

Einführung in Java und Konzepte objektorientierter Programmierung

Kristof Van Laerhoven
kvl@eti.uni-siegen.de

Michael Möller
michael.moeller@uni-siegen.de

Andreas Hoffmann
andreas.hoffmann@uni-siegen.de

Woche 1

- 1. Erstellen und Ausführen von Java-Programmen
- 2. Syntaktische Grundelemente in Java
- 3. Typen und Variablen
- 4. Anweisungen

Woche 2

- 5. Arrays und Strings

Woche 3

- 6. Objekte und Methoden, UML Teil 1

Woche 4

- 7. Vererbung und Polymorphie, UML Teil 2

Woche 5

- 8. Ausnahmebehandlung (Exceptions)

Woche 6

- 9. Dateien, Ströme und Serialisierung

Woche 7

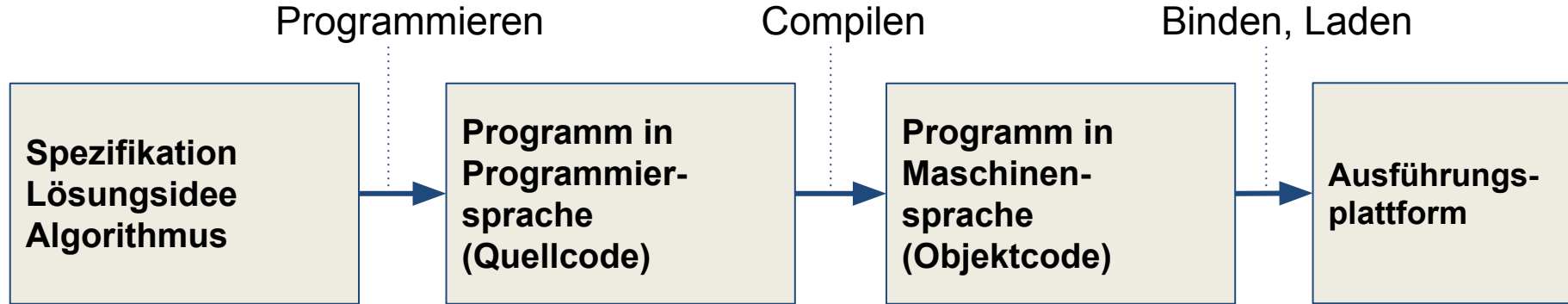
- 10. Threads und Sockets

Zusammenfassung

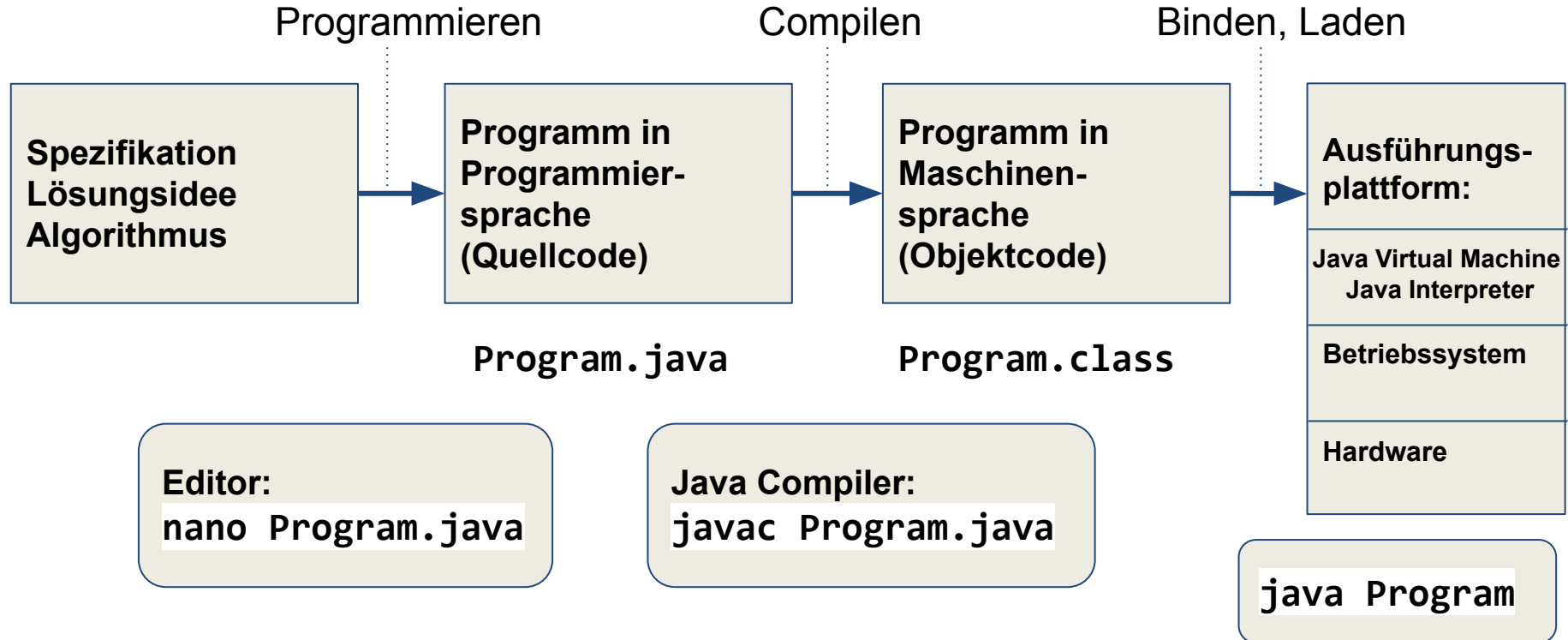
- <https://docs.oracle.com/javase/tutorial/>
- <https://beginnersbook.com/java-tutorial-for-beginners-with-examples/>
- <https://codingbat.com/java>
- <https://www.javabuch.de>

- <https://github.com/kristofvl/OFP>

Entstehungsschritte eines Programms:



Entstehungsschritte eines Programms in Java:



1.1. Programmstruktur:

- Jedes Java-Programm besteht aus mindestens einer Klasse
- Der Programmcode einer öffentlichen Java-Klasse steht in einer eigenen Quelldatei, die den Namen der Klasse trägt
 - private Klassen können in beliebigen Dateien stehen
- Eine Java Quelldatei hat die Endung .java

Anmeldung.java

```
public
class Anmeldung
{
    ...
}
```

Student.java

```
public
class Student
{
    ...
}
```

Hoersaal.java

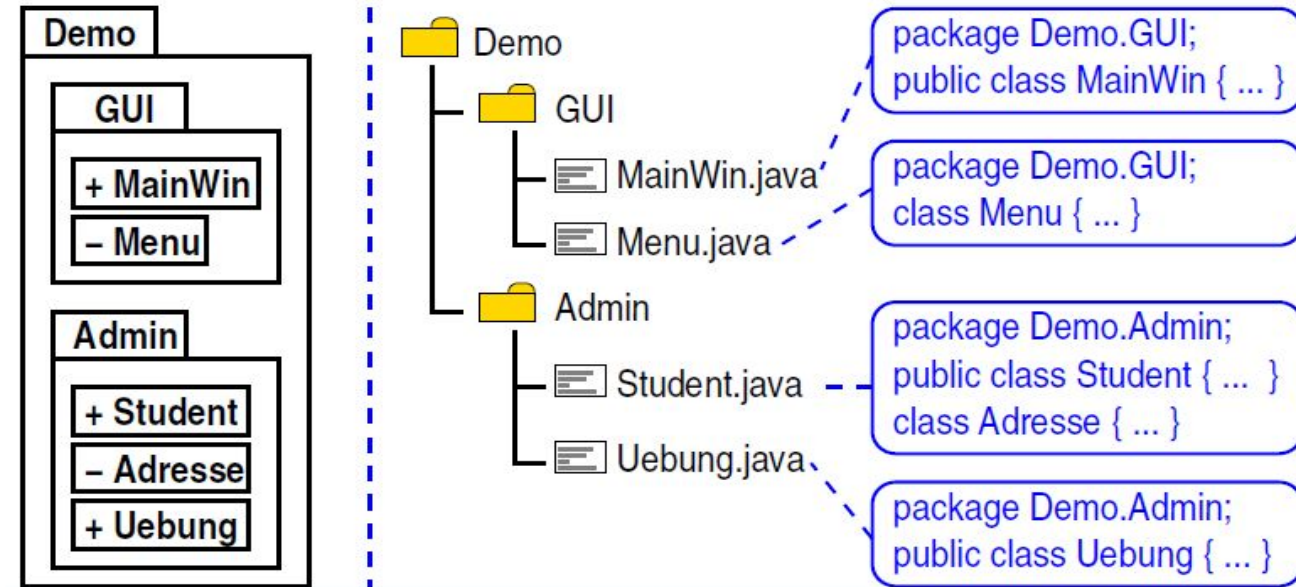
```
public
class Hoersaal
{
    ...
}
```

1. Erstellen von Programmen

```
/** The Birthday Paradox -- an illustration
 * Author:  kv1
 * Date:    first week
 */
class BDayParadox {
    /* p(x) --> probability of x happening
    p(x) = 1 - p(not x)
    p(2 persons have same birthday) = 1 - (365-1)/365 * (365-2)/365 * ... * (365 - (n-1))/365 */
    double prob(int n) {
        double p = 1.0;    // probability that out of n people, 2 have the same birthday
        for (int i = 0; i < n; i++) {    // i = 1, 2, ..., n-1
            p = p * (365-i)/365;
        }
        return 1 - p;
    }
    public static void main(String[] args) {
        int n = 23;
        BDayParadox b = new BDayParadox();
        System.out.println( "Out of " + n +
                           " people, the chance that 2 have the same birthday is: " + b.prob(n) );
    }
}
```

1.2. Programmstruktur und Pakete:

- Die Verzeichnisstruktur von Java-Quelldateien sollte der Paketstruktur entsprechen:



1.3. Übersetzung:

- Aufruf des Java-Compilers: `javac <name>.java`
 - Compiler versucht auch, alle weiteren benötigten Quelldateien zu übersetzen
 - ggf. alle Dateinamen angeben, z.B.: `javac *.java`
- Bei Verwendung von Paketen:
 - Java-Compiler im Wurzel-Verzeichnis starten
 - z.B. `javac Demo/GUI/MainWin.java`
 - oder: Option `-classpath` nutzen, um Wurzelverzeichnis anzugeben
 - z.B. `javac -classpath /home/meier/code Uebung.java`
- Compiler erzeugt für jede Klasse eine `.class`-Datei

1.4. Ausführung von Java-Programmen:

- Interpretation durch eine virtuelle Maschine
 - JVM: Java Virtual Machine
 - sie liest bei Bedarf `.class`-Dateien ein und arbeitet bei Operations-Aufrufen den Programmcode der Operation ab
- Java-Programme werden also nicht direkt vom Prozessor des Rechners ausgeführt:
 - Vorteile: Portabilität und Sicherheit
 - Java-Code läuft auf jedem Rechner, unabhängig von Prozessortyp und Betriebssystem
 - die JVM kann Zugriffe des Programms auf Ressourcen des Rechners (z.B. Dateien) einschränken
 - Nachteil: geringere Ausführungsgeschwindigkeit

1.5. Starten eines Java-Programms:

- Das Kommando `java <Klassenname>` startet eine JVM
 - die JVM lädt zunächst die angegebene Klasse
 - anschließend versucht sie, die Methode
`public static void main(String[] args)`
auszuführen
 - existiert diese Methode nicht, erfolgt eine Fehlermeldung
 - falls die Klasse in einem Paket liegt, muß der vollständige Name angegeben werden, z.B. `java MyPacket.MyClass`
- Bei Bedarf lädt die JVM während der Programmausführung weitere Klassen nach: damit diese gefunden werden, muß ggf. ein Klassen-Pfad (`-classpath`) definiert werden

1.6. Java-Klassenpfad:

- Wird vom Compiler und der JVM benutzt, um den Code von benötigten Klassen zu finden
- Besteht aus einem/mehreren Verzeichnissen oder Java-Archiven
 - Trennzeichen ':' (Linux) bzw. ';' (Windows)
 - z.B.: `java -classpath /lib/classes:. MyPkt.MyClass`
 - sucht in `/lib/classes` und im aktuellen Verzeichnis .
 - die Datei `MyClass.class` muß sich in einem Unterverzeichnis `MyPkt` befinden!
 - z.B.: `java -classpath /lib/myCode.jar MyClass`
 - sucht nur im Java-Archiv `/lib/myCode.jar`
 - ein Java-Archiv enthält einen kompletten Dateibaum (ggf. mit mehreren Paketen / Klassen)

1.6. Java-Klassenpfad:

- Der Klassenpfad kann auch über eine Umgebungsvariable gesetzt werden:
 - z.B.: `export CLASSPATH=/lib/classes:.` (in Linux)
`set CLASSPATH=D:\lib\classes;.` (in Windows)
- Der so definierte Klassenpfad gilt sowohl für den Compiler als auch die JVM

1.7. Entwicklungsumgebungen für Java-Programme:

- Entwicklungsumgebungen integrieren Editor, Java-Compiler und weitere Werkzeuge, u.a.
 - UML-Diagramme inkl. Erzeugung von Code-Rahmen
 - Erzeugung von Programmdokumentation
 - Debugger zur Fehlersuche (siehe später)
- Vorteil: durchgängige, einheitliche Software-Umgebung
- Beispiele:
 - **BlueJ**: speziell für die Ausbildung, sehr einfache UML-Unterstützung, Objekterzeugung
 - **Eclipse**: professionelle Umgebung, UML-Unterstützung mit Plugin
- selbst versuchen: → [Moodle](#): `3DGame.zip`

2.1. Reservierte Schlüsselwörter:

abstract	const	float	int	protected	throw
boolean	continue	for	interface	public	throws
break	default	future	long	rest	transient
byte	do	generic	native	return	true
byvalue	double	goto	new	short	try
case	else	if	null	static	var
cast	extends	implements	operator	super	void
catch	false	import	outer	switch	volatile
char	final	inner	package	synchronized	while
class	finally	instanceof	private	this	

Ablaufkontrolle

Konstante

andere (Deklarationen)

Datentypen

Objekt-Orientierung

reserviert für Erweiterungen

2.2. Namen (Identifikatoren):

- Für Klassen, Attribute, Operationen, Parameter, Variablen,...
- Können beliebige Länge haben
- Dürfen nur aus Buchstaben, Ziffern, '_' und '\$' bestehen
- Müssen mit einem Buchstaben, '_' oder '\$' beginnen
- Dürfen kein Schlüsselwort sein

Beispiele:	korrekt	falsch	Grund
	Summe	get Name	Leerzeichen verboten
	getName	Tuer-1	'-' verboten
	\$a114you	2hoch4	Ziffer am Anfang
	_1_2	while	reserviertes Wort

2.3. Konstanten:

2002 -2L 0xFABEL 2.1 0.1E-23 .1e+19 'a' '\n'
true false null "Hallo" ...

2.4. Operatoren:

+ - * / & && = == >= *= > >> >>> ...

