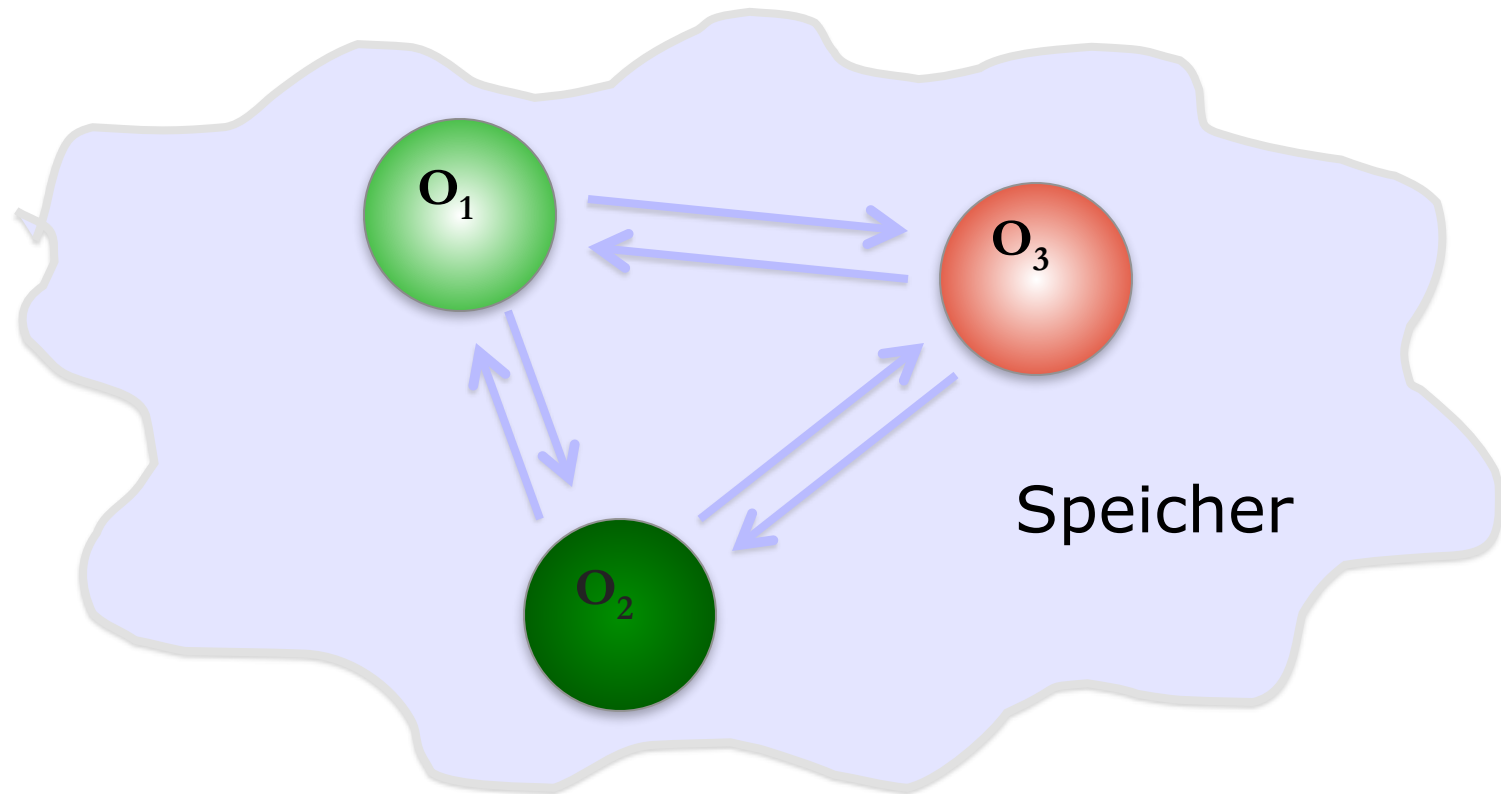
V-Table des Objekts
... Methoden ...

vptr

Objekt

OO-Programmausführung

Die OOP betrachtet eine Programmausführung als ein System kooperierender Objekte.



Objektorientiertes Programmieren

Wiederverwendbarkeit

"Any fool can write code that a computer can understand.
Good Programmers write code that humans can understand."

Martin Fowler in >>Refactoring<<

Objektorientiertes Programmieren

Vorgänge der realen Welt

- inhärent paralleles Ausführungsmodell

Trennung von Auftragserteilung und Auftragsdurchführung

- klar definierte Schnittstellen

Klassifikation und Vererbung

- Anpassbarkeit, Klassifikation und Spezialisierung von Programmteilen

Konzepte Objektorientierter Programmierung

- * Objekte
- * Klassen
- * Nachrichten
- * Kapselung
- * Vererbung
- * Polymorphismus

statisch

Was ist eine Klasse?

Eine Klasse ist ein Bauplan, um Objekte einer bestimmten Sorte zu erzeugen.

Ohne Klassen gibt es keine Objekte in vielen Programmiersprachen!

Klassen besitzen den Vorzug der Wiederverwendbarkeit.

Was ist eine Klasse ?

statisch

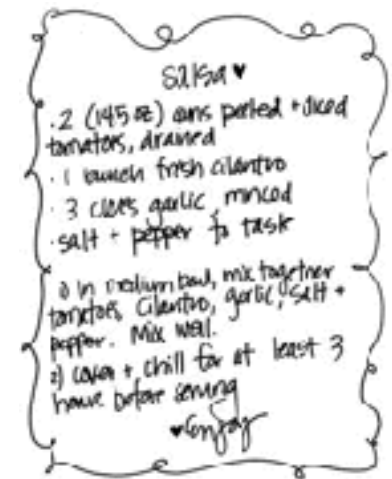
Stempel



Bauplan



Rezept



dynamisch

Was ist ein Objekt ?

Ein Objekt ist ein Softwarebündel aus **Variablen** und mit diesen Variablen zusammenhängenden **Methoden**.

Ein Objekt ist eine konkrete Ausprägung bzw. eine Instanz einer Klasse.

Jedes Objekt „weiß“, zu welcher Klasse es gehört.

Was sind Objekte?

dynamisch

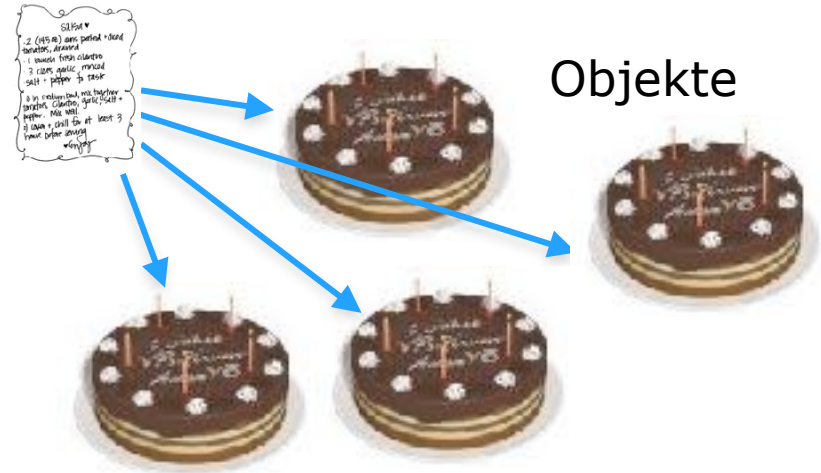
Klasse

Objekte



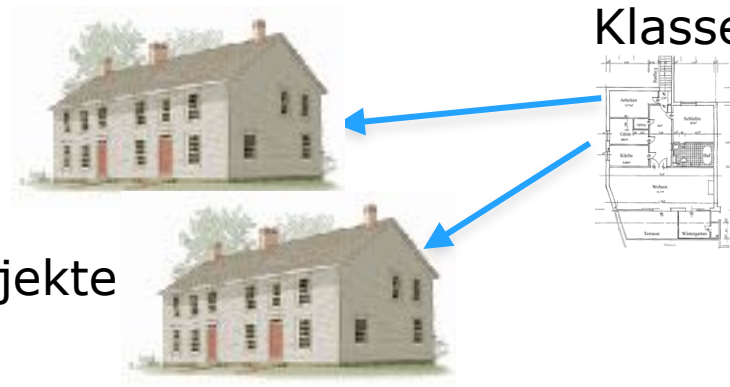
Klasse

Objekte

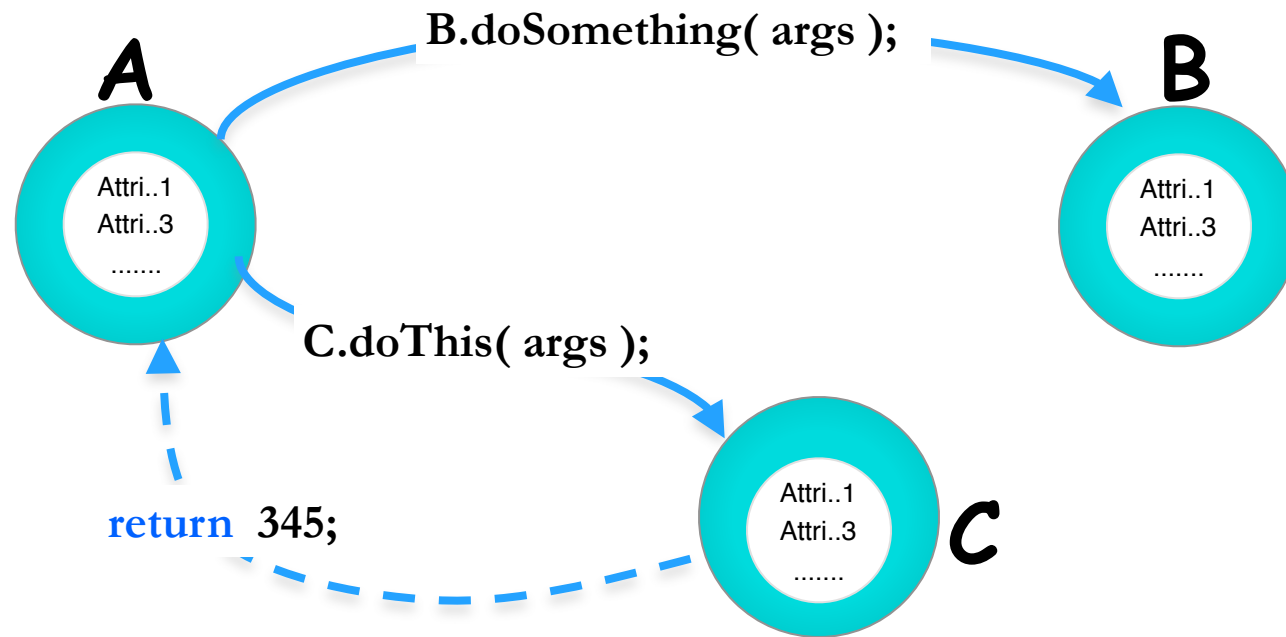


Klasse

Objekte



Was sind Nachrichten ?



Methodenaufrufe

- Empfänger
- Name der auszuführenden Methode
- Parameter

Nachrichten

`personX.calculateSomething(a, b)`


Empfänger


Befehl



Argumente



Bevor ich eine Nachricht zu einem Objekt schicke, muss dieses Empfängerobjekt im Speicher existieren.

OOP: Das Grundmodell

- Objekte haben einen **lokalen Zustand**.
- Objekte empfangen und bearbeiten **Nachrichten**.

Ein Objekt kann seinen Zustand ändern,
Nachrichten an andere Objekte verschicken,
neue Objekte erzeugen oder existierende Objekte löschen.
- Objekte sind grundsätzlich **selbständige Ausführungseinheiten**, die unabhängig voneinander und **parallel** arbeiten können.

Klasse-Definition

Attribute:

- *Eigenschaft₁*
- *Eigenschaft₂*
- • • •

Verhalten:

- *Methode₁*
- *Methode₂*
- *Methode₃*
- • • •

Beispiel: Katze-Klasse

Attribute :

- Name
- Besitzer
- Farbe
- hungrig
- • • • •

Verhalten:

- isst
- läuft
- kratzt
- schläft
- • •

Beispiel: Katze-Klasse

Attribute :

- Name
- Besitzer
- Farbe
- hungrig

.

Verhalten:

- isst
- läuft
- kratzt
- schläft

. . .

Katze-Objekt

Attribute :

- Name = TigerJim
- Besitzer = Vic
- Farbe = schwarz
- hungrig = wahr

.

Verhalten:

- isst
- läuft
- kratzt
- schläft

. . .

Was ist Java?

1991 Patrick Naughton und James Gosling

Programmierungsumgebung **Oak** bei Sun Microsystems

Das Ziel war, Anwendungen für elektronische Geräte und das interaktive Fernsehen leicht zu programmieren.

1992

Green Projekt

Oak Programmierungsumgebung

keine Erfolg!



Was ist Java?

- 1996 veröffentlichte *Sun Microsystems* die erste offizielle Entwicklungsumgebung für Java.
- 2007 Free and Open Source Software. GNU General Public License (GPL)
Sun Microsystems → *Oracle*

Beispielloser Erfolg:

Noch nie hat eine neue Programmiersprache in so kurzer Zeit so starke Verbreitung gefunden.

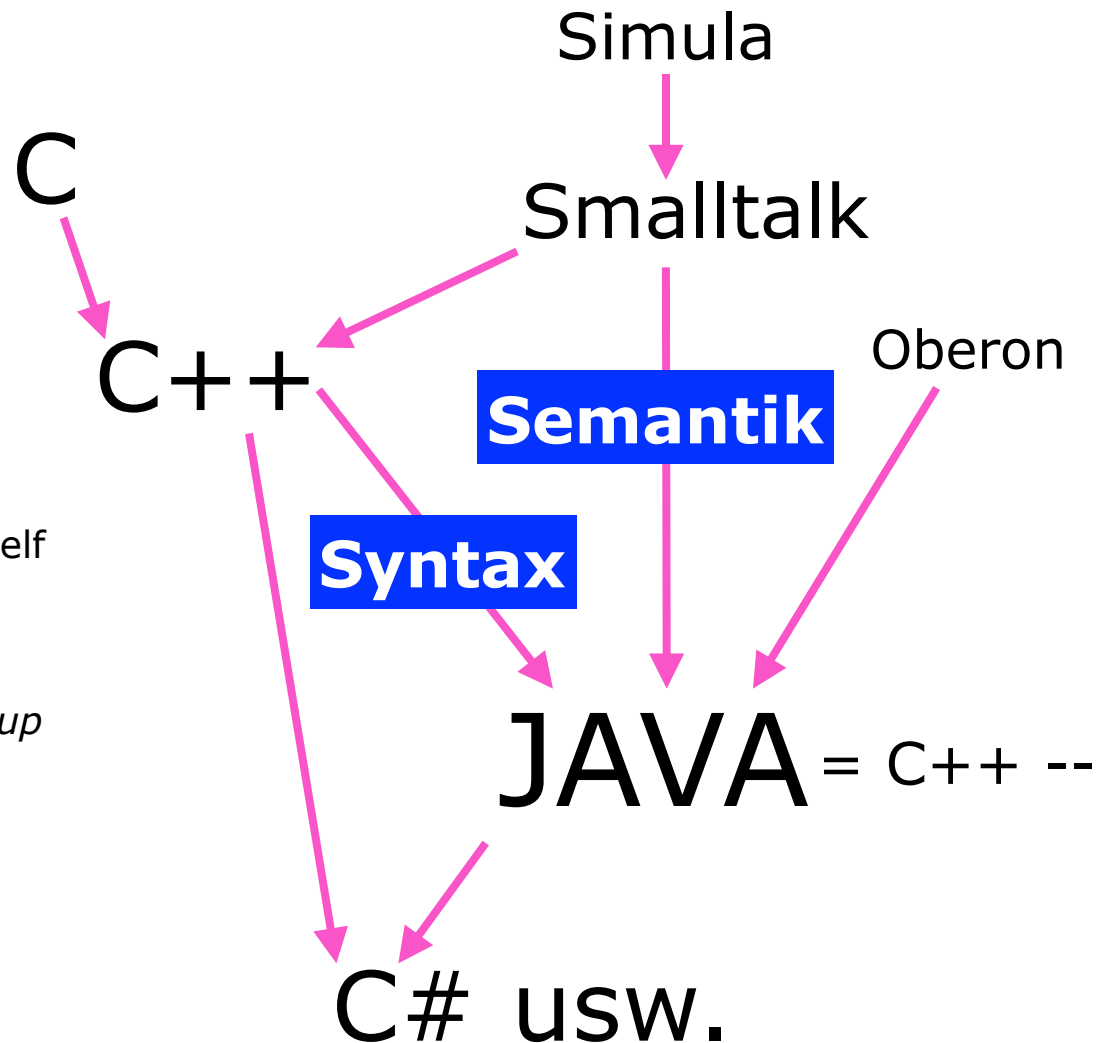
Java ist **22** Jahre alt

Es gibt **930 Millionen** *downloads* der *Java Runtime Environment* pro Jahr
3 Billionen Mobile-Geräte haben eine JVM Java Virtuelle Maschine