2.5. Klammern:

() [] { }

2.6. Trennzeichen:

, ; . sowie Leerräume, Tabstops und Zeilenwechsel

2.7. Kommentare:

```
// bis zum Ende der Zeile
```

```
/* über mehrere Zeilen  
   hinweg bis zu */
```

```
/** Dokumentation  
    (automatisch extrahiert durch javadoc)  
*/
```

falsch:

```
/* Kommentar darf */ nicht enthalten */
```

```
// probability that out of n people, 2
```

```
/* p(x) --> probability of x happening  
   p(x) = 1 - p(not x)  
   p(2 persons have same birthday) =  
   ...*/
```

```
/** The Birthday Paradox -- an illustra  
 *   Author:   kv1  
 *   Date:     last week  
 */
```

3.1. Vorbemerkung: Variablen

- Eine Variable bezeichnet einen Speicherbereich, der Daten eines gegebenen Typs speichern kann
- In Java gibt es drei Arten von Variablen:
 - **Datenfelder** (in Objekten): speichern die Attributwerte von Objekten
 - **formale Parameter** (in Methoden)
 - speichern die Werte der beim Aufruf der Methode übergebenen Werte (aktuelle Parameter)
 - existieren nur, während die Methode ausgeführt wird
 - **lokale Variable** (in Methoden)
 - existieren nur, während die Methode ausgeführt wird
 - dienen der temporären Speicherung von Daten

3.1. Vorbemerkung: Variablen

- Eigenschaften von Variablen in Java:
 - jede Variable muss vor ihrer Benutzung deklariert werden
 - der Wert einer Variablen kann durch eine Zuweisung jederzeit geändert werden
 - eine Variable kann überall stellvertretend für ihren Wert verwendet werden
 - eine Variable hat einen genau definierten Gültigkeitsbereich, in dem sie verwendet werden darf
 - jede Variable hat eine Lebensdauer, während der sie existiert

3.2. Datentypen

- Ein Typ definiert
 - eine Menge von Werten
 - die darauf anwendbaren Operationen
- Alle Variablen und auch alle Konstanten besitzen in Java zwingend einen Typ
 - bei Konstanten durch die Form (Syntax) festgelegt
 - bei Variablen durch deren Deklaration festgelegt
- Datentypen in Java:
 - **byte, short, int, long, float, double, char, boolean**
 - Arrays (Felder), Strings
 - plus: Klassen als benutzerdefinierte Typen

3.2. Datentypen

- Ganze Zahlen (integer)
 - Vier Typen mit unterschiedlichen Wertebereichen:

Typ	Bytes	Bits	Wertebereich
byte	1	8	-128 ... 127
short	2	16	-32 768 ... 32 767
int	4	32	-2 147 483 648 ... 2 147 483 647
long	8	64	$\sim -9 \cdot 10^{18} \dots 9 \cdot 10^{18}$

- Operationen:
 - Arithmetische-, Vergleichs-, Bit-, Zuweisungs-Operationen
 - Typkonversion
- Fehlerbehandlung:
 - Division (/ und %) durch 0: Ausnahme (siehe später)
 - Bei Überlauf: keine Behandlung (--> falsches Ergebnis)

Beispiel IntDemo

```
/**
Lassen Sie die Variable y von 1 bis 3 variieren und geben
Sie ihren neuen Wert jedes Mal in der Ausgabekonzole aus.
*/

class IntDemo {
    public static void main( String[] str ) {
        int y; // Variable für Integer
        y = 1; // Variable y wird auf den Wert 1 gesetzt
        System.out.print( "Zahl: " ); // Ausgabe "Zahl:"
        System.out.println( y ); // Ausgabe ys Wert + Sprung zur nächsten Zeile
    }
}
```

Beispiel Bits

```
/**
Die Integer-Variable kann auch anders dargestellt werden, zum Beispiel
als eine Reihe von Bits.
Binär zu Dezimal, https://docs.oracle.com/javase/8/docs/technotes/guides/language/binary-literals.html
*/

class Bits {
    public static void main( String[] str ) {
        int y; // Variable für Integer
        y = 0b01010101; // Bitfolge 01010101
        // Diese Anweisungen geben den Wert der Variablen y in der Konsole aus:
        System.out.print( "Binärzahl zu dezimal: " );
        System.out.println( y );    }
}
```


3.2. Datentypen

- Gleitkomma-Zahlen (floating point)
 - Zwei Typen mit unterschiedlichen Wertebereichen:

Typ	Bytes	Bits	Wertebereich	Genauigkeit
float	4	32	-3.4e38 ... 3.4e38	7 Stellen
double	8	64	-1.7e308 ... 1.7e308	17 Stellen

- Operationen:
 - arithmetische (incl. %), Vergleichs-, Zuweisungs- Operationen
 - Typkonversion
- Fehlerbehandlung:
 - Bei Überlauf und Division durch 0: [-] Infinity
 - Bei 0.0 / 0.0 und ähnlichem: Nan (Not a Number)

3.2. Datentypen

- Gleitkomma-Zahlen (floating point)
 - Reelle Zahlen können im Computer nicht exakt dargestellt werden
 - Java: IEEE 754 Standard (8 Byte, 17 Stellen Genauigkeit)
 - Dadurch entstehen Rundungsfehler
 - In der Gleitkomma-Arithmetik gelten Assoziativitäts- und Kommutativitätsgesetz nicht mehr
 - Beispiel:

```
double a, b, x, y;  
a = 5.0e-12;      // a =  0.000 000 000 005  
b = 1.0e+5;       // b = 100 000.000 000 000 00  
x = a + a + b;    // x = 100 000.000 000 000 01  
y = b + a + a;    // y = 100 000.000 000 000 00
```

3.2. Datentypen

● Einzelzeichen (char)

- | Typ | Bytes | Bits | Wertebereich |
|------|-------|------|---|
| char | 2 | 16 | Unicode (65536 Zeichen) \supset ASCII |
- Operationen:
 - Vergleichs-, Zuweisungs-Operationen
 - Inkrement, Dekrement und =
 - Typkonversion
- Fehlerbehandlung:
 - Bei Überlauf: Abschneiden der oberen Bits

Beispiel CharASCII (🌶️)

```
/**
ASCII-Tabelle: Kodierung für Zeichensätze
https://de.wikipedia.org/wiki/American\_Standard\_Code\_for\_Information\_Interchange
*/

class CharASCII {
    public static void main( String[] str ) {
        char symbol; // Variable für Zeichen
        symbol = '@';
        System.out.print( "Die ASCII-Kodierung für [" + symbol + "] ist " );
        System.out.println( (int) symbol ); // Methode zum Drucken auf die Konsole
    }
}
```

Beispiel Smiley (🌶️🌶️🌶️🌶️)

```
/**
Mit UniCode können mehr Symbole gedruckt werden (wenn das Terminal
UniCode akzeptiert): https://en.wikipedia.org/wiki/List\_of\_Unicode\_characters
https://docs.oracle.com/javase/6/docs/api/java/lang/Character.html#unicode
https://stackoverflow.com/questions/5903008/what-is-a-surrogate-pair-in-java
*/
class Smiley {
    public static void main( String[] str ) {
        char c1 = 0x7117; // Unicode Zeichen für: Soße aus Sojasoße, Mirin und Zucker
        System.out.println( c1 );
        char c2 = 0xD83D; // Unicode Zeichen 'GRINNING FACE' in hexadezimal,
        char c3 = 0xDE00; // als "surrogate pair"
        System.out.println( c2 + "" + c3 ); // für später: Wieso ""?
        char c4 = 7; // ASCII Zeichen für BEEP
        System.out.println( c4 );
    }
}
```

3.2. Datentypen

- Wahrheitswerte (boolean)

- | Typ | Wertebereich |
|---------|-----------------------------|
| boolean | true (wahr), false (falsch) |

- Operationen:

- Vergleich (nur == und !=), Zuweisungs-Operationen
- Boole'sche Operatoren: &&, ||, &, |, ^, !
- keine Typkonversion

- Vergleichsoperatoren erzeugen Ergebnis vom Typ boolean

- Beispiel:

```
boolean ende;  
ende = (zahl >= 100) || (eingabe == 0);
```

3.3. Konstanten

- Explizite Datenwerte im Programm
- Können bereits vom Compiler interpretiert werden
- Besitzen einen Typ, ersichtlich an ihrer Syntax:

Konstante	Typ
1, -9999, 0xFABE, 0125	int
1L, -9999L, 0xFABEL, 0125L	long
1.0, 2002.5d, 3.14159E8, 05e-9D	double
1.0f, 2002.5f, 3.14159E8f, 05e-9F	float
'A', '\u0041', '\\'', '\\\\', '\\n'	char
"String", "\"Hallo\"", sagte er\\n"	String (Klasse!)
true, false	boolean

3.4. Variablen

Deklaration

- Der Typ von Variablen wird durch ihre Deklaration festgelegt
- Eine Deklaration ist eine Anweisung
- Der Modifier **final** zeigt an, daß die initialisierte Variable nicht veränderbar ist (*"benannte Konstante"*)
- Beispiele:
 - `int x, y;`
 - `int zaehler = 0, produkt = 1;`
 - `final double PI = 3.14159265358979323846;`
 - `boolean istPrim;`

3.4. Variablen

Gültigkeitsbereich ("Scope")

- Eine in einer Methode deklarierte Variable (= lokale Variable) ist ab der Deklaration bis zum Ende des umgebenden Blocks (siehe [4.3](#)) gültig

```
{  
    int a = 3, b = 1;  
    if (a > 0) {  
        c = a + b; // Fehler: c ungültig  
        int c = 0;  
        c += a; // OK  
    }  
    b = c; // Fehler: c ungültig  
}
```

3.4. Variablen

Gültigkeitsbereich ("Scope")

- Die Gültigkeitsbereiche von lokalen Variablen mit demselben Namen dürfen nicht überlappen (siehe Beispiel ScopeDemo):

```
{  
    int a = 3, b = 1;  
    if (a > 0) {  
        int b;    // Fehler: b bereits deklariert  
        int c;    // OK  
    }  
    {  
        char c;   // OK  
    }  
    double a;     // Fehler: a bereits deklariert  
}
```

3.4. Typkonversionen

- Der Typ eines Ausdrucks kann (in Grenzen) angepaßt werden:
 - explizite Typkonversion
 - implizite (automatische) Typkonversion: `double d; int i; d = i + 1;`
- Explizite Typkonversion:
 - durch Voranstellen von (<Typ>), z.B.:

```
(double) 3 // == 3.0
(char) 65 // == 'a' → ASCII
(int) x + y // == ((int) x) + y
(int) (x + y)
```
- Bei Typkonversionen kann Information verloren gehen:

```
double d = 1.337; (int) d // == 1
```
- keine Konversion von / nach boolean: `boolean b = true; b = 32; // error`

3.4. Typkonversionen

- Implizite Typkonversion
 - Immer nur von "kleineren" zu "größeren" Typen:
byte → short → int → long → float → double
 ↑
 char
 - In Ausdrücken:

Mind. ein Op.	andere Operanden	konv. zu
double	double, float, long, int, short, byte, char	double
float	float, long, int, short, byte, char	float
long	long, int, short, byte, char	long
int, short, byte, char	int, short, byte, char	int

3.4. Typkonversionen

```
double x, y = 1.1;
int i, j = 5;
char c = 'a';
boolean b = true;
x = 3;           // OK: x == 3.0
x = 3/4;         // x == 0.0
x = c + y * j;   // OK, Konversion nach double
i = j + x;       // Fehler: (j + x) hat Typ double
i = (int)x;      // OK, i = ganzzahl. Anteil von x
i = (int)b;      // Fehler: keine Konversion von boolean
b = i;           // Fehler: keine Konversion zu boolean
b = (i != 0);    // OK, Vergleich liefert boolean
c = c + 1;       // Fehler: c + 1 hat Typ int
c += 1;          // OK! c == 'b'
```

3.4. Typkonversionen

```
class TypUebung {  
    public static void main( String[] s ) {  
        /* Deklarieren und initialisieren Sie die folgenden Variablen und geben Sie  
           sie in der Konsole aus: */  
        /* Variable zur Speicherung der Weltbevölkerung: */  
        /* Variable zur Speicherung der Bevölkerungszahl eines Landes,  
           gerundet in Millionen : */  
        /* Variable zur Speicherung des prozentualen Anteils der Bevölkerung eines  
           Landes an der Weltbevölkerung: */  
        /* Variable, die speichert, ob die Bevölkerung eines Landes mehr als 1%  
           der Weltbevölkerung ausmacht: */  
        /* Variablen zum Speichern dieses Symbols: 😊 */  
        /* Konstante zum Speichern des Flaggensymbols eines Landes, z.B. 🇧🇪 : */  
        /* Konstante zum Speichern des Namens eines Landes, z.B. Belgien : */  
    }  
}
```

4.1. Ausdrücke und Zuweisungen

4.2. Ausdrücke und Priorität

4.3. Anweisungsfolge / Block

4.4. Auswahl-Anweisungen

4.4.1. Die **if**-Anweisung

4.4.2. Die **switch**-Anweisung

4.5. Wiederholungs-Anweisungen

4.5.1. Die **while**-Schleife

4.5.1. Die **do while**-Schleife

4.5.1. Die **for**-Schleife

4.6. Programmier-Konventionen

4.1. Ausdrücke und Zuweisungen:

- Ein Ausdruck berechnet einen Wert
- Ein Ausdruck kann u.a. folgende Formen annehmen:

```
int x, y, produkt;  
boolean b;  
x = 2; produkt = 1;  
y = x * 7 / 2;           // y = ?  
b = ( (x + y) >= 13 );   // b = ?  
x += -5;                 // x = ?  
y = y | 8;               // y = ?  
produkt *= x - y;        // produkt = ?  
b = produkt != 3117;     // b = ?
```


4.1. Ausdrücke und Zuweisungen

Erlaubte Operatoren:

Arithmetische Operatoren:	+	-	*	/	%	++	--
Bitoperatoren:	&		^	~	<<	>>	>>>
Vergleichsoperatoren:	>	>=	==	!=	<=	<	? :
Logische Operatoren:	&	&&			!		
Zuweisungsoperatoren:	=	+=	-=	*=	/=	%=	
	&=	=	^=	<<=	>>=	>>>=	

Unärer Operator

Binärer Operator

Unärer und binärer Operator

Ternärer Operator

4.1. Ausdrücke und Zuweisungen

Arithmetische Operatoren:

```
int x, y, breite, laenge, produkt, rest, quotient;

breite    = x + y;           // Addition
laenge    = laenge - 1;     // Subtraktion
produkt    = x * y;         // Multiplikation
quotient  = x / y;         // Division
rest      = x % y;         // Modulo (Restoperation)
x++;      // Inkrement: x = x + 1
x--;      // Dekrement: x = x - 1
laenge = 4 * (x + y);      // Klammerung
laenge = breite = 1;      // Eine Zuweisung ist wieder ein Ausdruck
produkt = (x = 2) * (y = 5); // Möglich, aber schlechter Stil
```

4.1. Ausdrücke und Zuweisungen

Bitoperatoren:

```
int x, y, z;

x = y & z;           // UND
x = y | z;           // ODER
x = y ^ z;           // Exklusives ODER
x = ~ y;             // Komplement
x = y << 1;          // Linksschieben (x = y * 2)
x = y >> 1;          // Rechtsschieben (schiebt Vorzeichen nach)

x = y >>> 4;         // Rechtsschieben (schiebt 0 nach)
x = x | 010;         // Setzt Bit 3 in x (010 == 8, Oktalzahl)
y = (x & 0xF0) >> 4; // y = Bits 7..4 von x (0xF0 == 240, Hexadezimal)
```

4.1. Ausdrücke und Zuweisungen

Vergleichsoperatoren:

```
int x, y, summe, zaehler, minumum;

if (x > y) ...           // größer
if (x >= y) ...          // größer oder gleich
if (summe == x + y) ...  // gleich
if (x != y) ...          // ungleich
if (zaehler < 100) ...   // kleiner
if (x + 1 <= 1 - y) ...  // kleiner oder gleich

minimum = (x < y)? x : y; // Operator mit 3 Operanden:
                        // Falls ( x < y ), dann x, sonst y
```

4.1. Ausdrücke und Zuweisungen

logische Operatoren:

```
int x, y, zaehler;  
boolean leseFlag, schreibFlag;  
  
if (leseFlag || schreibFlag) ... // logisches ODER, schreibFlag nicht  
                                // ausgewertet wenn leseFlag == true  
if (leseFlag | schreibFlag) ... // logisches ODER, immer beide  
                                // Operanden ausgewertet  
if ((x != 0) && (y / x < 5)) ... // logisches UND, (y / x < 5) nicht  
                                // ausgewertet wenn x == 0  
if (leseFlag & (zaehler < 10)) ... // logisches UND, immer beide  
                                // Operanden ausgewertet  
if (!(x < 0)) ...                // entspricht if (x >= 0)
```

4.1. Ausdrücke und Zuweisungen

Zuweisungen / Zuweisungsoperatoren:

- Eine Zuweisung ist eine Anweisung, die einer Variablen einen Wert zuweist: Sie speichert den Wert im Speicherbereich der Variablen ab

- z.B.

```
int summe, x, y, rest;  
x = y = 7;           // eine Zuweisung ist wieder ein Ausdruck  
summe = x + y / 2;   // summe = ( x + (y/2) )  
x += -5;             // x = x + (-5)  
rest %= x - y / 2;   // rest = rest % ( x - y/2 )
```

- Schritte:
 1. bestimme Speicherbereich der Variablen
 2. berechne Wert des Ausdrucks
 3. speichere Wert im Speicherbereich ab

4.2. Ausdrücke und Priorität

- Reihenfolge richtet sich nach der Priorität der Operatoren
 - Operatoren höherer Priorität werden vor jenen niedrigerer Priorität ausgewertet
 - bei Operatoren gleicher Priorität nach der Assoziativität von links nach rechts / von rechts nach links
- Gute Praxis: Im Zweifelsfall Klammern verwenden

Prio.	Operatoren	Assoziativität	Prio.	Operatoren	Assoziativität
14	(), [], postfix ++, postfix --	von links	6	&	von links
13	unäres +, unäres -, präfix ++, präfix --, ~, !	von rechts	5	^	von links
12	(<Typ>), new	von links	4		von links
11	*, /, %	von links	3	&&	von links
10	+, -	von links	2		von links
9	<<, >>, >>>	von links	1	?:	von links
8	<, <=, >, >=	von links	0	=, +=, -=, *=, /=, %=, <<=, >>=,	von rechts
7	==, !=	von links	0	>>>=, &=, =, ^=	von rechts

4.2. Ausdrücke und Priorität

- Beispiele:
Priorität und Assoziativität

```
a << b * c + d;    // a << ((b * c) + d)
a / b * c % d;     // (((a / b) * c) % d)
~a;                // ~(a)
a += b = c + d;    // a += (b = c + d)
a+++b;             // a++ + b
```

Präfix und Postfix Inkrement / Dekrement:

```
a = 10;
b = ++a;
// a = 11, b = 11
```

```
a = 10;
b = a++;
// a = 11, b = 10
```


4.3. Anweisungsfolge / Block

- In einer Anweisungsfolge werden mehrere Anweisungen hintereinander ausgeführt: `laenge = 10; breite = 15; flaeche = laenge * breite;`
- jede Anweisung wird mit einem Strichpunkt (;) beendet
- Anweisungsfolgen treten z.B. im Rumpf von Methoden auf
- Eine Anweisungsfolge kann mit {} zu einem Block geklammert werden, z.B.:

```
{  
    laenge = 10; breite = 15;  
    flaeche = laenge * breite;  
}
```
- Ein Block ist überall da erlaubt, wo eine Anweisung stehen kann, der abschließende Strichpunkt entfällt dann

4.4. Auswahl-Anweisungen

- Eine Auswahlanweisung dient dazu, die Ausführung von Anweisungen von einer Bedingung abhängig zu machen

- Bedingte Anweisung:

```
if ( <Bedingung> )    // Falls Bedingung erfüllt,  
    <Anweisung> ;    // führe Anweisung aus
```