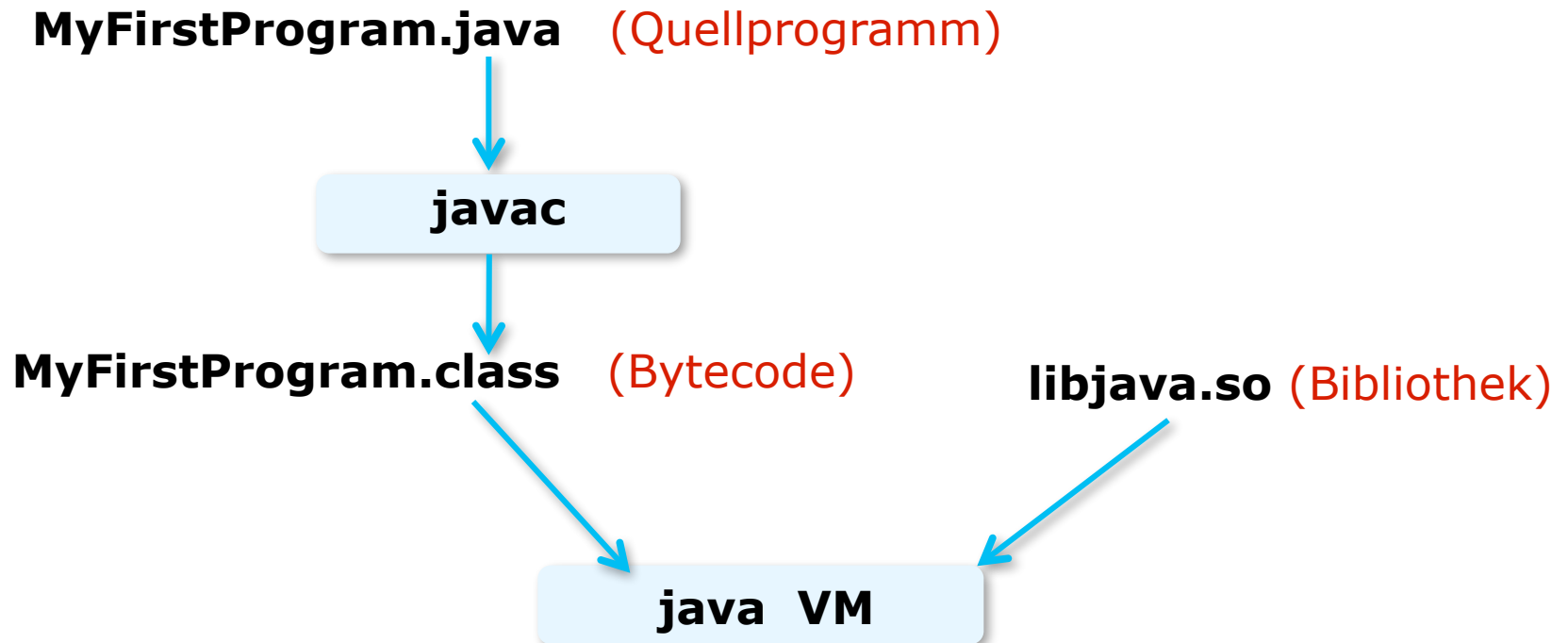
Stammbaum von Java

"In C++ it's harder to shoot yourself in the foot, but when you do, you blow off your whole leg."

Bjarne Stroustrup



Übersetzen/Ausführen



Übersetzen:	<code>javac MyFirstProgram.java</code>
Ausführen:	<code>java MyFirstProgram</code>

Java ist plattformunabhängig

Quellprogramm

```
public class ...
  public read (a)[....
    b = readNum();
    if (a<b) then
      a = a*a;
    else
      a = a+b;
    ...
```

Java-Compiler
javac

Bytecode

```
iLOAD  #1 C
iLOAD  #2 B
iMULT  #1 #2 #3
iLOAD  #4 A
iADD   #3 #4 #1
iSTORE #1 C
iLOAD  #4 A
iADD   #3 #4 #1
```

Interpreter

JVM

JVM

JVM

JVM

Java-Interpreter (JVM)
sind überall.



Java-Programme

- Java-Programme bestehen aus einer oder mehreren *Klassen*
- Eine Klasse ist in der Regel in einer eigenen, gleichnamigen Datei mit der Endung **.java** definiert, z. B.

MyFirstProgram.java

- Die Programmausführung beginnt immer mit der Methode **main** einer der Klassen.

Java-Programm

Beispiel:

```
/* Ein einfaches aber vollständiges Java-Programm */  
public class MyFirstProgram {  
  
    public static void main ( String[] args ) {  
        System.out.println( "Es läuft ! " );  
    }  
  
} // end of class MyFirstProgram
```

Jedes Java-Programm muss mindestens eine Methode namens **main** haben. Hier fängt die Programmausführung an.

Kommentare in Java

Zeilenend-Kommentare

`//` von hier aus bis zum Ende der Zeile wird dieser Text ignoriert

Block-Kommentare

`/*` alle diese Zeilen hier werden von
dem **javac** völlig ignoriert
`*/`

Javadoc-Kommentare

`/**` dieser Text wird von dem javadoc-Programm verwendet,
um automatische Dokumentation in html-Format zu
erzeugen
`*/`

Einfache Anweisungen in Java

Die einfachsten Befehle in jeder Programmiersprache sind die Zuweisungen.

Zuweisung in Java

Variablenname

=

Ausdruck

;

Beispiele:

```
b = 3 * b ;
```

```
c = a / b ;
```

Semikolon

Variablendeklarationen

Im Unterschied zu Python müssen in Java alle Variablen vor der ersten Anwendung deklariert werden.

Modifizierer	Typ	Name	Wert
	int	breite ;	
	int	hoehe = 10 ;	
public	float	radius = 0.0 ;	
private	float	radius = 0.0 ;	

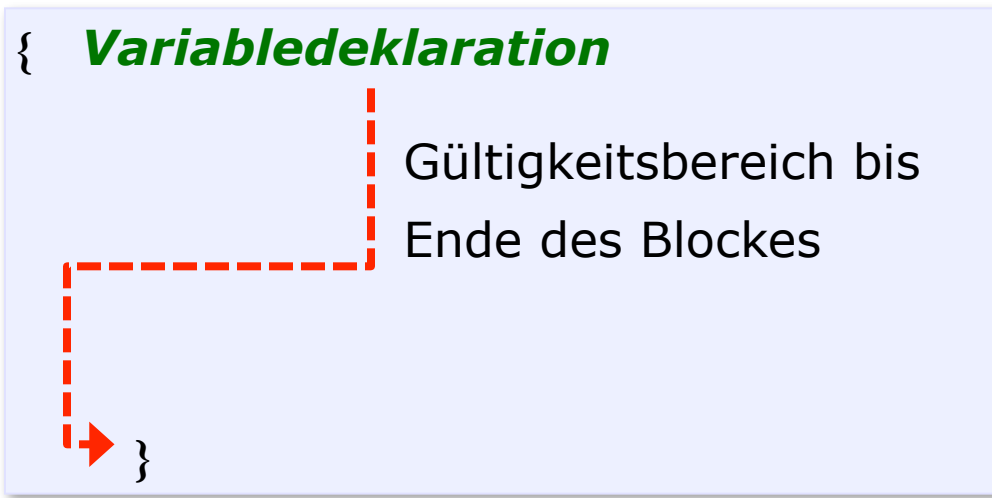


Der **Modifizierer** bestimmt die Zugriffsrechte, die andere Objekte auf eine Variable (Attribut) haben.

Variablen in Java

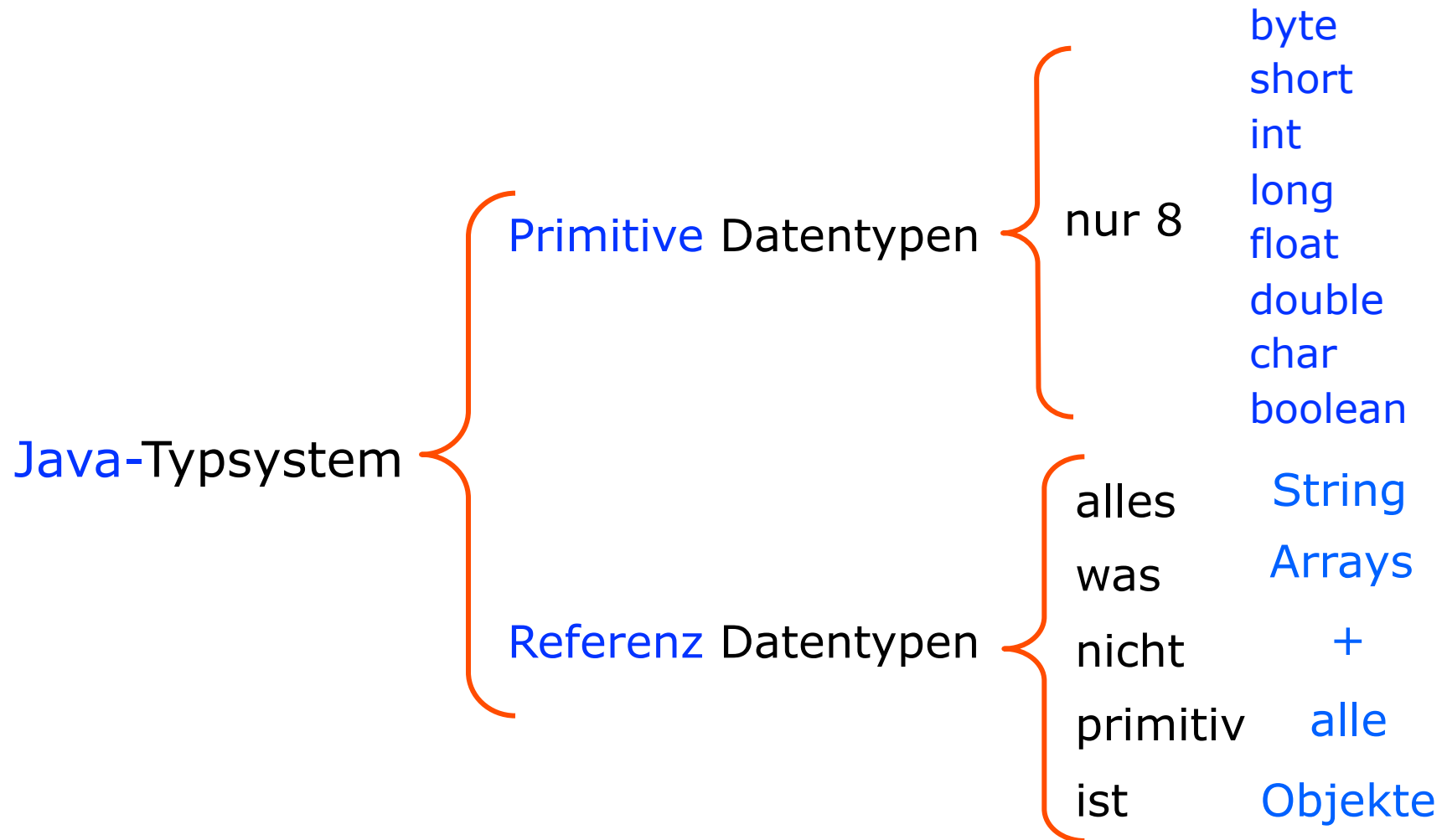
Variablen können überall deklariert werden, aber nicht außerhalb von Klassendefinitionen.

Ihr Gültigkeitsbereich erstreckt sich von der Stelle ihrer Deklaration bis zum Ende des *Blocks*, in dem sie deklariert wurden.



Java-Typsystem

- Java ist **streng typisiert**, d.h. jeder Ausdruck hat einen wohldefinierten Typ.
- Die Einhaltung aller durch das Typsystem definierten Regeln wird
 - zuerst **statisch** vom **Übersetzer** überprüft.
 - und dann **dynamisch** vom **Interpreter**.
- Neue Objekttypen werden durch Klassen definiert. Variablen solcher Typen sind **Referenzen**:



Primitive Datentypen in Java

Java hat **8** elementare Datentypen und legt für jeden primitiven Datentyp eine **feste Speichergröße** und damit einen **festen Zahlenbereich** fest.

Typ	Bits	Zahlenbereich	Standardwert
byte	8	-128 . . . 127	0
short	16	-32.768 . . 32.767	0
int	32	-2^{31} . . . $2^{31}-1$	0
long	64	-2^{63} . . . $2^{63}-1$	0
float	32	ca. $3.402 \cdot 10^{38}$. . $1.402 \cdot 10^{-45}$	0.0
double	64	ca. $1.798 \cdot 10^{308}$. . $4.94 \cdot 10^{-324}$	0.0
boolean	1	true, false	false
char	16	Unicode	\u0000

Strings

Strings sind in Java kein Basistyp, sondern eine Bibliotheksklasse

`java.lang.String`

`String s = "hallo";` äquivalent zu `String s = new String ("hallo");`

Methoden:

<code>s.toUpperCase()</code>➤	"HALLO"
<code>s.charAt(4)</code>➤	'O'
<code>s.length()</code>➤	5
<code>s.equals(s)</code>➤	true
<code>s.replace('A', 'Ä')</code>➤	"HÄLLO"
<code>s.substring(2, 4)</code>➤	"LL"

Strings

Strings in Java sind immer **konstant**, d.h. unveränderlich.

Folgende Zuweisungen erzeugen jeweils neue String-Objekte und sind äquivalent.

```
String s = "hallo";
```

s.length() Wert → 5

```
s = s + "Welt!";
```

||

```
s = new String(s + "Welt!");
```

Arithmetische Operatoren

unär

Operator	Zeichen	Rtg.	Beispiel	Priorität
negatives Vorzeichen	-	←	-x	13

binär

Operator	Zeichen	Rtg.	Beispiel	Priorität
Multiplikation	*	→	a * b	12
Division	/	→	3 / 5	12
Rest	%	→	6 % 5	12
Addition	+	→	10 + 3	11
Subtraktion	-	→	10 - 3	11

Ausdrücke	Wert	Type
7 / 2 →	3	int
1 / 2 →	0	int
1 / 2.0 →	0.5	double
4 % 2 →	0	int
7 % 2 →	1	int
1 % 2 →	1	int

Einfache Befehle in Java

...

// Deklaration von Variablen

int a, b, c ;

// Zuweisungen

a = 7 % 2 ;

b = a * a ;

c = a / b + 1 ;

...

Inkrement- und Dekrement-Operatoren

Operator	Benutzung	Beschreibung	Priorität
++	++ op	Inkrementiert op um 1	13
++	op ++	Inkrementiert op um 1	13
--	-- op	dekrememtiert op um 1	13
--	op --	dekrememtiert op um 1	13

Beispiele:

```
int i = 10, j = 0;
```

```
j = ++i;
```

```
j = i++;
```

j: 11

i: 11

j: 11

i: 12

```
int a = 5;
```

```
int b = 4;
```

```
int c = 0;
```

```
int d = 0;
```

```
c = --a + b++;
```

```
c = c + ++a;
```

```
d = c++ - ++a + b--;
```

a: 4

b: 5

c: 8

a: 5

b: 5

c: 13

Was ist der Inhalt der Variablen

a?

b?

c?

und

d?