- Alternativauswahl:

```
if ( <Bedingung> )    // Falls Bedingung erfüllt,  
    <Anweisung1> ;    // führe Anweisung1 aus  
else                  // sonst:  
    <Anweisung2> ;    // führe Anweisung2 aus
```

4.4. Auswahl-Anweisungen

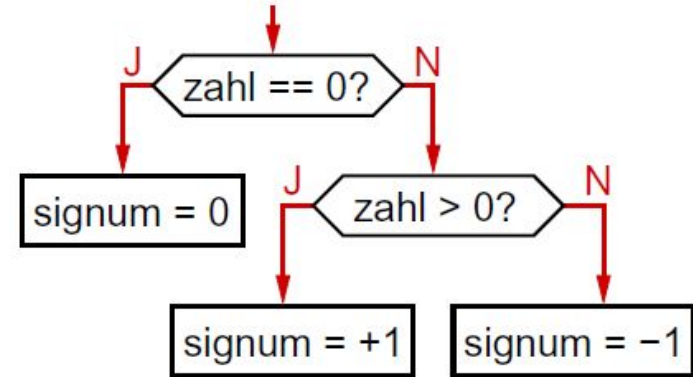
Beispiel: Berechne das Signum einer Zahl

Mit if-Anweisung:

```
if (zahl == 0)
    signum = 0;
else
    if (zahl > 0)
        signum = +1;
    else
        signum = -1;
```

Mit Auswahloperator:

```
signum = zahl == 0 ? 0 : ( zahl > 0 ? +1 : -1 );
```



4.4. Auswahl-Anweisungen

- Verschachtelte **if**-Anweisungen

```
if ( <Bedingung1> )  
    if ( <Bedingung2> )  
        <Anweisung1> ;  
else  
    <Anweisung2> ;
```



```
if ( <Bedingung1> ) {  
    if ( <Bedingung2> )  
        <Anweisung1> ;  
}  
else  
    <Anweisung2> ;
```

```
if ( <Bedingung1> )  
    if ( <Bedingung2> )  
        <Anweisung1> ;  
else  
    <Anweisung2> ;
```



```
if ( <Bedingung1> ) {  
    if ( <Bedingung2> )  
        <Anweisung1> ;  
else  
    <Anweisung2> ;  
}
```

4.4. Auswahl-Anweisungen

- Verschachtelte **if**-Anweisungen:
Beispiel Menuauswahl:

```
if (menuItem == 1) {  
    ...  
} else if (menuItem == 2) {  
    ...  
} else if ((menuItem == 3) || (menuItem == 4)) {  
    ...  
} else if (menuItem == 5) {  
    ...  
} else {  
    ...  
}
```

4.4. Auswahl-Anweisungen

- Die **switch**-Anweisung wählt abhängig vom Wert eines Ausdrucks eine von mehreren Alternativen aus:

```
switch (menuItem) {  
    case 1: ...  
        break;  
    case 2: ...  
        break;  
    case 3:  
    case 4: ...  
        break;  
    case 5: ...  
        break;  
    default: ...  
        break;  
}
```

```
switch ( <Ausdruck> ) {  
    case <Wert1>:                // Ausdruck muss ganzzahlig sein  
        <Anweisungen1>         // Falls Ausdruck == Wert1:  
        break;                 //   Anweisungen1 ausführen  
    case <Wert2>:                //   verlasse den switch-Block  
        <Anweisungen2>         // Falls Ausdruck == Wert2:  
        break;                 //   Anweisungen2 ausführen  
    ...                          //   verlasse den switch-Block  
    default:                    // Falls keiner dieser Fälle:  
        <Anweisungen3>         //   Anweisungen3 ausführen  
        break;                 //   verlasse den switch-Block  
}
```

4.5. Wiederholungs-Anweisungen

4.5.1. Die **while**-Schleife:

```
int i, summe;  
summe = 0;  
i = 1;  
while ( i < 10 ) {  
    summe += i;  
    i++;  
}
```

```
while ( <Wiederholungsbedingung> )  
    <Anweisung> ;
```



Diese Anweisung wird wiederholt ausgeführt
(Schleifenrumpf)

Spezielle Anweisungen (in einem Block):

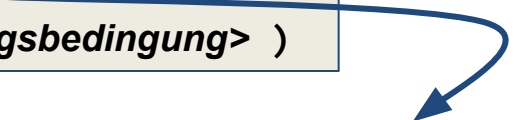
continue	verlasse den Block und beginne mit dem nächsten Durchlauf
break	verlasse die Schleife

4.5. Wiederholungs-Anweisungen

4.5.2. Die **do while**-Schleife:

```
int i, summe;  
summe = 0;  
i = 1;  
do {  
    summe += i;  
    i++;  
} while ( i <= 10 );
```

```
do  
    <Anweisung> ;  
while ( <Wiederholungsbedingung> )
```



Diese Anweisung wird wiederholt und mindestens einmal ausgeführt (Schleifenrumpf)

Spezielle Anweisungen (in einem Block):

continue	verlasse den Block und beginne mit dem nächsten Durchlauf
break	verlasse die Schleife

4.5. Wiederholungs-Anweisungen

4.5.3. Die for-Schleife:

```
int i, summe;  
summe = 0;  
for (i = 1; i <= 10; i++)  
    summe += i;
```

```
int i, summe;  
for (i = 1, summe = 0;  
     i <= 10;  
     summe += i, i++) ;
```

schlechter Stil, nur zur Demonstration:
mehrere, mit Komma getrennte Ausdrücke
und ein leerer Rumpf sind möglich

Initialisierung der
Schleifenvariablen

Wiederholungs-
bedingung

Weiterzählen der
Schleifenvariablen

```
for ( <Ausdruck1> ; <Bedingung> ; <Ausdruck2> )  
    <Anweisung> ;
```

Diese Anweisung wird wiederholt ausgeführt
(Schleifenrumpf)

Spezielle Anweisungen (in einem Block):

continue

verlasse den Block und beginne mit
dem nächsten Durchlauf

break

verlasse die Schleife

4.5. Wiederholungs-Anweisungen

- Wann welche Schleife?
 - **for**-Schleife:
 - für Zählschleifen (z.B. " für alle $i = 1 \dots N$ ")
 - wenn natürlicherweise eine Schleifenvariable vorhanden ist / benötigt wird
 - **while**-Schleife:
 - wenn die (maximale) Zahl der Wiederholungen nicht im Voraus bekannt ist,
 - bei komplexen Wiederholungsbedingungen
 - **do while**-Schleife:
 - wenn der Schleifenrumpf mindestens einmal ausgeführt werden soll

4.5. Wiederholungs-Anweisungen

- Beim Entwurf einer Schleife ist zu beachten:
 - Initialisierung
 - Abbruchbedingung
 - Terminierung: der Schleifenrumpf muß irgendwann die Abbruchbedingung herstellen
 - Unnütze Wiederholung von Berechnungen vermeiden
- Häufige Fehler:
 - Falsche / fehlende Initialisierung
 - Falsche Abbruchbedingung (Schleife terminiert nicht)
 - "Off by one"
 - **while** statt **do while** und umgekehrt

Beispiel Schleife01 (🌶️)

/**

Implementiere ein Java-Programm, das mit 999.0 beginnt und dann in einer Schleife die Hälfte der vorherigen Zahl ausgibt, bis die Zahl kleiner als 20.0 ist.

Benutze `System.out.print()` und `System.out.println()` für die Ausgabe.

*/

```
class Schleife01 {  
    public static void main( String[] str ) {  
  
    }  
}
```

Beispiel Schleife02 (🌶️)

```
/**  
    Implementieren Sie ein Java-Programm, das die ASCII-Zeichen von Position  
    127 bis 32 (inklusive) in der ASCII-Tabelle in einer Zeile ausgibt.  
  
    Benutze System.out.print() und System.out.println() für die Ausgabe.  
    */  
  
class Schleife02 {  
    public static void main( String[] str ) {  
  
    }  
}
```

Beispiel Schleife03 (🌶️🌶️)

```
/**  
    Implementiere ein Programm, dass von 1 bis 26 zählt, aber statt der Zahl  
    "hoppla" ausgibt, wenn diese Zahl ein Vielfaches von 7 ist.  
  
    Benutze System.out.print() und System.out.println() für die Ausgabe.  
*/  
  
class Schleife03 {  
    public static void main( String[] str ) {  
  
    }  
}
```

Beispiel Schleife04 (🌶️🌶️)

/**

Implementiere ein Java-Programm, das alle 11 möglichen Noten auflistet: 1,0 2,0 3,0 4,0 5,0 1,3 2,3 3,3 1,7 2,7 3,7

Kann dieses Programm auch mit einer einzigen for-Schleife und einer einzigen if-Bedingung implementiert werden?

Benutze `System.out.print()` und `System.out.println()` für die Ausgabe.

*/

```
class Schleife04 {  
    public static void main( String[] str ) {  
  
    }  
}
```

Beispiel Schleife05 (🌶️🌶️)

```
/**  
    Implementiere ein Java-Programm, das eine zufällige Anzahl von 'X' Zeichen  
    ausgibt. Verwende die Methode Math.random(), die eine Fließkommazahl  
    zwischen 0 und 1 zurückgibt, und höre auf, „X“ zu drucken, wenn sie eine Zahl  
    kleiner als 0,09 zurückgibt. Verwende *keine* Variablen.  
  
    Benutze System.out.print() und System.out.println() für die Ausgabe.  
*/  
  
class Schleife05 {  
    public static void main( String[] str ) {  
  
    }  
}
```


Beispiel Schleife06 (🌶️🌶️🌶️)

```
/**
 * Implementiere ein Java-Programm, das ein Dreieck mit einer bestimmten Breite
 * und Höhe „size“ zeichnet. Diese Variable wird als Kommandozeilenargument aus
 * dem Terminal gelesen (siehe unten). Für "java Schleife06.java 4" z.B.:
 *
 * X
 * XX
 * XXX
 * XXXX
 *
 * Benutze System.out.print() und System.out.println() für die Ausgabe.
 */

class Schleife06 {
    public static void main( String[] str ) {
        int size = Integer.parseInt(str[0]); // size = 1. Argument
    }
}
```

Beispiel Schleife07 (🌶️🌶️🌶️)

```
/**
Implementiere ein Programm, das auf der Konsole ein großes X aus dem
Zeichen X ausgibt, abhängig von der Variablen int size (size = 3, 4, ..., 10):
size = 3:      size = 4:      size = 5:      etc.
  X X          X  X          X   X
   X           XX           X  X
  X X          XX           X
                X  X
                X  X
                X   X

*/

class Schleife07 {
    public static void main( String[] str ) {

    }
}
```

Beispiel Schleife17 (🌶️🌶️🌶️🌶️)

```
/** Implementiere ein Java-Programm, das das Bluetooth-Symbol mit einer bestimmten ungeraden  
Breite mit 'B' und Leerzeichen zeichnet. Verwende 'width' als Breite und zeichne das Symbol  
nur, wenn sein Wert ungerade ist. Für "java Schleife17.java 5" z.B. ist die Ausgabe:
```

```
  B
 BB
B B B
 BBB
  B
 BBB
B B B
  BB
  B
```

```
Benutze System.out.print() und System.out.println() für die Ausgabe. */
```

```
class Schleife17 {
    public static void main( String[] str ) {
        int width = Integer.parseInt(str[0]); // size = 1. Argument
    }
}
```

4.6. Programmier-Konventionen

- Verwenden Sie aussagekräftige Namen
- Verdeutlichen Sie die Programmstruktur durch Einrückungen (Indentation)
- Bei Unklarheit: auch einzelne Anweisungen mit { } klammern
- Nach **if**, **switch**, **for**, **while** Leerraum lassen
- Verwenden Sie Leerzeilen und Kommentare um Programm-abschnitte zu trennen
- Beispiel:

```
int z,t;for(System.out.println(2),z=3;z<100;z+=2){  
for(t=3;t<z&&z%t!=0;t+=2);if(t==z)System.out.println(z);}
```

4.6. Programmier-Konventionen

```
int zahl, teiler;
boolean prim;
// Berechnen der Primzahlen, starte mit 2:
System.out.println(2);
for ( zahl = 3; zahl < 100; zahl += 2 ) { // für zahl = 3, 5, 7, ..., 99:
    prim = true;
    for ( teiler = 3; teiler < zahl; teiler += 2 ) { // für teiler = 3, ..., zahl-2:
        if ( zahl % teiler == 0 ) {
            prim = false;
            teiler = zahl; // verlasse die for-teiler-Schleife
        }
    }
    if ( prim )
        System.out.println(zahl);
}
```

Objektorientierte und Formale Programmierung

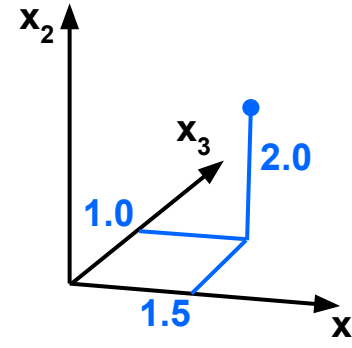
-- Java Grundlagen --

5. Arrays und Strings

5.1. Arrays

- In einem Array kann man mehrere Variablen des gleichen Typs zusammenfassen
- Beispiel: Koordinaten eines Punkts im Raum:
 - Mathematisch: $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = (1.5, 2.0, 1.0)$
 - In Java: Array mit Elementen vom Typ double:

```
double[] x = new double[3];  
x[0] = 1.5;  
x[1] = 2.0;  
x[2] = 1.0;  
// oder kürzer:  
double[] x = { 1.5, 2.0, 1.0 };
```

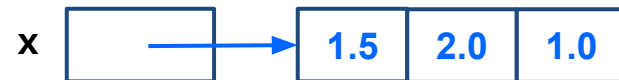


5.1. Arrays

- Arrays müssen dynamisch angelegt werden: **new**
- Das eigentliche Array wird über eine Referenz angesprochen:

```
double pi = 3.14159265;  
double[] x = new double[3];  
x[0] = 1.5;  
x[1] = 2.0;  
x[2] = 1.0;
```

double pi 3.14159265



- Ein Element eines Arrays wird über einen ganzzahligen Index ausgewählt
 - Z.B. `x[1]` oder `x[i+j]`
 - Das erste Element hat den Index `0`
 - Ausnahme bei Indexüberlauf: `IndexOutOfBoundsException`
- Arrays besitzen ein "Attribut" **length**: Anzahl der Elemente `x.length == 3`

5.1. Arrays

● Beispiele (1)

- Deklaration:

```
double[] x;    // Bevorzugte Schreibweise  
double y[];    // Geht auch
```

- Erzeugung des (eigentlichen) Arrays

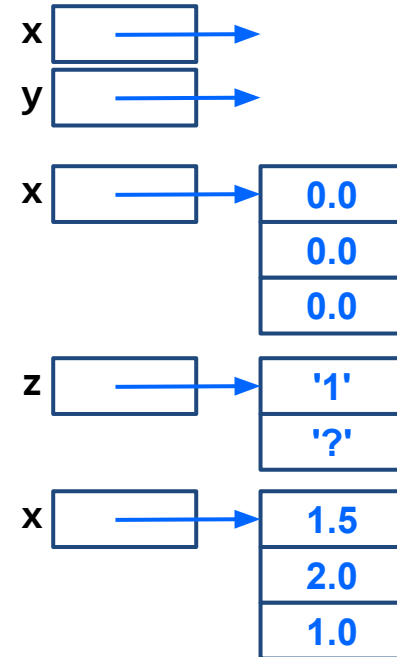
```
x = new double[3]; // 3 Elemente, Initialwerte 0.0
```

- Deklaration, Erzeugung und Initialisierung

```
char[] z = { '1', '?' }; // automatisches new
```

- Zugriff auf Array-Elemente:

```
for (int i = 0; i < x.length; i++)  
    x[i] = 3.0 - i;  
x[0] = 1.5; // ersetzt alten Wert 3.0
```



5.1. Arrays

● Beispiele (2)

- Spezieller Wert `null` verweist nirgendwohin:

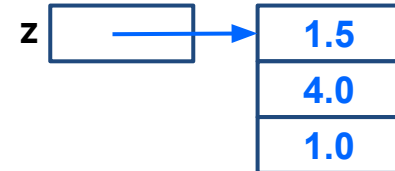
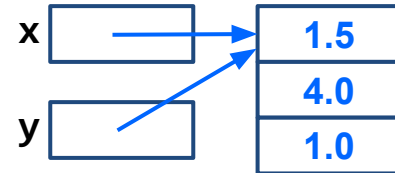
```
double[] y = null;
```

- Zugriffe auf das ganze Array:

```
// erzeugt neue Referenzvariable y, die  
// auf dasselbe Array zeigt wie x:
```

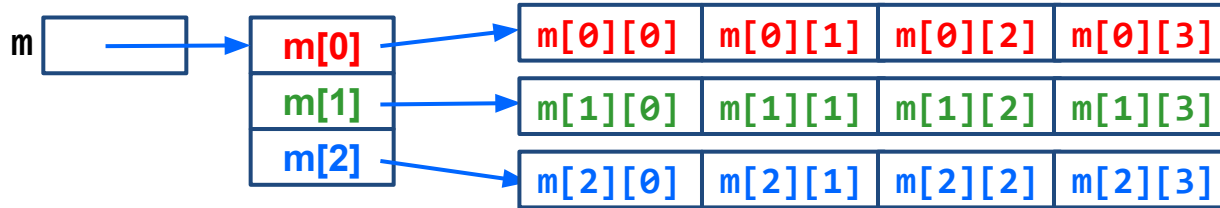
```
double[] y = x;  
// ändert auch den Wert von x[2]:  
y[1] = 4.0;
```

```
// So wird z eine echte Kopie von x:  
double[] z = new double[x.length];  
for (int j = 0; j < x.length; j++)  
    z[j] = x[j];
```