6

4

14

12


Vergleichsoperatoren

binär

Kleiner	<	→	op1 < op2	9
Größer	>	→	op1 > op2	9
Kleiner oder gleich	<=	→	op1 <= op2	9
Größer oder gleich	>=	→	op1 >= op2	9
Gleichheit	==	→	op1 == op2	8
Ungleichheit	!=	→	op1 != op2	8

Logische Operatoren

unär

Operator	Zeichen	Rtg.	Beispiel	Priorität
logische Negation	!		!a	13

binär

UND	&&		a && b	4
ODER	 		a b	3

ternär

bedingter Ausdruck	B ? A₁ : A₂		!true ? 4 : 9	2
--------------------	--	---	----------------------	---

Typ-Operatoren

unär

Operator	Zeichen	Rtg.	Beispiel	Priorität
"cast"-Operator	<i>(typ)</i>		(int) a	13

binär

Typvergleich für Objekte	instanceof		a instanceof Button	9
--------------------------	-------------------	--	----------------------------	---

```
Button knopf = new Button();
```

```
...
```

```
if ( knopf instanceof Button )
```

```
    knopf.setBackground(Color.yellow);
```

```
...
```

Java-Operatoren

Bezeichnung	Operator	Priorität
Komponentenzugriff bei Klassen	.	15
Komponentenzugriff bei Feldern	[]	15
Methodenaufruf	()	15
Unäre Operatoren	++,--,+,-,~,!	14
Explizite Typkonvertierung	()	13
Multiplikative Operatoren	*, /, %	12
Additive Operatoren	+, -	11
Schiebeoperatoren	<<, >>, >>>	10
Vergleichsoperatoren	<, >, <=, >=	9
Vergleichsoperatoren (Gleichheit, Ungleichheit)	==, !=	8
bitweise UND	&	7
bitweise exklusives ODER	^	6
bitweise inklusives ODER		5
logisches UND	&&	4
logisches ODER		3
Bedingungsoperator	? :	2
Zuweisungsoperatoren	=, *=, -=, usw.	1

Arrays

In Java kann man eine Folge gleichartiger Objekte unter einem Namen zusammenfassen. (Reihungen)

Eindimensionale Felder (Arrays)


Beispiel:

```
int[] zahlen;  
zahlen = new int[12];
```



```
int[] zahlen = new int[12];
```

```
int[] primes = { 2, 3, 5, 7, 11, 13, 17 };
```



Arrays können beliebige Datentypen enthalten, auch weitere Arrays.

Arrays

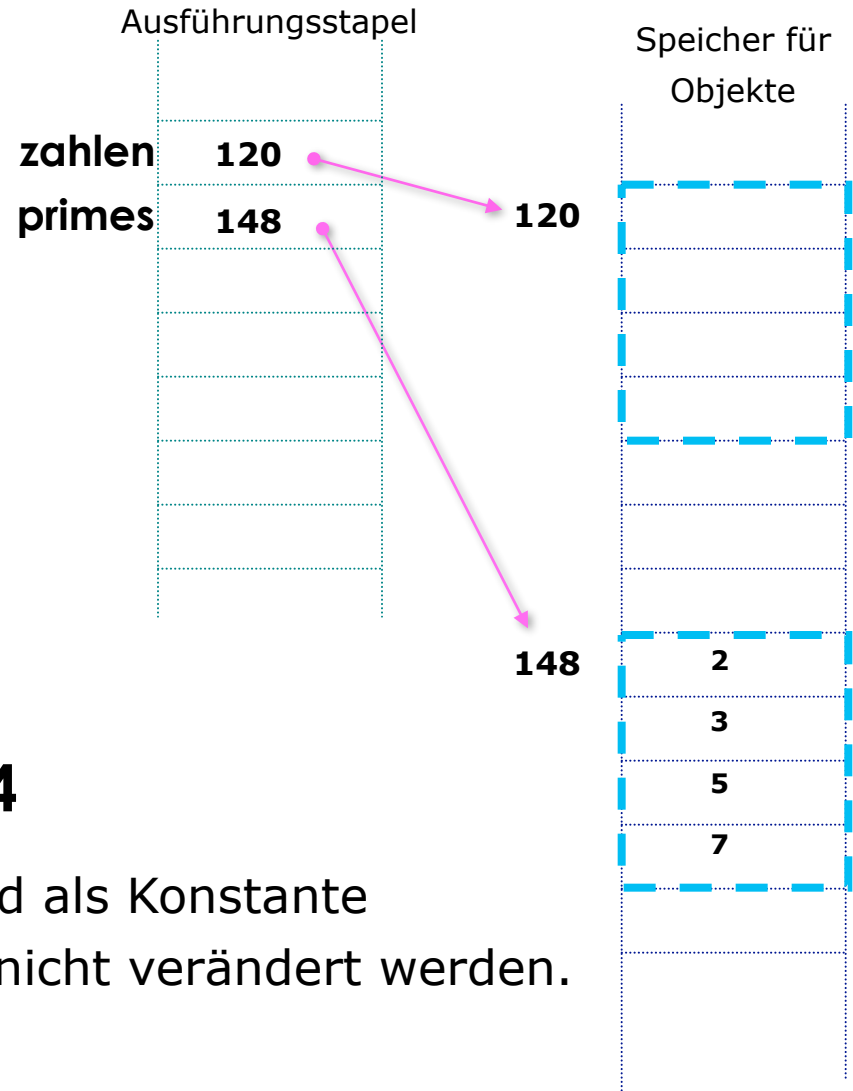
```
int[] zahlen;
```

```
zahlen = new int[4];
```

```
int[] primes = { 2, 3, 5, 7 };
```

primes.length → 4

Das **length**-Attribut des Arrays wird als Konstante generiert und kann gelesen, aber nicht verändert werden.



Arrays

Mehrdimensionales Array: Array von Arrays.

```
double[][] matrix = new double[3][3];
```

2.0	3.0	1.0
6.0	0.0	7.0
2.0	8.0	4.0

matrix[1][2] → 7.0

↑
Zeilen

↑
Spalten

Arrays

```
java Main eins zwei 1000
```

↓ ↓ ↓
`args[0]` `args[1]` `args[2]`

```
public class Main {  
    public static void main( String[] args ) {  
        if ( args.length > 2 ) {  
            System.out.print( args[0] );  
            System.out.print( args[1] );  
            System.out.print( args[2] );  
        }  
    }  
} // end of class Main
```

Ausgabe:

```
eins zwei 1000
```

Anweisungen zur Ablaufsteuerung

Einzelne Anweisungen (*statements*) werden mit einem Semikolon abgeschlossen:

```
a = b * c;
```

Anweisungen werden durch geschweifte Klammern zu *Blöcken* zusammengefasst:

```
{  
    a = b * r;  
    x = a + z/a;  
}
```

Die Abarbeitungsreihenfolge der Anweisungen verläuft normalerweise von oben nach unten und von links nach rechts.

Anweisungen zur Ablaufsteuerung

if

switch

while

do-while

for

Anweisungen zur Ablaufsteuerung

```
if ( Ausdruck )  
  { Anweisungen }  
else  
  { Anweisungen }
```

```
switch ( Ausdruck ) {  
  case Konstante1: Anweisungen break;  
  case Konstante2: Anweisungen break;  
  . . . . . USW.  
  default : Anweisungen  
}
```

```
while ( Bedingung )  
{  
  Anweisungen  
}
```

```
do {  
  Anweisungen  
}  
while ( Bedingung ) ;
```

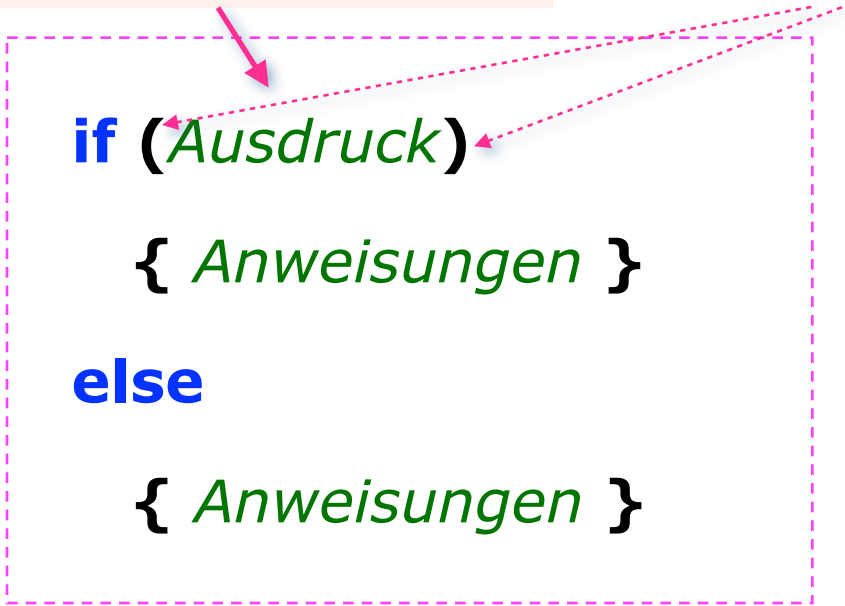
Semikolon

```
for ( Initialisierung ; Bedingung ; Inkrement )  
  { Anweisungen }
```

if-else-Anweisung

Wahrheitswert → **boolean**

Runde Klammern



```
if (Ausdruck)  
    { Anweisungen }  
else  
    { Anweisungen }
```