5.1. Arrays

- Mehrdimensionale Arrays
 - Die Elemente eines Arrays können auch Referenzen auf Arrays enthalten:



- Deklaration der Referenzvariable:

```
int[][] m;           // 2-dimensionales Array: Matrix
int m[][];           // 2-dimensionales Array: Matrix, alternativ
int[][][] mat3d;     // 3-dimensionale Matrix
int m3d[][][];       // 3-dimensionale Matrix, alternativ
```

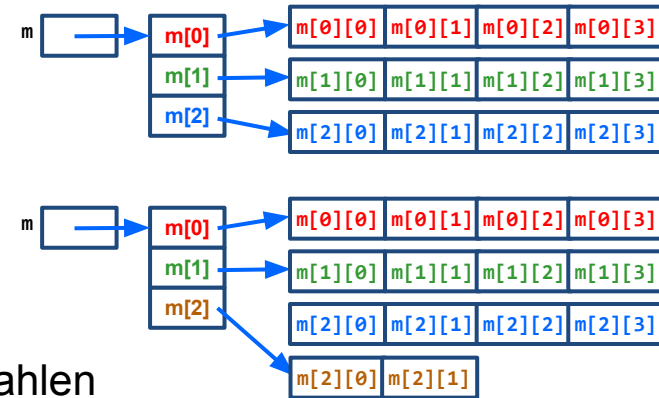
5.1. Arrays

- Mehrdimensionale Arrays
 - Erzeugung des Arrays:

```
m = new int[3][4]; // m = | 0 0 0 0 |  
                        //      | 0 0 0 0 |  
                        //      | 0 0 0 0 |
```

```
m[2] = new int[2]; //      | 0 0 0 0 |  
m[2][1] = 7;      // m = | 0 0 0 0 |  
                        //      | 0 7
```

- `m[2]` ist eine Referenz auf ein Array von `int`-Zahlen
- `m[2]` hat den Typ `int[]`
- `m.length == 3`
- Die Länge der Zeilen kann variieren:
`m[0].length == 4`, aber `m[2].length == 2`



5.1. Arrays

- Mehrdimensionale Arrays

- `m = new int[3][4];` ist gleichbedeutend mit:

```
m = new int[3][];           // Lege Array für 3 Zeilenverweise an
for (int i=0; i<3; i++)     // Initialisiere die
    m[i] = new int[4];      // Zeilenverweise
```

- aber:

`m = new int[][4];` führt zu Fehlermeldung, da

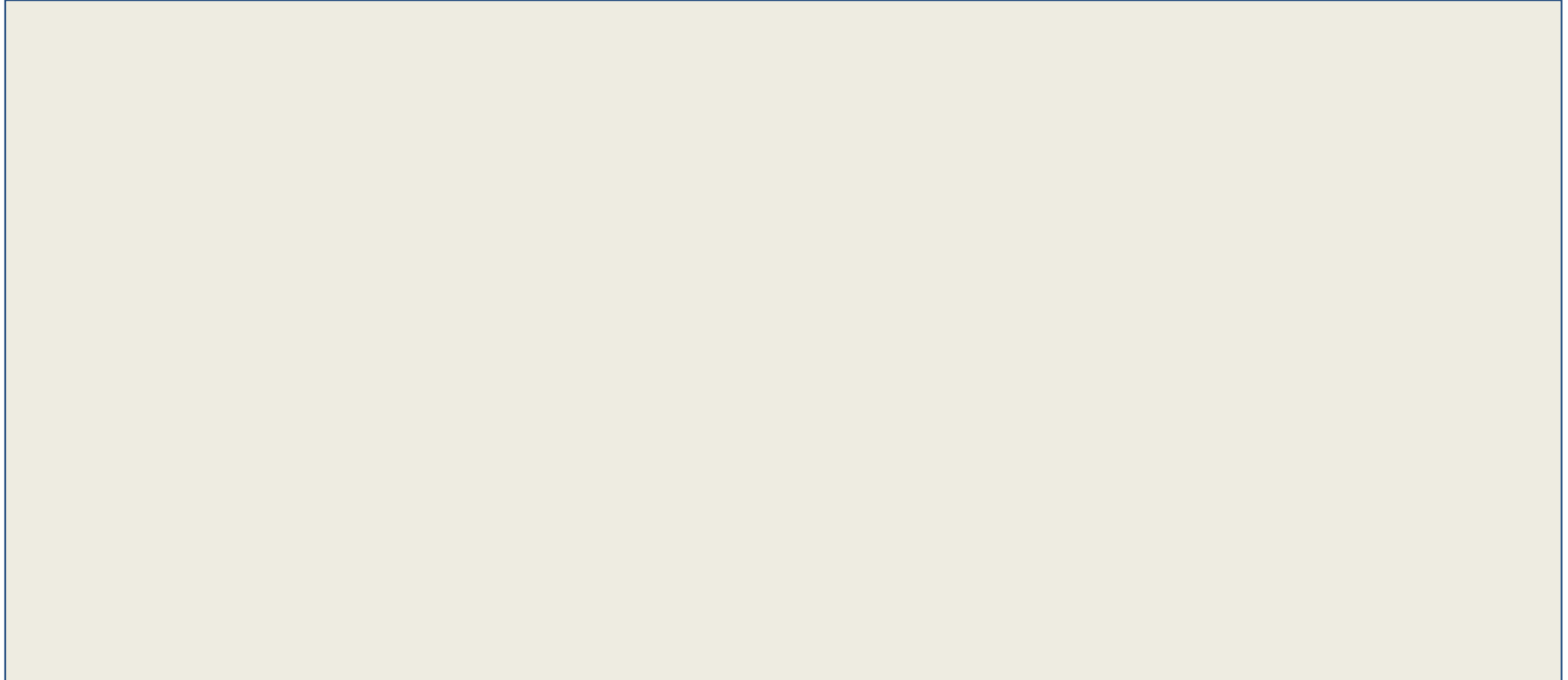
- die Referenzen auf die Zeilen nirgends abgespeichert werden können
- die Anzahl der zu erzeugenden Zeilen unbekannt ist

Beispiel (🌶️🌶️)

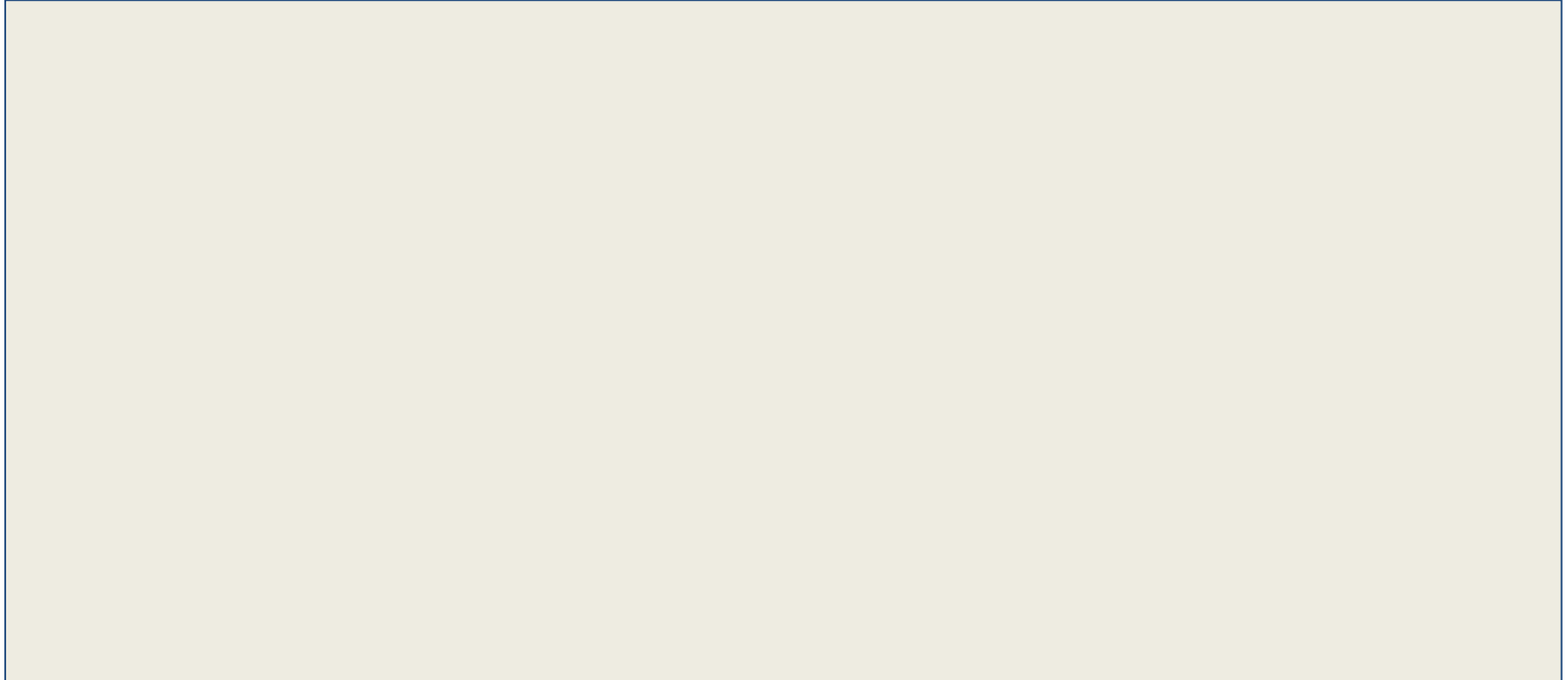
```
/** Histogramm: Zeichnen Sie ein Histogramm:
 */
import java.util.Random;    // random number generator

class Histogram {
    public static void main( String[] str ) {
        int[] liste;  liste = new int[25];
        Random rand = new Random();
        // Fuelle liste mit beliebige Integer in [0,24]:
        for ( byte i = 0; i < liste.length; i++ ) {
            liste[i] = rand.nextInt( liste.length );
        }
        // Fuelle erst ein 2D-Array von Booleans
        // Zeichne dann ein Histogramm (pro-Tipp: benutze "\u2589"-Block)
    }
}
```

Beispiel



Beispiel



5.2. Strings

- In Java ist ein String (Zeichenkette) eine Folge von (Unicode-)Zeichen
Beispiel:

```
String motto = "Wir lernen Java!"; // automatisches new
```

- **motto** ist eine Referenzvariable
 - speichert nicht den String, sondern nur die Referenz darauf
- jedes Zeichen hat eine Position (gezählt ab 0):



5.2. Strings

- Nach einer neuen Zuweisung an die Referenzvariable, z.B.

```
motto = "Carpe Diem";
```

ist der String "Wir lernen Java!" nicht mehr zugreifbar und wird vom Garbage Collector gelöscht

- Die folgenden Zuweisungen sind unterschiedlich:

```
motto = null; // motto zeigt auf keinen String mehr
```

```
motto = ""; // motto zeigt auf den leeren String
```

5.2. Strings

Operationen auf Strings

- Zusammenfügen (Konkatenation):

```
String s = "Zahl 1" + "2";           // "Zahl12"  
s += " ist gleich 3";                // "Zahl12 ist gleich 3"  
s = "elf ist " + 1 + 1;              // "elf ist 11"  
s = 1 + 1 + " ist zwei";             // "2 ist zwei"  
s = 0.5 + 0.5 + " ist " + 1;         // "1.0 ist 1"
```

- Der Operator + ist überladen:

- **int + int**: Addition
- **String + String**: Konkatenation
- **String + int** und **int + String**: Umwandlung der Zahl in einen String und anschließende Konkatenation
- analog für andere Datentypen

5.2. Strings

Operationen auf Strings

- Vergleichsoperatoren (nur == und !=):
 - der Operator == liefert true, wenn beide Operanden auf denselben String verweisen (Objektidentität)
- Vergleich:
 - `s1.equals(s2)` liefert true, wenn `s1` und `s2` zeichenweise übereinstimmen
 - `s1.compareTo(s2)`
 - < 0, wenn `s1` alphabetisch vor `s2`
 - = 0, wenn `s1` und `s2` zeichenweise übereinstimmen
 - > 0, wenn `s1` alphabetisch nach `s2`
- Länge: `motto.length()`

5.2. Strings

Weitere Methoden



- Vergleich mit Anfang und Ende:

```
boolean a = motto.startsWith("Wir");    // true
boolean b = motto.endsWith(".");        // false
```

- Zugriff auf einzelne Zeichen:

```
char c = motto.charAt(5);               // 'e'
```

- Suche nach Zeichen:

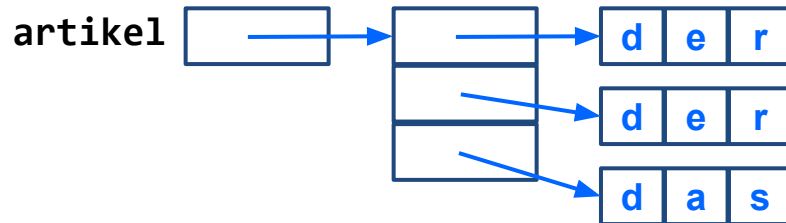
```
int i = motto.indexOf('e', 0);           // ab Index 0: 5
int j = motto.lastIndexOf('e', 15);      // ab Index 15: 8
```

- Ausschneiden / Ersetzen:

```
String s = motto.substring(11, 15);      // "Java!"
String t = motto.replace('a', 'A');      // "Wir lernen JAvA!"
```

5.3. Strings und Arrays

- String in Array von char umwandeln:
`char[] textArray = motto.toCharArray();`
- Array von char in String umwandeln:
`String str = new String(textArray);`
- Array von Strings:
`String[] artikel = { "der", "die", "das" };`



5.3. Strings und Arrays

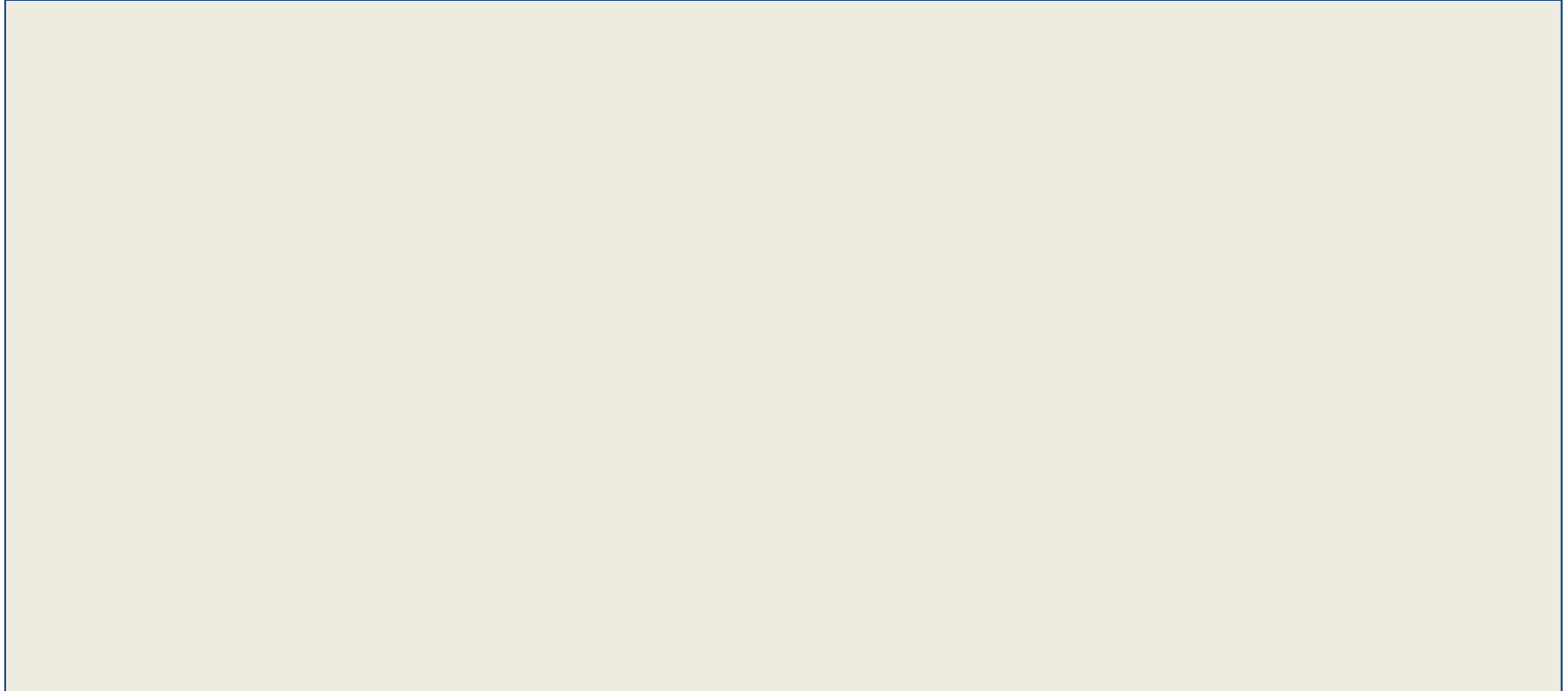
- Strings können nicht verändert werden ("**immutable**")
 - die Operation `+` oder z.B. die Methode `replace` erzeugen jeweils einen neuen String
 - dadurch verhalten sich Strings wie einfache Datentypen
 - z.B. *call-by-value-Semantik bei Parameter übergabe*
- Es gibt auch eine Klasse `StringBuffer`, deren Objekte auch verändert werden können
 - Geschwindigkeitsvorteil, wenn viele Manipulationen an Strings vorgenommen werden
- Dokumentation der Klasse `String` im WWW unter
<https://docs.oracle.com/en/java/javase/20/docs/api/java.base/java/lang/String.html>