if-else-Anweisung

```
...  
if( punkte >= 90 )  
    note = 1;  
else if ( punkte >= 80 )  
    note = 2;  
else if ( punkte >= 70 )  
    note = 3;  
else if ( punkte >= 60 )  
    note = 4;  
else note = 5;  
...
```

switch-Anweisung

```
switch (Ausdruck) {  
    case Konstante1: Anweisungen break;  
    case Konstante2: Anweisungen break;  
    . . . . . usw.  
    default : Anweisungen  
}
```

switch-Anweisung

short
int
long
char

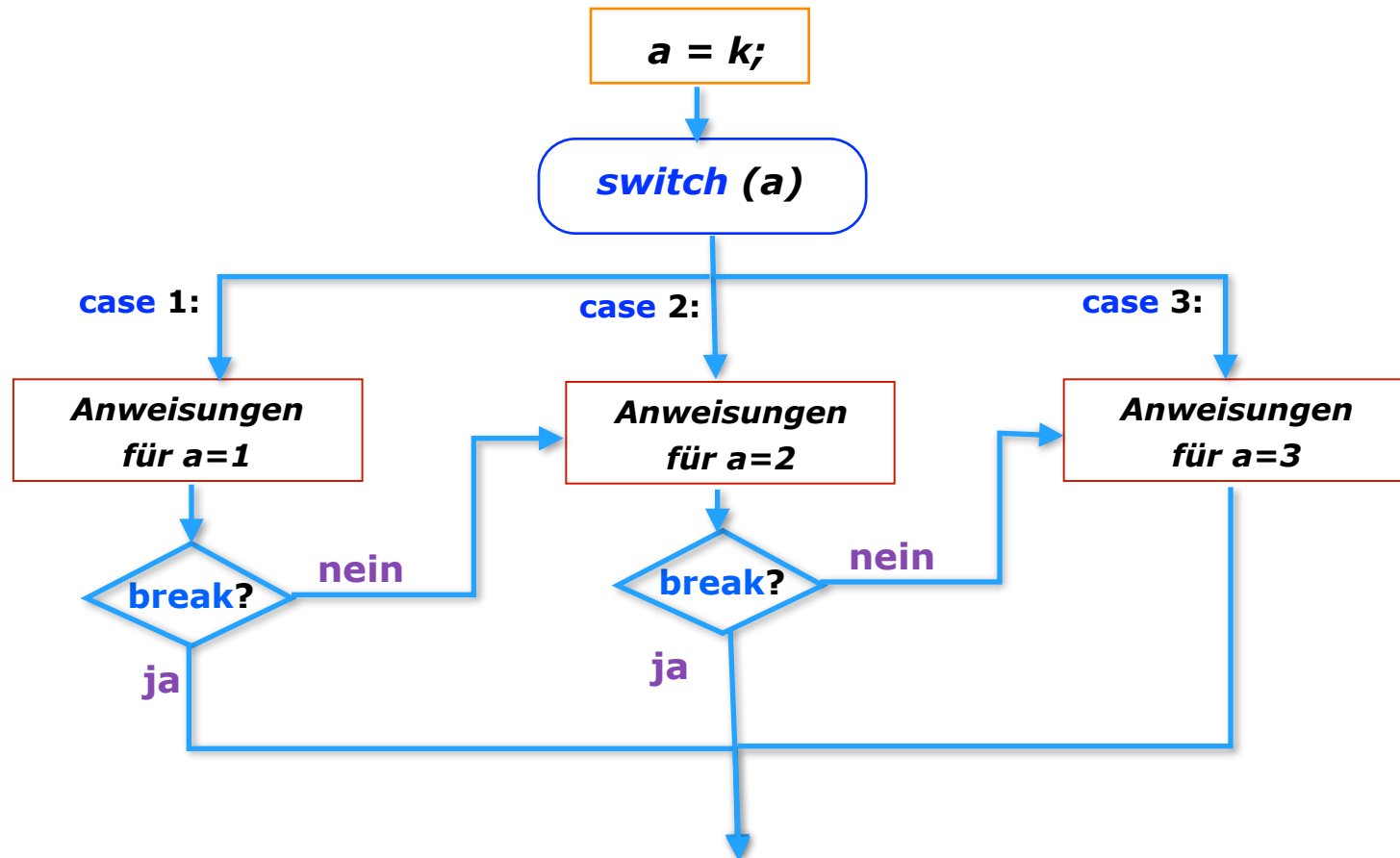
Die switch-Anweisung erlaubt die Verzweigung in Abhängigkeit vom Wert eines *ganzzahligen* Ausdrucks, der mit einer Reihe von Konstanten verglichen wird.

```
switch ( Ausdruck ) {  
    case Konstante1 : {  
        Anweisung1  
        Anweisung2  
        ...  
    }  
    break;  
    case Konstante2 : Anweisungen  
    break;  
    . . . . . USW.  
    default: Anweisungen  
}
```


switch-Anweisung



Kontrollfluss

Beispiel:



switch-Anweisung

```
int zahl; String word;  
...  
switch( zahl ) {  
    case 1: word = "eins"; break;  
    case 2: word = "zwei"; break;  
    case 3: word = "drei"; break;  
    case 4: word = "vier"; break;  
    case 5: word = "fünf"; break;  
    case 6: word = "sechs"; break;  
    case 7: word = "sieben"; break;  
    case 8: word = "acht"; break;  
    case 9: word = "neun"; break;  
    case 0: word = "null"; break;  
    default: word = "error";  
}
```



switch-Anweisung

```
int monat, jahr, tage;  
...  
switch( monat ) {  
    case 1:  
    case 3:  
    case 5:  
    case 7:  
    case 8:  
    case 10:  
    case 12: tage = 31; break;  
    case 4:  
    case 6:  
    case 9:  
    case 11: tage = 30; break;  
    case 2: if ( jahr % 4 == 0 && ( jahr % 100 != 0 || jahr % 400 == 0 ) )  
            tage = 29;  
            else  
                tage = 28; break;  
    default: System.out.println("falsche Monatsangabe");  
}
```

while-Schleife

Wahrheitswert → **boolean**

while (*Bedingung*)

{

Anweisungen

}

Runde Klammern

do-while-Anweisung

```
do {  
    Anweisungen  
}  
while (Bedingung) ;
```

Bei der **do-while**-Anweisung wird zunächst der Schleifen-Rumpf ausgeführt und *anschließend* die Bedingung überprüft.

Die Math-Klasse

```
static double sin (double a)
static double cos (double a)
static double tan (double a)
```

```
static double random ()
static long round (double a)
static double ceil (double a)
static double floor (double a)
```

```
static double log (double a)
static double pow (double a, double b)
static double exp (double a)
static double sqrt (double a)
```

Anwendungsbeispiel:

```
double x,y;
. . . . .
double zufallszahl = Math.random();
. . . . .
x = Math.sin (y);
. . . . .
```

for-Anweisung

```
for ( Initialisierung ; Bedingung ; Inkrement ) {  
    Anweisungen  
}
```

for-Schleifen verwenden wir, wenn die Einschränkungen der Schleife (*Initialisierung, Bedingung und Inkrementierung*) bekannt sind.

1. Die *Initialisierung* wird einmal zu Beginn ausgeführt.
2. Der Schleifenrumpf wird ausgeführt, solange die *Bedingung* erfüllt ist.
3. Nach jedem Durchlauf des Rumpfes wird die Anweisung im *Inkrement* einmal ausgeführt und die *Bedingung* erneut geprüft.

for-Anweisung

```
...  
for (int i=0; i<=100; i++ ) {  
    System.out.println( i*i );  
}  
...
```

Die **Initialisierung** wird einmal zu Beginn ausgeführt.

Vor jedem Durchlauf des Rumpfes wird die **Bedingung** geprüft.

Nach jedem Durchlauf des Rumpfes wird die Anweisung im **Inkrement** einmal ausgeführt.

for-Anweisung

Die **Initialisierung** kann mehrere Initialisierungen von Variablen beinhalten.