Beispiel (🌶️🌶️🌶️)

```
/** SimpleWordle: Implementieren Sie ein einfaches Wordle-Spiel mit einer
    unendlichen Anzahl von Versuchen. Verwenden Sie diese Vorlage:
 */
import java.util.Random;    // random number generator
import java.util.Scanner;   // scan strings in der Konsole
class SimpleWordle {
    public static String dialog( String hint ) { // Zeige Hinweis, Eingabe
        System.out.print( hint + ", what is your guess? ");
        Scanner scan = new Scanner(System.in);
        return scan.next();
    }
    public static void main( String[] str ) {
        String loesung = "weary"; // Loesungswort
        String hint = "-----"; // Hinweis mit gefundenen Buchstaben
        // Implementiere ein Schleife, die dialog() ausführt & aufhört
        // wenn das Wort geraten wurde:
    }
}
```

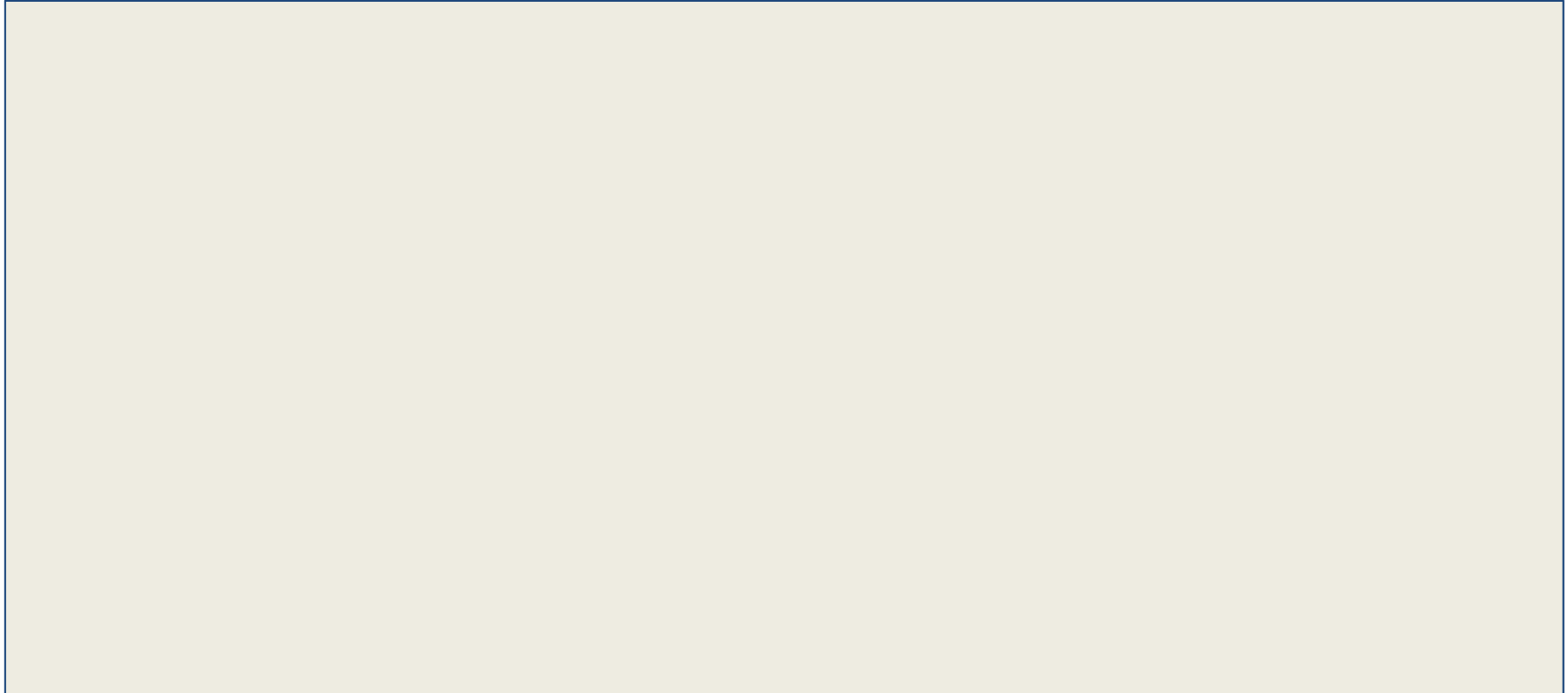

5. Arrays und Strings

Beispiel (🌶️🌶️🌶️)



5. Arrays und Strings

Beispiel (🌶️🌶️🌶️)



Objektorientierte und Formale Programmierung

-- Java Grundlagen --

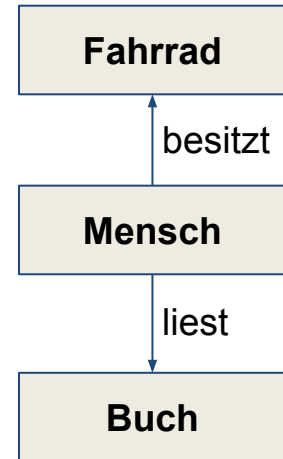
6. Objekte und Methoden

Prinzipien der Objektorientierung

Abstraktion der Wirklichkeit:

wird als Menge **interagierender Objekte** modelliert

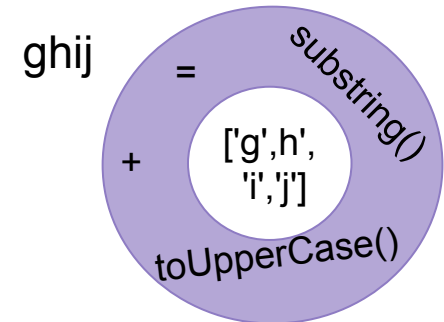
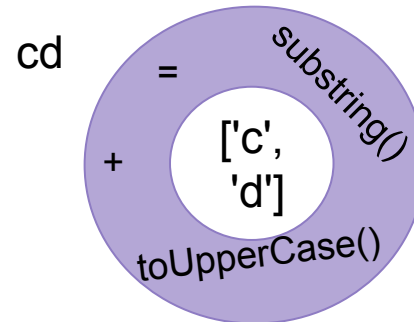
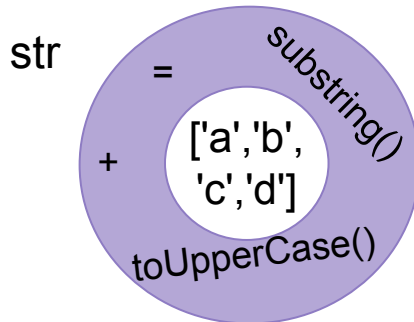
- Ein Objekt
 - hat bestimmte **Eigenschaften (Attribute)** / einen **Zustand**
 - reagiert mit einem bestimmten **Verhalten** (beschrieben durch eine Menge von **Operationen / Methoden**) auf die Umgebung
 - steht in bestimmten **Beziehungen** zu anderen Objekten
- Objekte werden zu **Klassen** zusammengefaßt
 - Abstraktion von der konkreten Ausprägung eines Objekts
 - Objekte mit gleichen Attributen und gleichem Verhalten werden gemeinsam betrachtet
 - Klasse ist Stellvertreter / Bauplan für diese Objekte



Beispiel: String Klasse

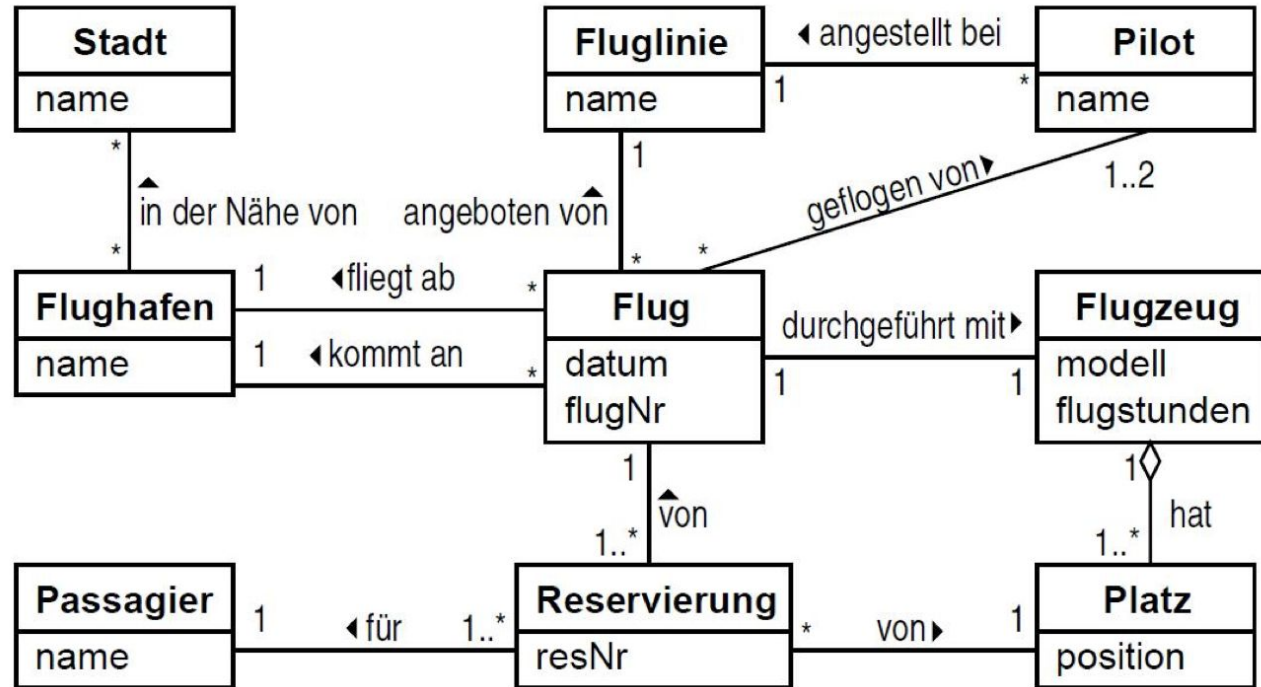
```
class StringBeispiel {  
    public void beispiel( String ghij ) {  
        String str = "abcd";           // str ist Objekt von Klasse String: "abcd"  
        String cd = str.substring(2, 4); // cd ist Objekt von Klasse String: "cd"  
        System.out.println(str.toUpperCase()+"ef"); // Objekt von Klasse String "abcdef"  
    }  
}  
// ...  
StringBeispiel bsp = new StringBeispiel(); bsp.beispiel("ghij");
```

Objekte:



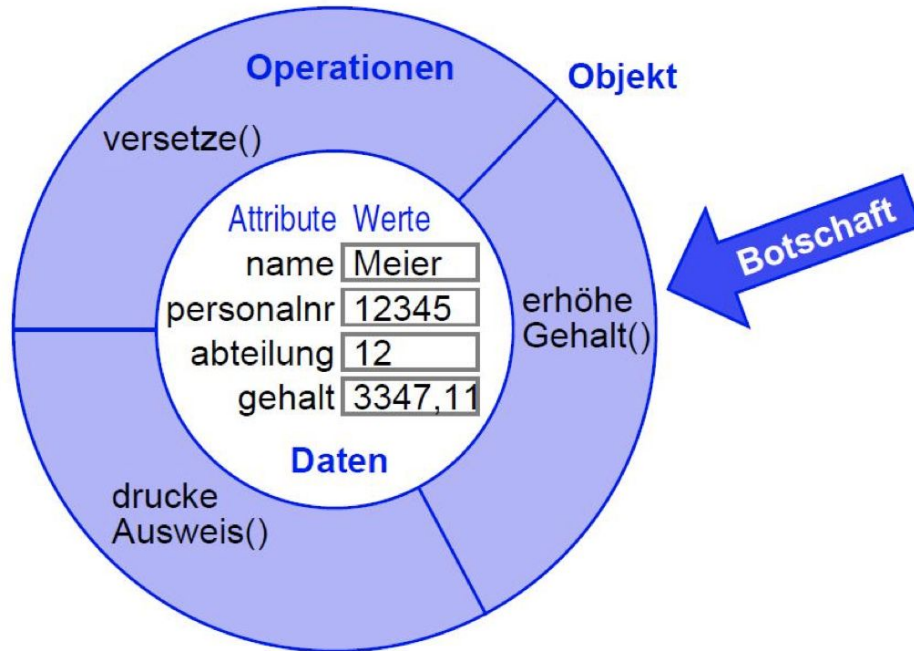
Prinzipien der Objektorientierung

- Beispiel UML
Klassendiagramm:
(wird später
eingeführt)



Prinzipien der Objektorientierung

Objekt und Geheimnisprinzip:

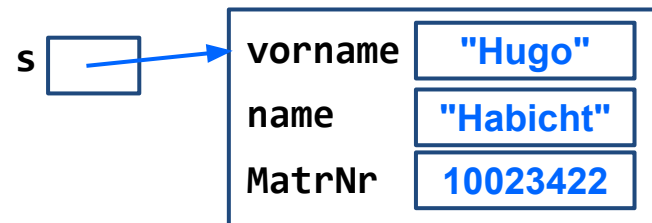


- Ein wesentliches Konzept von objektorientierter Programmierung ist, dass der innere Aufbau eines Objekts für andere Objekte zum großen Teil unzugänglich ist. Viele Teile eines Objekts bleiben geheim, befinden sich sozusagen in einer geschützten Kapsel. Man nennt das Geheimnisprinzip auch **Kapselung**.

6.1. Erzeugung von Objekten

- Objekte werden in Java nicht direkt in Variablen gespeichert (wie auch für Arrays)
- In Variablen werden nur Referenzen auf solche Speicherbereiche gespeichert:

```
class Student {  
    String vorname, name;  
    long matrNr;  
    ...  
}  
...  
Student s = new Student("Hugo", "Habicht", 10023422);
```



- Erzeugung eines Objekts: Der Speicherbereich für die Daten wird dynamisch durch den Operator `new` angelegt

6.1. Erzeugung von Objekten

Konstruktoren

- Eine Klasse kann spezielle Operationen, *Konstruktoren*, definieren, die bei der Erzeugung eines Objekts ausgeführt werden
- Typische Aufgaben eines Konstruktors:
 - Initialisierung der Attributwerte des neuen Objekts
 - ggf. Erzeugung existenzabhängiger Teil-Objekte
- Ein Konstruktor hat immer denselben Namen wie die Klasse
 - er kann Parameter besitzen, hat aber keinen Ergebnistyp (nicht einmal **void**)
 - Konstruktoren können auch überladen werden (siehe nächste Folie)
- Definiert eine Klasse keinen Konstruktor, wird automatisch ein parameterloser Konstruktor erzeugt
 - Attribute werden mit Standardwerten (**0** bzw. **null**) initialisiert

6.1. Erzeugung von Objekten

Konstruktoren

```
class Kugel {  
    // Klassenattribute:  
    final static double PI = 3.14159265;  
    static int anzahl = 0; // zählt erzeugte Kugeln  
  
    // Attributes:  
    double xMitte, yMitte, zMitte, radius;  
  
    // Parameterloser Konstruktor (Default-Konstruktor):  
    Kugel() {  
        radius = 1;           // setze Radius des neuen Objekts,  
                               // andere Attribute sind mit 0 initialisiert  
        anzahl++; // Klassenvariable erhöhen  
    }  
}
```

6.1. Erzeugung von Objekten

Konstruktoren

```
Kugel(double x, double y, double z) {  
    this();           // ruft Konstruktor Kugel() auf  
    xMitte = x;       // Aufruf von this() muss die  
    yMitte = y;       // allererste Anweisung sein  
    zMitte = z;  
}  
Kugel(double x, double y, double z, double r) {  
    this(x, y, z);    // ruft Kugel(x,y,z) auf  
    radius = r;       // ändert Wert des Radius auf r  
}  
}
```

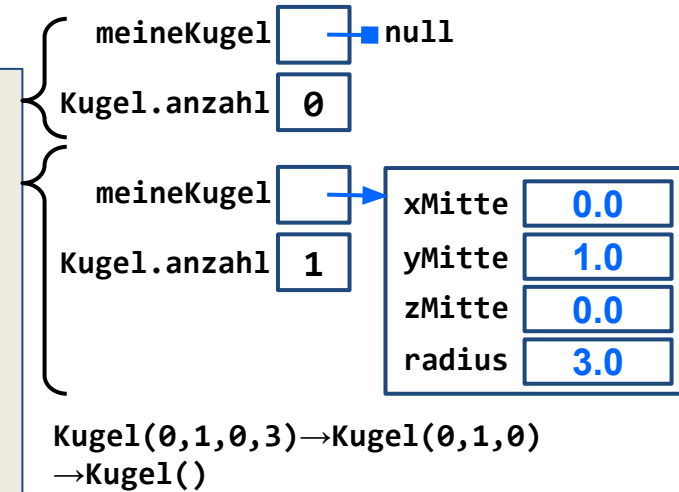
- `this([<Parameterliste>])` als erste Anweisung führt zum Aufruf eines anderen Konstruktors für dasselbe Objekt

6.1. Erzeugung von Objekten

- Der Ausdruck `new <Klassenname> ([<Parameterliste>])` erzeugt (instanziert) ein neues Objekt der angegebenen Klasse
 - Wert des Ausdrucks ist eine Referenz auf das Objekt
- Beispiele: `new Kugel()`
`new Kugel(1, 0, 0)`
- Ablauf bei der Instanziierung:
 - Anlegen des Objekts im Speicher
 - dabei Belegung der Attribute mit Standardwerten
 - Aufruf des Konstruktors für das neue Objekt
 - passend zu Anzahl / Typen der übergebenen Parameter
 - Rückgabe der Referenz auf das Objekt

6.1. Erzeugung von Objekten

```
Kugel meineKugel;  
meineKugel = new Kugel(0, 1, 0, 3);  
...  
Kugel() {  
    radius = 1;  
    anzahl++;  
}  
Kugel(double x, double y, double z) {  
    this();  
    xMitte = x; yMitte = y; zMitte = z;  
}  
Kugel(double x, double y, double z, double r) {  
    this(x, y, z);  
    radius = r;  
}
```



6.1. Erzeugung von Objekten

Weitere Möglichkeiten zur Initialisierung von Attributen

- Gemeinsam mit der Deklaration:

```
static int anzahl = 0;  
double radius = 1.0; // Default-Radius: 1.0
```

- In einem eigenen, speziellen Block der Klassendefinition:

```
class Kugel {  
    ...  
    static {           // wird nur einmal durchlaufen,  
        anzahl = 0;    // wenn die Klasse geladen wird  
    }  
    {                 // wird für jedes erzeugte Objekt  
        radius = 2.0;  // ausgeführt (vor dem Konstruktor)  
    }  
}
```

6.1. Erzeugung von Objekten

Lebensdauer von Objekten

```
{
    Student s;
    {
        Student thomas = new Student("Thomas");
        Student tom = new Student("Tom");
        // 'Tom' und 'Thomas' sind verschiedene Studenten
        tom = thomas;
        // Hier wurde nur die Referenz kopiert
        // tom und thomas verweisen jetzt auf dasselbe Objekt
        // Das Objekt 'Tom' ist nicht mehr zugreifbar
        s = thomas;
    }
    // Das Objekt 'Thomas' existiert noch (s ist eine Referenz auf 'Thomas')
}
```

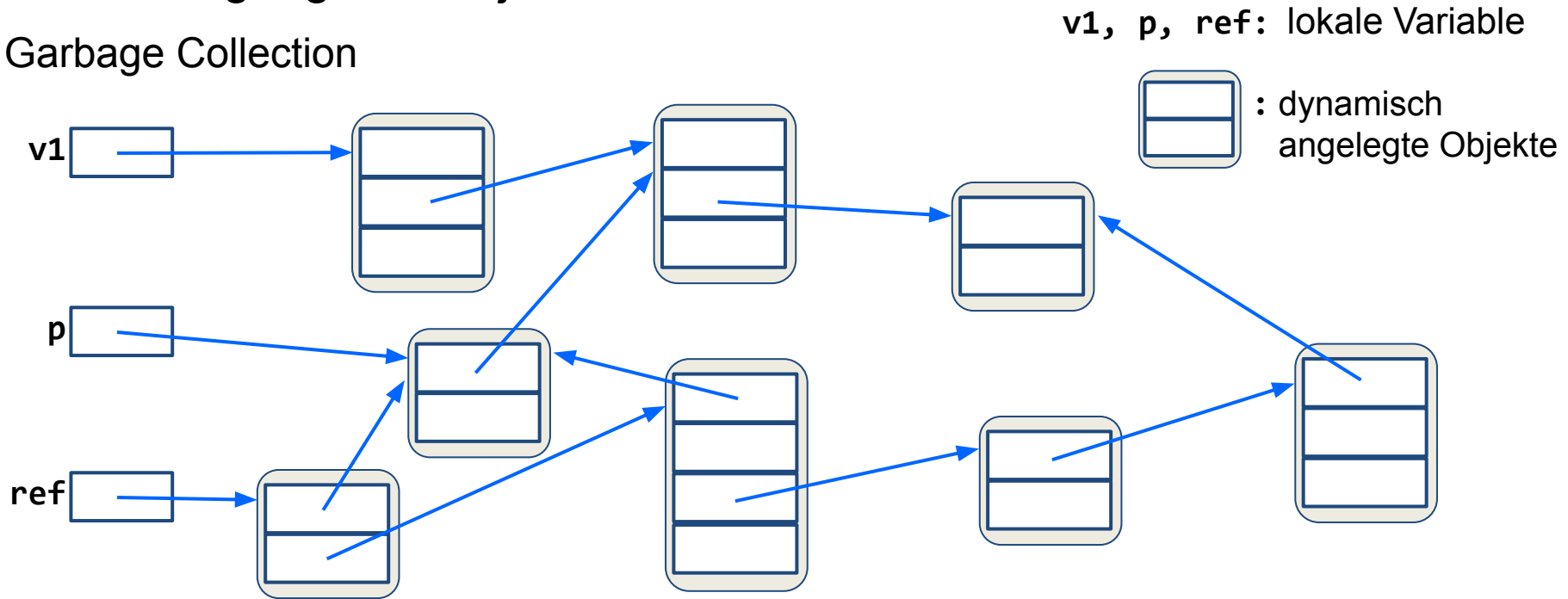
6.1. Erzeugung von Objekten

Freigabe von Speicherbereichen

- In Java existiert ein einmal erzeugtes Objekt weiter, solange es noch eine Möglichkeit gibt auf das Objekt zuzugreifen
 - d.h., solange es noch über eine Kette von Referenzen von einer Variable aus erreicht werden kann
- Ein mit **new** angelegter Speicherbereich (Objekt oder Array) wird von der JVM automatisch wieder freigegeben, wenn
 - Speicherplatz benötigt wird und
 - der Speicherbereich nicht mehr zugreifbar ist
- **Garbage Collection:**
Die Suche nach solchen Speicherbereichen und deren Freigabe

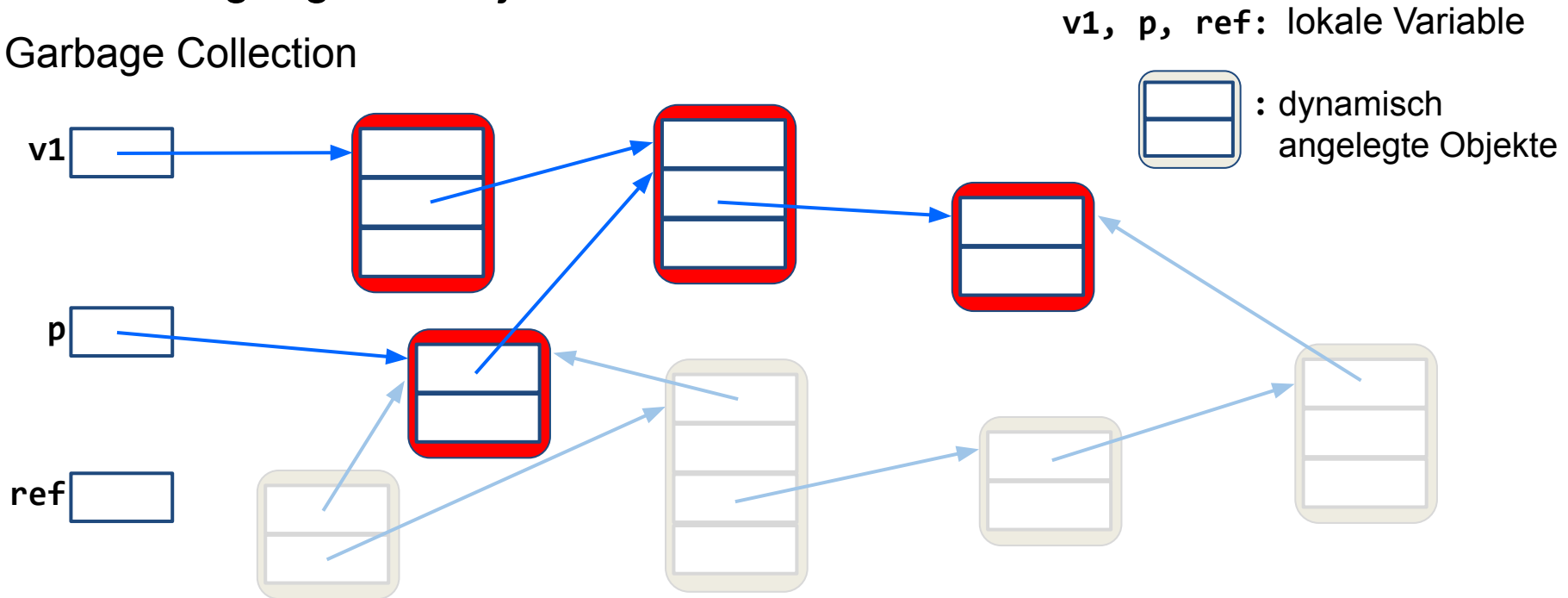
6.1. Erzeugung von Objekten

Garbage Collection



6.1. Erzeugung von Objekten

Garbage Collection



Wenn **ref** ungültig oder mit **null** belegt wird, markiert der Garbage Collector alle erreichbaren Objekte. Die unmarkierten werden gelöscht.

6.2. Zugriff auf Attribute und Methoden

- Die Attribute und Methoden eines Objekts können über eine Objektreferenz angesprochen werden:
 - `<Objektreferenz> . <Attributname>`
 - `<Objektreferenz> . <Methodenname> ([<Parameterliste>])`
 - gilt auch für Klassenattribute / -methoden
 - Voraussetzung: Sichtbarkeit erlaubt den Zugriff
 - Beispiele:
 - `meineKugel.radius` // Attribut
 - `meineKugel.anzahl` // Klassenattribut
 - `meineKugel.volumen()` // Methode
- Klassenattribute/-methoden können unabhängig von einer Objektreferenz auch über den Klassennamen angesprochen werden
 - Beispiel: `Kugel.anzahl`

6.2. Zugriff auf Attribute und Methoden

Namen

- Ein **qualifizierter Name** ist ein Name mit expliziter Angabe des Objekts oder der Klasse
 - z.B.: `meineKugel.radius`, `Kugel.anzahl`
- Alle andere Namen (ohne Punkt) heißen **einfache Namen**
 - z.B.: `radius`, `i`, `anzahl`
- Eine Methode kann auf folgende Methoden über einfache Namen zugreifen:
 - die Klassenmethoden der eigenen Klasse (`static`)
 - die Methoden des eigenen Objekts (nur, wenn die aufrufende Methode keine Klassenmethode ist)

6.2. Zugriff auf Attribute und Methoden

Namen

- Eine Methode greift immer über einfache Namen zu auf:
 - ihre Parameter
 - ihre lokalen Variablen
- Beispiel (neue Methode der Klasse Kugel):

```
double volumen() {  
    return 4.0/3.0 * PI * radius * radius * radius;  
} // PI: Klassenattribut; radius: Attribut
```

6.2. Zugriff auf Attribute und Methoden

Die Referenzvariable `this`

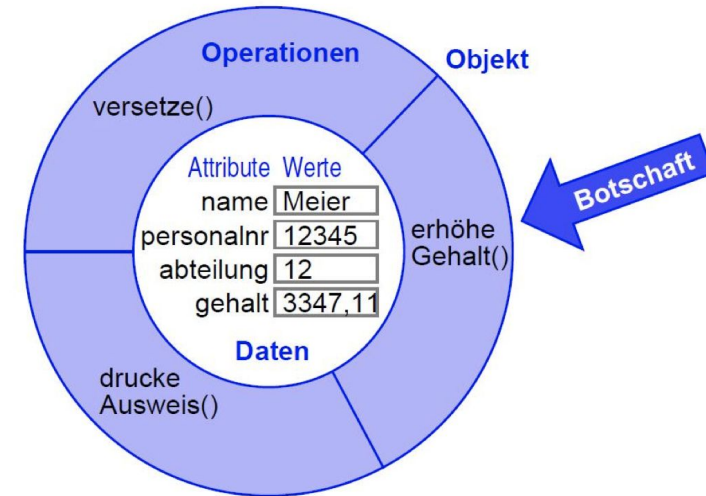
- Existiert in jeder Methode (außer in Klassenmethoden)
- Muss / darf nicht deklariert werden
- Zeigt immer auf das eigene Objekt
 - Vorstellung: `this` ist ein Parameter, in dem der Methode eine Referenz auf das eigene Objekt übergeben wird
- Nützlich z.B. zur Unterscheidung von Attributen und gleichnamigen Parametern:

```
void neuerRadius(double radius)
{
    // radius: Parameter; this.radius: Attribut
    this.radius = radius;
}
```

6.2. Zugriff auf Attribute und Methoden

Sichtbarkeit von Attributen und Methoden

- Java erlaubt die Festlegung, welche Attribute und Operationen einer Klasse "von Außen" sichtbar sein sollen, zur Realisierung von Geheimnisprinzip und Datenkapselung
- Wir unterscheiden drei Sichtbarkeiten:
 - **public** (öffentlich): sichtbar für alle Klassen
 - auf Attribut kann von allen Klassen aus zugegriffen werden
 - Operation kann von allen Klassen aufgerufen werden
 - **private** (privat): sichtbar nur innerhalb der Klasse
 - kein Zugriff/Aufruf durch andere Klassen möglich
 - **protected** (geschützt): sichtbar nur innerhalb der Klasse und ihren Unterklassen



6.2. Zugriff auf Attribute und Methoden

Sichtbarkeit von Attributen und Methoden: Get- und Set-Methoden

- In der Implementierung einer Klasse sollten Attribute immer `private` sein.
Wahrung des Geheimnisprinzips: kein direkter Zugriff
- Konvention: Zugriff auf Attribute von außen nur über Get- und Set-Methoden:

```
private String name;           // Attribut
public String getName();       // Get-Methode
public void setName(String aName); // Set-Methode
```

- Vorteil: Kapselung der Zugriffe
 - feste Schnittstelle nach außen, unabhängig von konkreter Speicherung bzw. Darstellung der Daten
 - Get- und Set-Methoden können Prüfungen vornehmen

6.2. Zugriff auf Attribute und Methoden

Beispiel

```
class Klasse {  
    Klasse1 o1; // Klasse hat ein Objekt von Klasse1  
    Klasse2 o2; // Klasse hat ein Objekt von Klasse2  
    Klasse() {  
        // Erzeuge o1 und o2 als Attribute in Klasse, und  
        // verlinke o2 als Attribut in o1, so dass o2.a1 == o1.k2.a1:  
  
    }  
    public static void main(String[] s) { // main ist eine Klassenmethode  
        // Erzeuge ein Objekt von Klasse:  
  
    }  
}
```

6.2. Zugriff auf Attribute und Methoden

Beispiel

```
class Klasse2 {  
    public double a1 = 0.0;  
    Klasse2(double a1) {  
        this.a1 = a1;  
        System.out.println("K2");  
    }  
}  
  
class Klasse1 {  
    private String a1 = "hi";  
    public Klasse2 k2 = null;    // Klasse1 hat ein Objekt von Klasse2  
    Klasse1(Klasse2 k2) {  
        this.k2 = k2;  
        System.out.println("K1");  
    }  
}
```

6.2. Zugriff auf Attribute und Methoden

Beispiel: Lösung

```
class Klasse {
    Klasse1 o1; // Klasse hat ein Objekt von Klasse1
    Klasse2 o2; // Klasse hat ein Objekt von Klasse2
    Klasse() {
        // Erzeuge o1 und o2 als Attribute in Klasse, und
        // verlinke o2 als Attribut in o1, so dass o2.a1 == o1.k2.a1:
        o2 = new Klasse2( 2.3 );
        o1 = new Klasse1( o2 );
        System.out.println( o1.k2.a1 );
    }
    public static void main(String[] s) { // main ist eine Klassenmethode
        // Erzeuge ein Objekt von Klasse:
        Klasse k = new Klasse();
    }
}
```

6.2. Zugriff auf Attribute und Methoden

Klassenattributen und Klassenmethoden: `static`

- Im Unterschied zur Klassenmethode muss eine Objektmethode an einem konkret instanziierten Objekt aufgerufen werden, bei Klassenmethoden *nicht*
- Werden statische Variablen von einem Objekt verändert, ist diese Veränderung auch in allen anderen Objekten sichtbar
- z.B. für:
 - Variablen die zur Klasse gehören: `Kugel.anzahl`
 - Konstanten: `public static int MIN_VALUE;` , `Kugel.PI`
 - Hilfsmethoden: `static double parseDate(String s)`
 - Klassenverwaltung: `static int zaehleKugel() { return anzahl; }`

6.3. Aufruf von Methoden

- Syntax: `<Name> ([<Parameterliste>])`
`<Parameterliste> ::= <Ausdruck> {, <Ausdruck>}`
- Name ist ggf. ein qualifizierter Name

```
class Kugel {  
    ...  
    private double kubus(double x) { // x: formaler Parameter  
        return x * x * x;  
    }  
    public double volumen() {  
        return 4.0/3.0 * PI * kubus(radius); // kubus(radius):Methodenaufruf  
                                              // radius: aktueller Parameter (Argument)  
    }  
}
```

6.3. Aufruf von Methoden

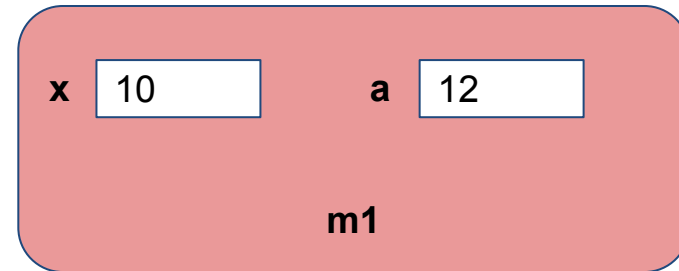
Ablauf eines Methodenaufrufs

1. Die Ausdrücke in der Parameterliste werden ausgewertet
2. **call by value**: Die Werte der aktuellen Parameter werden an die formalen Parameter der Methode zugewiesen
3. Der Methodenrumpf wird ausgeführt
 - bis zum Ende (nur bei **void**-Methoden erlaubt)
 - oder bis zur Anweisung **return** [**<Ausdruck>**]
 - der Wert von **<Ausdruck>** bestimmt ggf. den Wert des Methodenaufrufs (der selbst ein Ausdruck ist)
4. Die Abarbeitung der aufrufenden Methode wird nach dem Methodenaufruf fortgesetzt