Nach jedem Durchlauf können mehrere Variablen verändert werden.

```
...  
for ( i=0, j=20, k=5; i<=10; i++, j--, k+=5 )  
{  
    System.out.println( (i+j)*k );  
}  
...
```

```
...
double sum = 0;
for(int i=1; i<=n; i++)
{
    sum += 1/i*i;
}
double pi = Math.sqrt(sum*6);
...
```

falsch!

pi ⇒ 2.449489742783178

```
...
double sum = 0;
for(int i=1; i<=n; i++)
{
    sum += 1.0/(i*i);
}
double pi = Math.sqrt(sum*6);
...
```

falsch!

pi ⇒ 3.1414971639472147

Infinity

50000*50000 ⇒ -1794867295

65535*65535 ⇒ 0

$$R_n = \sum_{i=1}^n \frac{1}{i^2} = \frac{1}{1^2} + \frac{1}{2^2} + \dots + \frac{1}{n^2} \approx \frac{\pi^2}{6}$$

```
...
double sum = 0;
for(int i=1; i<=n; i++)
{
    sum += 1/(i*i);
}
double pi = Math.sqrt(sum*6);
...
```

falsch!

pi ⇒ 2.449489742783178

ArithmeticException

```
...
double sum = 0;
for(int i=1; i<=n; i++)
{
    sum += 1/(1.0*i*i);
}
double pi = Math.sqrt(sum*6);
```

richtig!

pi ⇒ 3.141583104326456

for-Anweisung

Jeder der drei Ausdrücke in der **for**-Schleife kann auch fehlen; die Semikola müssen aber trotzdem gesetzt werden.

```
for ( ; ; ) {  
    ...  
}
```

||

```
while ( true ) {  
    ...  
}
```

Endlosschleifen!!

Variante der **for**-Schleife

for-Schleifen besuchen oft jede Position eines Arrays oder einer Datenstruktur, und aus diesem Grund haben die Java-Entwickler ab Java 1.5 eine abgekürzte Form der **for**-Schleife eingeführt.

```
for ( Typ Element : Datenstrukturname )  
    { Anweisungen }
```

Jedes Element der Datenstruktur wird der Variablen "Element" zugewiesen und innerhalb der Anweisungsblöcke benutzt.

Variante der **for**-Schleife

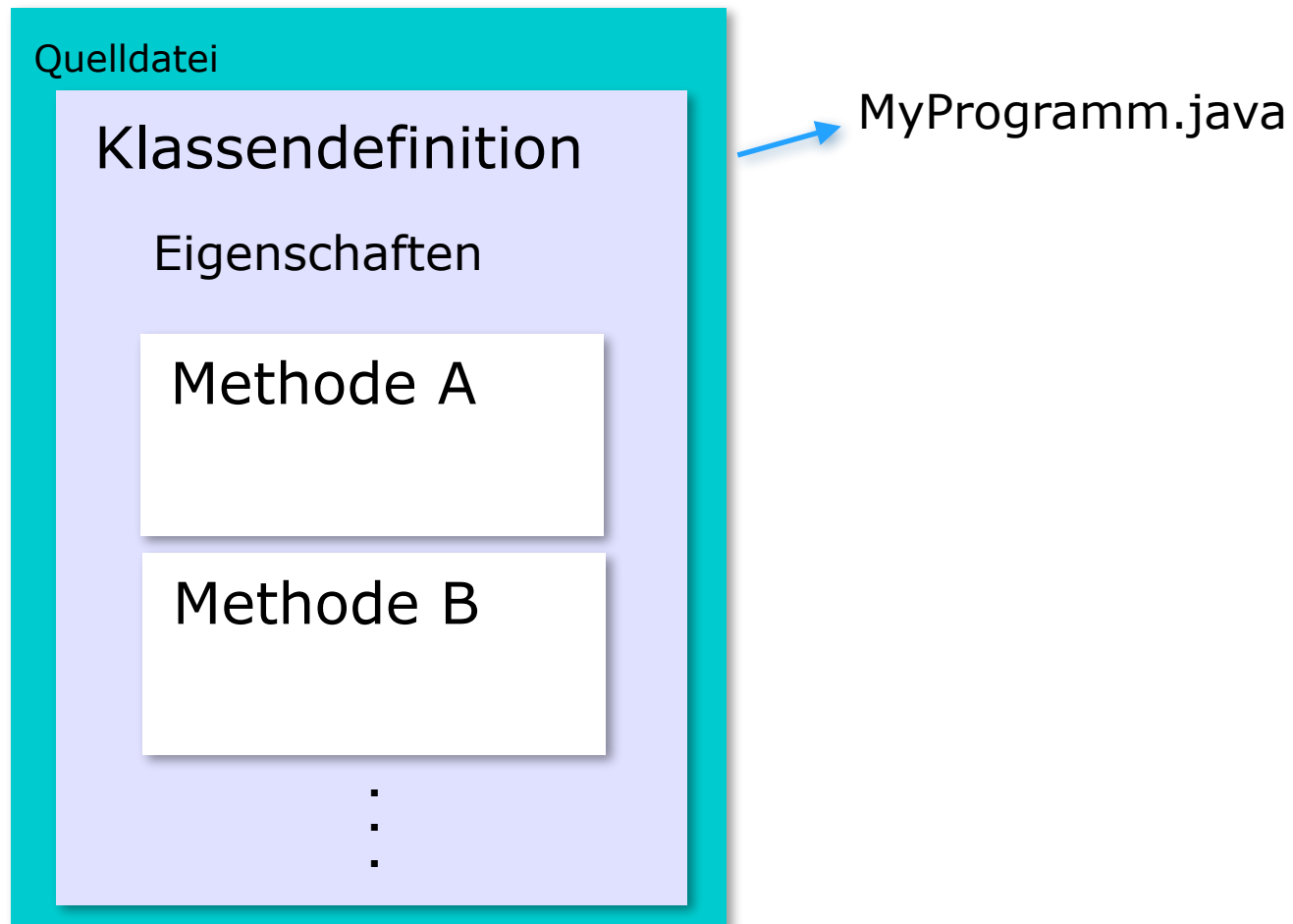
Beispiel:

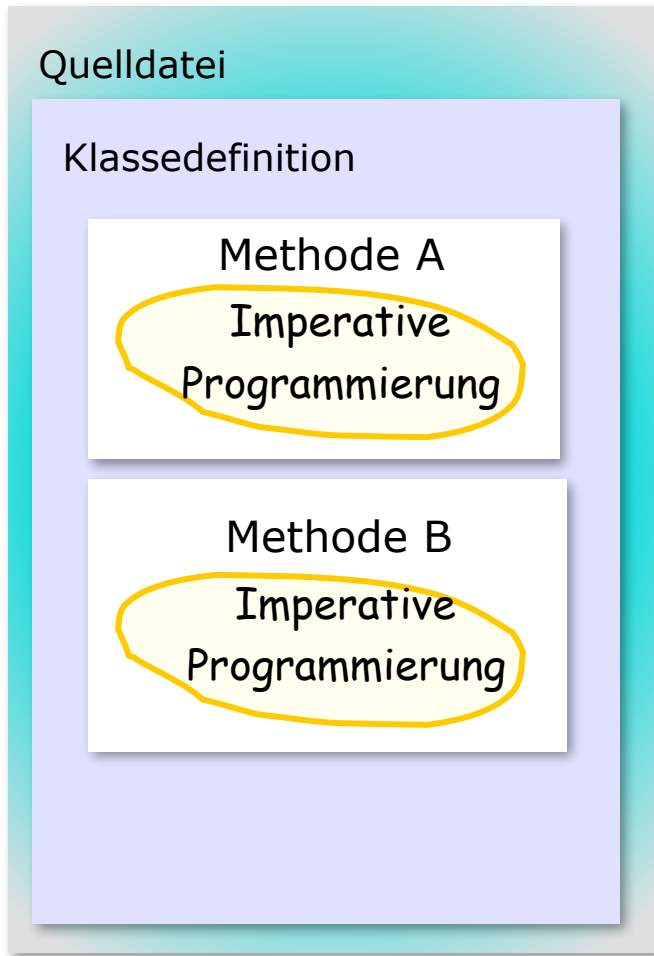
```
...  
double[] nums = { 3.4, 5.6, 1.2, 2.3, 5.6, 7.8 };  
double prod = 1;  
  
for ( double d : nums ) {  
    prod = prod*d;  
}  
...
```