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```



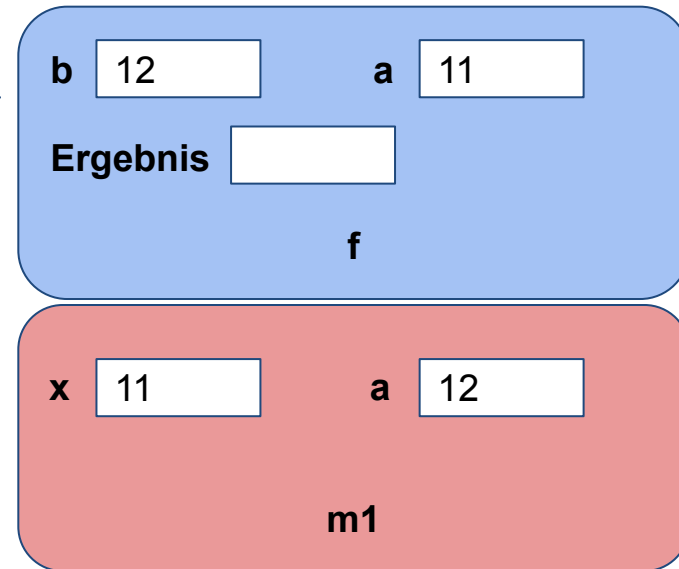
Stapel (Keller, Stack) von Aktivierungsrahmen,
die lokale Variable der Methoden speichern



6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```

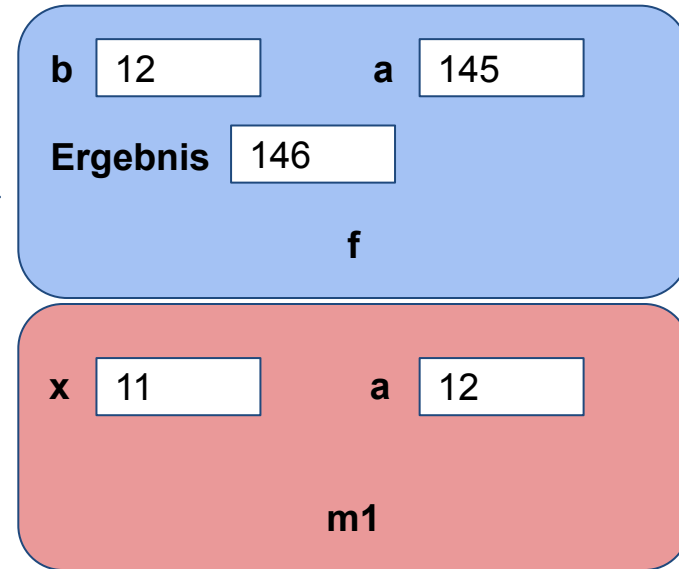


Stapel (Keller, Stack) von Aktivierungsrahmen,
die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```

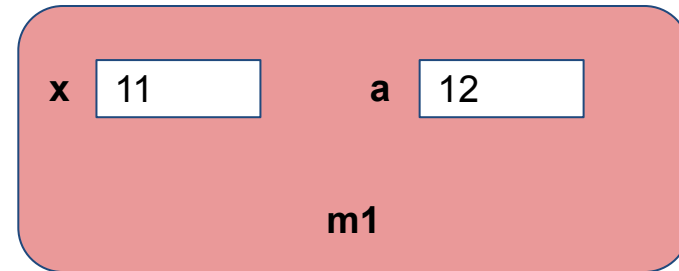


**Stapel (Keller, Stack) von Aktivierungsrahmen,
die lokale Variable der Methoden speichern**

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```

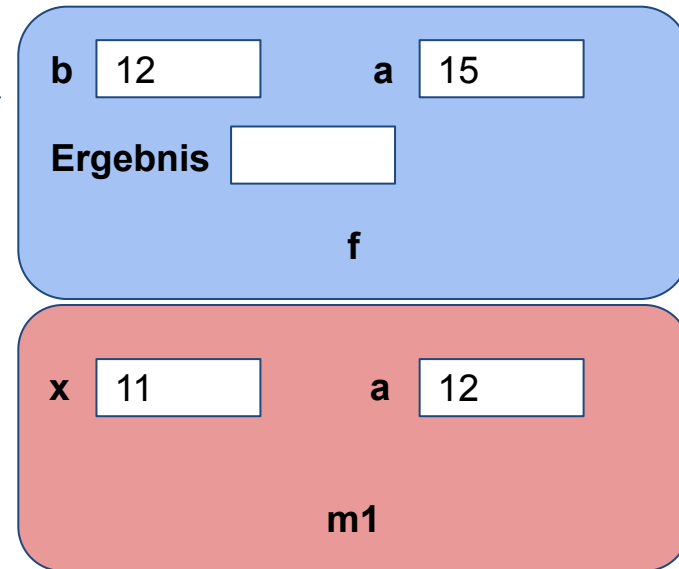


Stapel (Keller, Stack) von Aktivierungsrahmen,
die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```

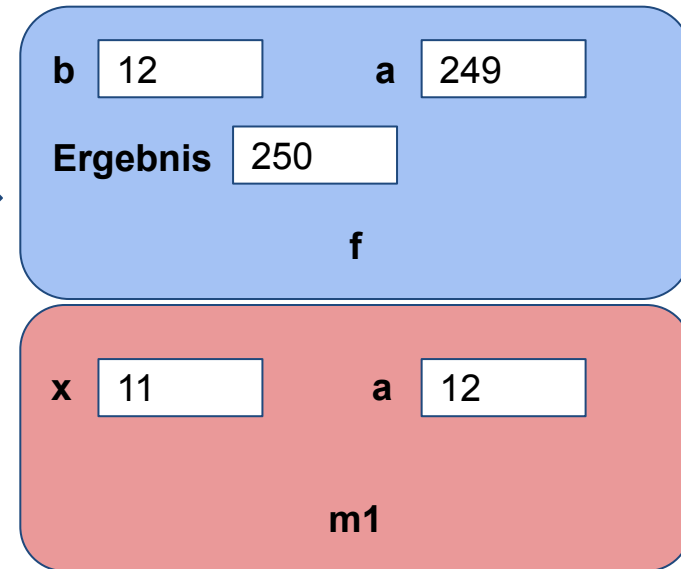


Stapel (Keller, Stack) von Aktivierungsrahmen,
die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```

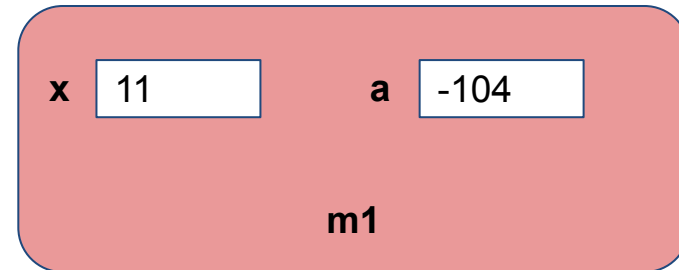


**Stapel (Keller, Stack) von Aktivierungsrahmen,
die lokale Variable der Methoden speichern**

6.3. Aufruf von Methoden

Ablauf eines Methodenaufrufs

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```



Stapel (Keller, Stack) von Aktivierungsrahmen,
die lokale Variable der Methoden speichern

6.3. Aufruf von Methoden

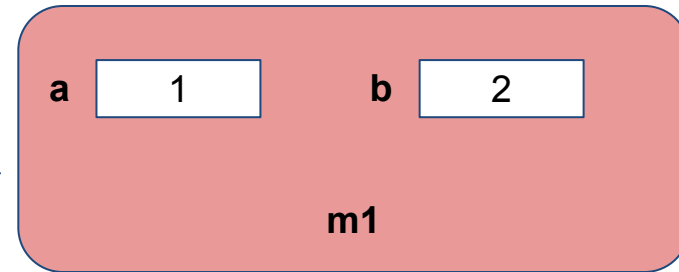
Referenzen als Parameter

- Auch Objekte können Parameter von Methoden sein
 - Übergeben werden dabei jedoch nicht die Objekte, sondern nur Referenzen auf diese Objekte: **call by reference**
 - Übergabe der Objektreferenz mittels „call by value“ entspricht der Semantik von „call by reference“
- Die aufgerufene Methode kann dabei die referenzierten Objekte verändern
 - so lassen sich auch Ein-/Ausgabe-Parameter realisieren
 - unerwartetes Verändern übergebener Objekte sollte aber vermieden werden
- Analog können Objekt-Referenzen auch als Ergebnis einer Methode auftreten

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {  
    void swap(int x, int y) {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    void m1() {  
        int a, b;  
        a = 1; b = 2;  
        swap(a, b); // a = 1, b = 2  
    }  
}
```



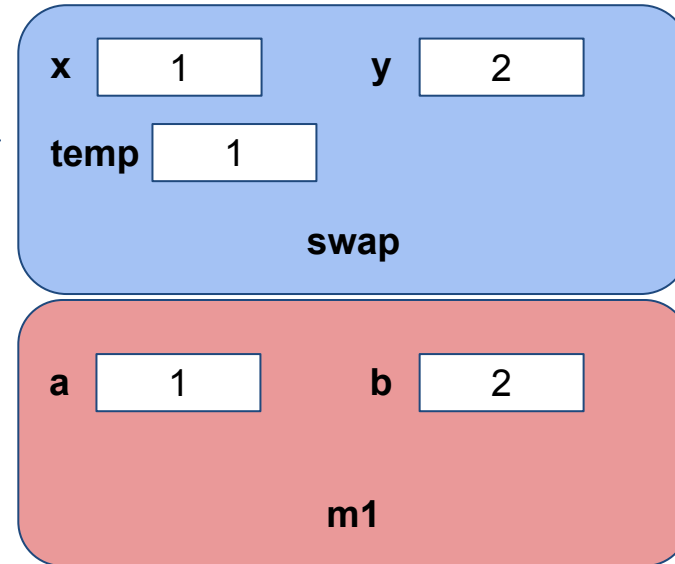
Stapel (Keller, Stack)

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {  
    void swap(int x, int y) {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    void m1() {  
        int a, b;  
        a = 1; b = 2;  
        swap(a, b); // a = 1, b = 2  
    }  
}
```

„**call by value**“: Die Werte der aktuellen Parameter werden an die formalen Parameter der Methode zugewiesen:

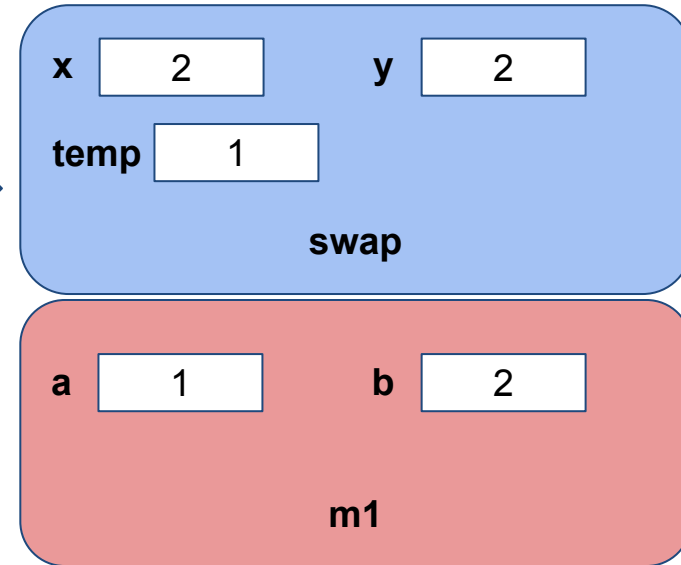


Stapel (Keller, Stack)

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {  
    void swap(int x, int y) {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    void m1() {  
        int a, b;  
        a = 1; b = 2;  
        swap(a, b); // a = 1, b = 2  
    }  
}
```

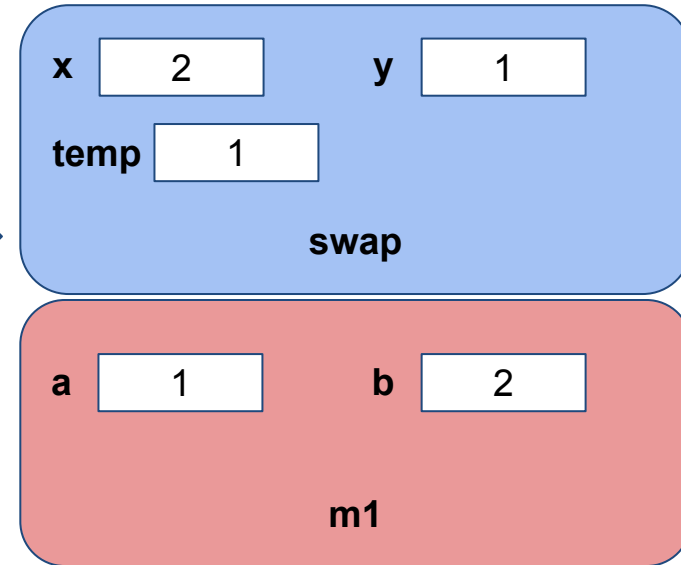


Stapel (Keller, Stack)

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {  
    void swap(int x, int y) {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    void m1() {  
        int a, b;  
        a = 1; b = 2;  
        swap(a, b); // a = 1, b = 2  
    }  
}
```

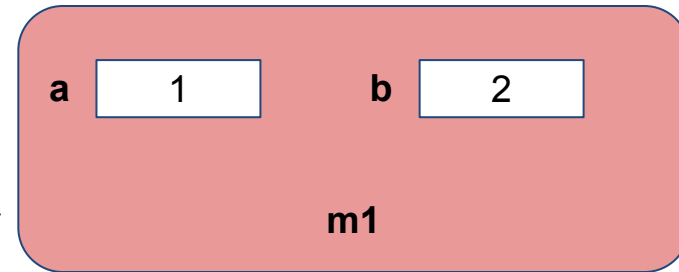


Stapel (Keller, Stack)

6.3. Aufruf von Methoden

Referenzen als Parameter

```
class Example1 {  
    void swap(int x, int y) {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    void m1() {  
        int a, b;  
        a = 1; b = 2;  
        swap(a, b); // a = 1, b = 2  
    }  
}
```



Stapel (Keller, Stack)

6.3. Aufruf von Methoden

Referenzen als Parameter

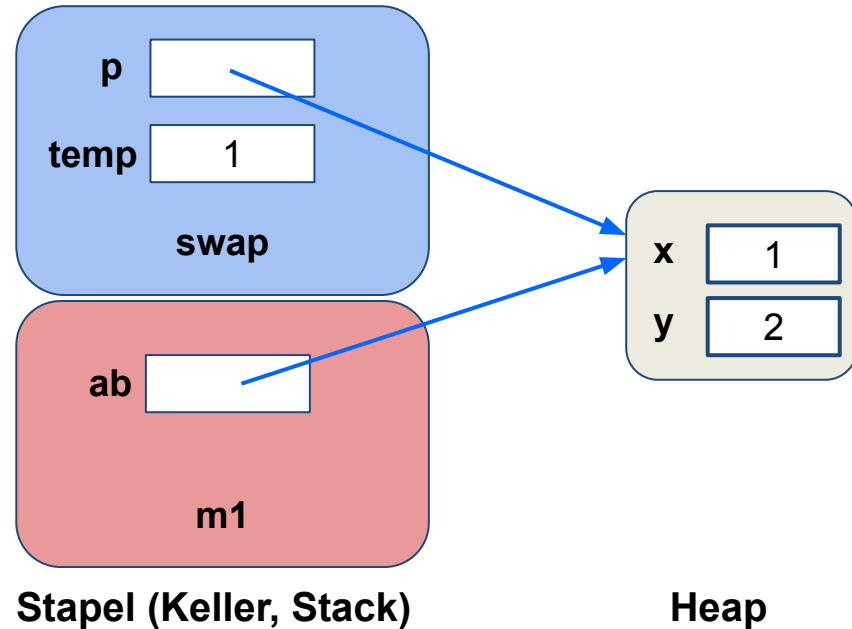
```
class Example1 {  
    void swap(int x, int y) {  
        int temp = x;  
        x = y;  
        y = temp;  
    }  
    void m1() {  
        int a, b;  
        a = 1; b = 2;  
        swap(a, b); // a = 1, b = 2  
    }  
}
```

```
class Pair {  
    public int x, y;  
}  
class Example2 {  
    void swap(Pair p) {  
        int temp = p.x;  
        p.x = p.y;  
        p.y = temp;  
    }  
    void m1() {  
        Pair ab = new Pair();  
        ab.x = 1; ab.y = 2;  
        swap(ab); // ab.x=2, ab.y=1  
    }  
}
```