Hello World-Beispiel

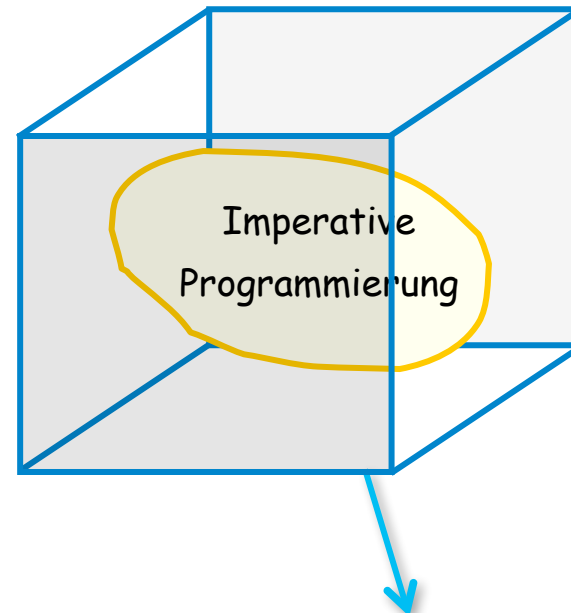
```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        // Prints "Hello, World" to  
        // the terminal window.  
        System.out.println("Hello, World");  
    }  
  
}
```

Grundstruktur eines Java-Programms





Der imperative Bestandteil eines Java-Programms befindet sich innerhalb der Methoden.



objektorientierte Verpackung

Java-Anwendung

Rechtecke.java

```
class Rechteck {  
    // Attribute  
    . . . .  
    // Konstruktoren  
    . . . .  
    // Methoden  
    . . . .  
    . . . .  
    . . . .  
}
```

Kreis.java

```
class Kreis {  
    // Attribute  
    . . . .  
    // Konstruktoren  
    . . . .  
    // Methoden  
    . . . .  
    . . . .  
    . . . .  
    . . . .  
}
```

Blatt.java

```
class Blatt {  
    // Attribute  
    . . . .  
    // Konstruktoren  
    . . . .  
    // Methoden  
    . . . .  
    . . . .  
    .. main ( . . . . ) { .. } ← start  
    . . . .  
}
```

Klassenmethoden

Methoden enthalten den ablauffähigen Programmcode.

Es gibt keine **Methoden** außerhalb von Klassen.

Methoden dürfen nicht geschachtelt werden.

Beispiel:

Modifizierer

Rückgabetyt

Methodenname

Parameter

```
public static int umfang (int width, int height) {  
    return 2*(width + height);  
}
```

Klassenmethoden

$$\pi = 4 \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

```
public class ForStatements {  
  
    public static double pi_leibnitz( int n_max ){  
        double sum = 0;  
        for (int n = 0; n<=n_max; n++ ){  
            if ( n%2 == 0 )  
                sum = sum + (1.0)/(2*n+1);  
            else  
                sum = sum - (1.0)/(2*n+1);  
        }  
        return 4*sum;  
    }  
}
```

Klassenmethoden

Klassenmethoden werden als **static** deklariert und über den Klassennamen aufgerufen:

```
public class MyMath {  
    public static long factorial( int n ) {  
        . . .  
    }  
}
```

Anwendung innerhalb
anderer Klassen



```
...  
long f = MyMath.factorial( 29 );  
...
```

Klassenmethoden

Klassenmethoden haben keinen Zugriff auf Instanzvariablen.

Klassenmethoden dürfen nur andere Klassenvariablen oder lokale Variablen verwenden.

Innerhalb einer als static deklarierten Methode (Klassenmethode) dürfen nur andere statische Methoden aufgerufen werden.

return-Anweisung

Die **return**-Anweisung beendet frühzeitig die Ausführung einer Methode.

In einer Methode kann es mehrere **return**-Anweisungen geben.

Methoden, die als Funktionen definiert sind, d.h. ein Ergebnis liefern, müssen durch eine **return**-Anweisung dieses Ergebnis zurückgeben.

Wenn eine Methode kein Ergebnis zurückgibt, wird das Schlüsselwort **void** als Rückgabebetyp verwendet, und eine **return**-Anweisung ist nicht erforderlich.

return-Anweisung

```
public static int fakultaet ( int zahl ) {  
    int fak = 1;  
    if ( ( zahl == 0 ) || ( zahl == 1 ) )  
        return fak ;  
    else {  
        while ( zahl > 1 ) {  
            fak = fak*zahl;  
            zahl = zahl - 1;  
        }  
        return fak ;  
    }  
} // end of factorial
```

Die **return**-Anweisung beendet den Lauf der Funktion (Methode) und sorgt für die Übergabe des Ergebnisses.

Packages

Java bietet die Möglichkeit, miteinander in Beziehung stehende Klassen zu Paketen (packages) zusammenzufassen. Die Zugehörigkeit zu einem Paket wird über die **package**-Direktive deklariert.

Einzelne oder alle Klassen eines anderen Pakets werden mit der **import**-Direktive "sichtbar" gemacht.

Packages

```
package beispiele;  
  
import java.lang.*; // Standard  
  
public class Person { ... }
```

```
package uebungen;  
import beispiele.*;  
  
...  
    java.lang.String str = "";  
    Person p1;  
  
...
```



Packages und Klassennamen

Wird kein Package bestimmt, so gehört die Klasse automatisch ins globale, unbenannte Package.

Der vollständig qualifizierte Name einer Klasse wird aus dem Klassennamen und allen umschließenden Package-Namen gebildet,

z.B. . . .

java.awt.Button button;

. . .

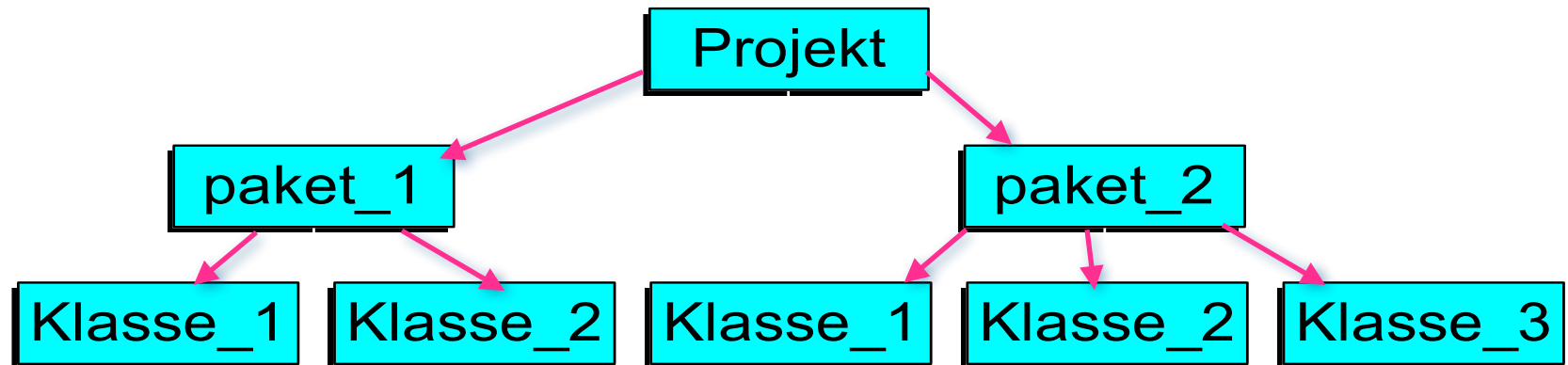
Wenn eine entsprechende **import**-Anweisung vorhanden ist

import java.awt.*;

. . .

schreibe ich nur → Button button;

java-Anwendungen



```
packet packet_1 ;  
public class Klasse_1 {  
    . . . . .  
}
```

Java Konventionen

Motivation:

Java-Code-Konventionen sind aus vielen Gründen sehr wichtig.

Konventionen wurden von erfahrenen Entwicklern konzipiert, um die Kosten der Softwareentwicklung zu senken.

Konventionen verbessern deutlich die Lesbarkeit und das Verständnis von Softwaresystemen.

- a) von den **Entwicklern** selber.
- b) von anderen **Mitgliedern des Teams**.
- c) für zukünftige **Wartung**.

Offizieller Link für Code-Konventionen von Sun-Microsystems.

<http://java.sun.com/docs/codeconv/>

Layout

Einrücken

Rücken Sie den Code innerhalb neuer Blöcke mindestens um 4 Leerzeichen ein.

Zeilenlänge

Vermeiden Sie Zeilen, die länger als **80** Zeichen sind. Gebrochene längere Zeilen machen gedruckte Programme schwierig zu lesen.

Vermeiden Sie lange Ausdrücke, und unterbrechen Sie diese vor einem Operator oder Komma, oder beginnen Sie den Ausdruck in einer neuen Zeile.

Sehr leicht automatisierbar in modernen Programmierumgebungen!