6.3. Aufruf von Methoden

Referenzen als Parameter

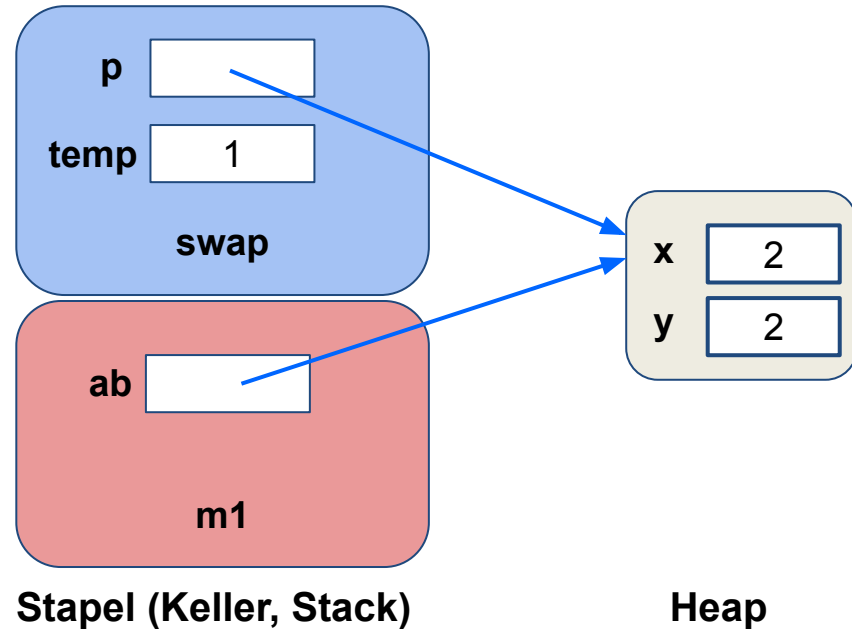
```
void swap(Pair p) {  
    int temp = p.x;  
    p.x = p.y;  
    p.y = temp;  
}  
  
void m1() {  
    Pair ab = new Pair();  
    ab.x = 1; ab.y = 2;  
    swap(ab); // ab.x=2, ab.y=1  
}
```



6.3. Aufruf von Methoden

Referenzen als Parameter

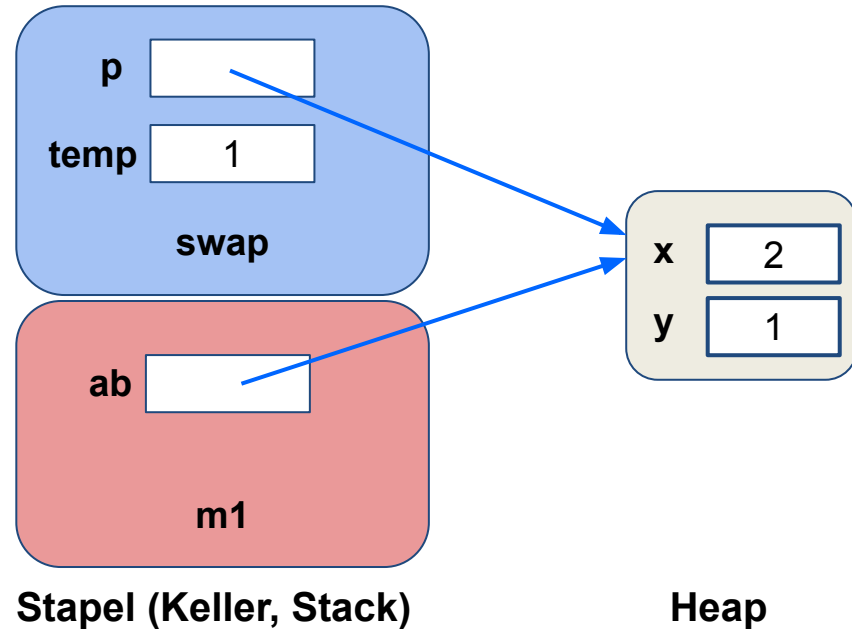
```
void swap(Pair p) {  
    int temp = p.x;  
    p.x = p.y;  
    p.y = temp;  
}  
  
void m1() {  
    Pair ab = new Pair();  
    ab.x = 1; ab.y = 2;  
    swap(ab); // ab.x=2, ab.y=1  
}
```



6.3. Aufruf von Methoden

Referenzen als Parameter

```
void swap(Pair p) {  
    int temp = p.x;  
    p.x = p.y;  
    p.y = temp;  
}  
  
void m1() {  
    Pair ab = new Pair();  
    ab.x = 1; ab.y = 2;  
    swap(ab); // ab.x=2, ab.y=1  
}
```



6.3. Aufruf von Methoden - Beispiel

```
/** WordleClass: Implementieren Sie das einfache Wordle-Spiel jetzt als Klasse:
 */
import java.util.Scanner; // scan strings in der Konsole

class WordleClass {
    /*...*/ // Hier kommen die Attribute

    private String dialog( String hint ) { /*...*/ } // private Methode
    public void run() { /*...*/ } // public Methode

    public static void main( String[] str ) { // main ist eine Klassenmethode
        WordleClass w = new WordleClass();
        w.run();
    }
}
```




6.3. Aufruf von Methoden

Überladen von Methoden

Bei überladenen Methoden wird die zu Anzahl und Typen der aktuellen Parameter passende Methode bereits vom Compiler ausgewählt

```
class Drucker {  
    static void drucke(int Zahl) {  
        System.out.println(Zahl);  
    }  
    static void drucke(String Name) {  
        System.out.println(Name);  
    }  
}  
// ...  
Drucker.drucke(10);
```

Unzulässig:

```
class KursVerwaltung {  
    // Suche Kurs, Ergebnis: Kursnr  
    int suche(String stichwort) {  
        // ...  
    }  
    // Suche Kurs, Ergebnis: Titel  
    String suche(String stichwort) {  
        // ...  
    }  
}
```

6.3. Aufruf von Methoden

Rekursion

- Eine Methode kann sich auch selbst aufrufen
- Beispiel: Fakultätsfunktion

```
static long fak(long n) {  
    if (n == 0)  
        return 1L;  
    else if (n > 0)  
        return n * fak(n-1);  
}
```

Mathematische Definition:

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n - 1)! & \text{für } n > 0 \end{cases}$$

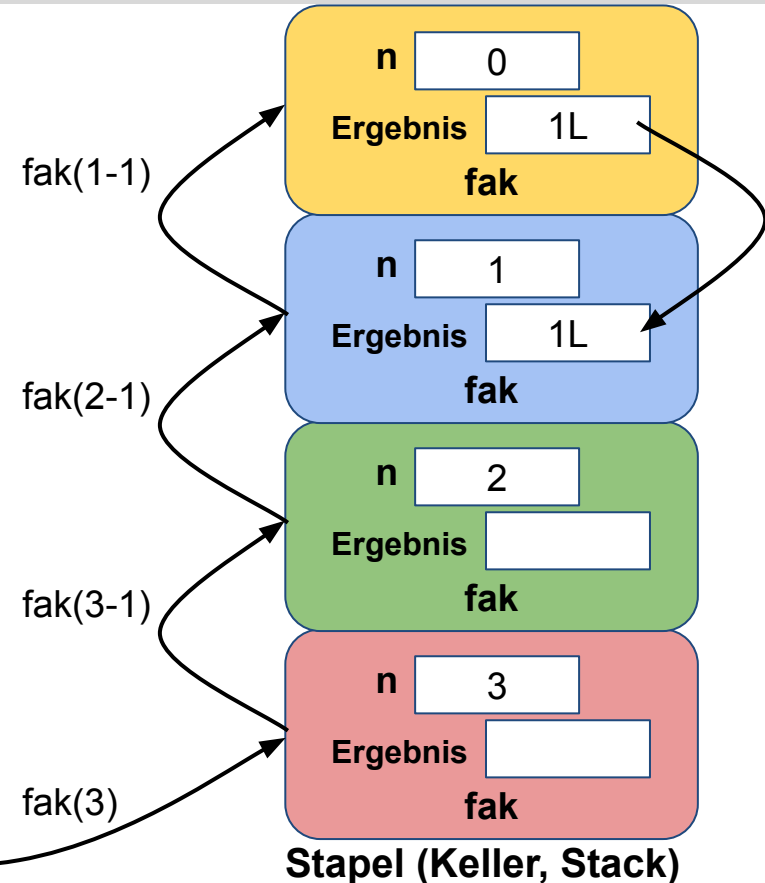
- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs- rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

6.3. Aufruf von Methoden

Rekursion

- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs-rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

```
static long fak(long n) {  
    if (n == 0)  
        return 1L;  
    else if (n > 0)  
        return n * fak(n-1);  
}  
long f = fak(3);
```

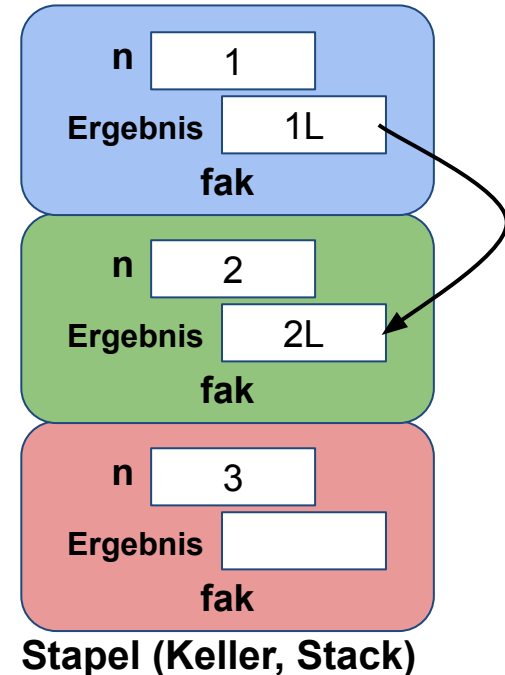


6.3. Aufruf von Methoden

Rekursion

- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs-rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

```
static long fak(long n) {  
    if (n == 0)  
        return 1L;  
    else if (n > 0)  
        return n * fak(n-1);  
}  
long f = fak(3);
```

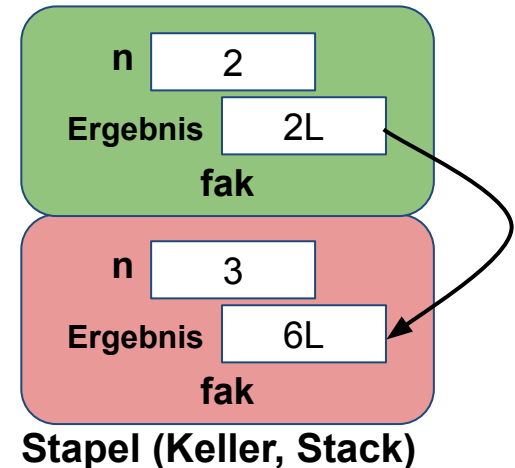


6.3. Aufruf von Methoden

Rekursion

- Jede Aktivierung einer Methode bekommt einen eigenen Aktivierungs-rahmen auf dem Stapel, damit hat jede Aktivierung ihre eigenen lokalen Variablen

```
static long fak(long n) {  
    if (n == 0)  
        return 1L;  
    else if (n > 0)  
        return n * fak(n-1);  
}  
long f = fak(3);
```



6.4. Arrays von Objekten

- Wie in einfachen Variablen werden in Arrays nur Objektreferenzen gespeichert
- Beispiel: Array mit 2 Studenten

```
Student[] a1 = new Student[2]; // Initialisierung mit null  
a1[0] = new Student("Hans"); // Initialisierung der Elemente  
a1[1] = new Student("Fritz");
```

- Kürzer:

```
// Deklaration mit Initialisierung der Elemente:  
Student[] a2 = { new Student("Hans"), new Student("Fritz") };
```



6.4. Arrays von Objekten - Beispiel: SnailGame v1 ([github](#))

```
class Game {
    private Snail player;
    private Snail[] enemy;
    private boolean bang = false;

    public Game() { /* ... */ }
    public void keyPressed(KeyEvent e) {...}
    public void keyReleased(KeyEvent e) {}
    public void keyTyped(KeyEvent e) {}
    public void paintComponent( Graphics g ) {...}
    public static void main(String[] a) {...}
}
```

```
class Snail {
    private short len = 7;
    private int[] xPos = null;
    private int[] yPos = null;

    Snail(short len) { /* ... */ }
    Snail(short len, int x, int y) {...}
    boolean isOverlapped( Snail snail ) {...}
    void moveBody() { /* ... */ }
    void moveHead(int x, int y) { /* ... */ }
    int getLength() { /* ... */ }
    int getPosX(int i) { /* ... */ }
    int getPosY(int i) { /* ... */ }
}
```

6.5. Einführung in UML

- UML = Unified Modelling Language, Entwicklung seit 1994
- standardisierte (graphische) Sprache zur objektorientierten Modellierung von (Software-)Systemen
- UML definiert eine Vielzahl von Diagrammtypen: unterschiedliche Sichtweisen des modellierten Systems
 - statische vs. dynamische Aspekte
 - unterschiedlicher Abstraktionsgrad
- UML unterstützt sowohl Objekt-Orientierte Analyse (OOA) als auch Objekt-Orientiertes Design (OOD)

6.5. Einführung in UML: Klassen, Wiederholung aus Java:

- Eine Klasse definiert für eine Kollektion gleichartiger Objekte
 - deren Struktur, d.h. die **Attribute** (nicht die Werte)
 - das Verhalten, d.h. die **Operationen**
 - die möglichen Beziehungen (**Assoziationen**) zu anderen Objekten, einschließlich der Generalisierungs-Beziehung
 - Eine Klasse besitzt einen Mechanismus, um neue Objekte zu erzeugen
- Die Klasse ist der "Bauplan" für diese Objekte

6.5. Einführung in UML: Klassen

- Anforderungserhebung, Analyse

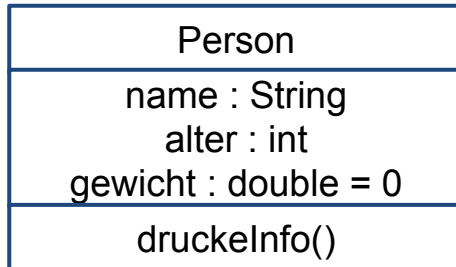
Entwickler: „Was ist euch wichtig?“

Anwender: „Der Kunde.“

Entwickler: „Was ist denn ein Kunde, welche Merkmale sind für euch relevant?“

Anwender: „Der Kunde hat einen Namen, eine Anschrift und eine Bonität, die wir überprüfen.“

- Design (Klasse in UML):



- Implementierung (Klasse in Java):

```
class Person {
    String name;
    int alter;
    double gewicht = 0;
    public void druckeInfo() { /* ... */ }
}
```

6.5. Einführung in UML: Objekte

• Allgemeines Schema:

<Objektname>:<Klassenname>
<Attributname 1> = <Wert 1> ... <Attributname n> = <Wert n>

• z.B.:

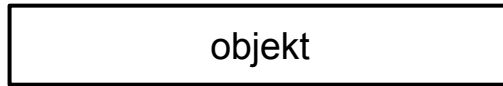
myVideo : Video
author = "R.A." length = 3.32 ytlink = "dQw4w9WgXcQ"

- Objekt- und Attributnamen klein geschrieben, Klassennamen groß geschrieben
- Die Operationen werden hier nicht angegeben, da sie für alle Objekte einer Klasse identisch sind

```
class Video {  
    String author;  
    double length = 0.0;  
    String ytlink;  
    /*...*/  
}  
Video rr;  
rr = new Video("R.A.", 3.32, "dQw4w9WgXcQ");
```

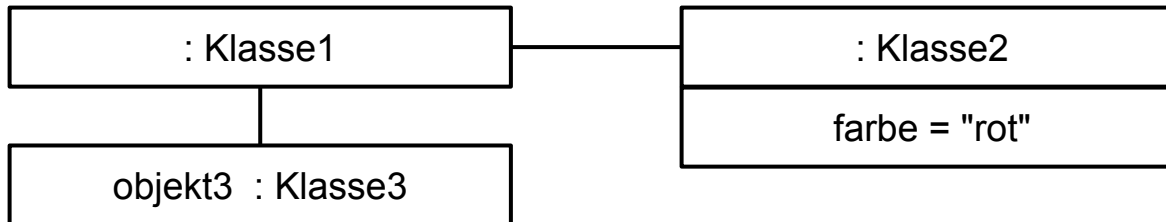
6.5. Einführung in UML: Objekte

- Darstellungsvarianten:



- Objekt ohne Klasse: Klasse geht aus Zusammenhang hervor
- Anonymes Objekt

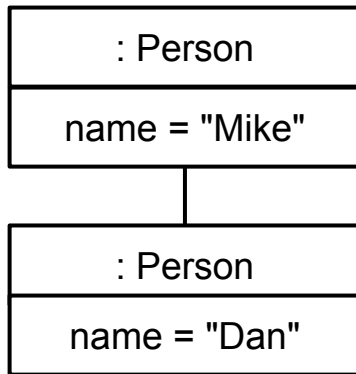
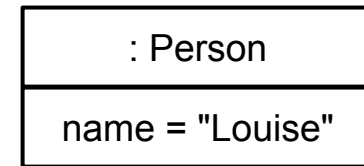
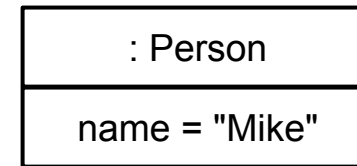
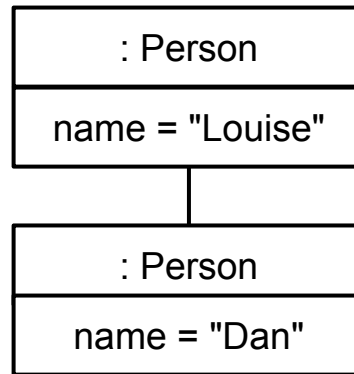
- Objektdiagramm mit Objektbeziehungen:
Beziehungen werden durch Verbindungslinien zwischen Objekten dargestellt



6.5. Einführung in UML: Objekte

Identität und Gleichheit von Objekten

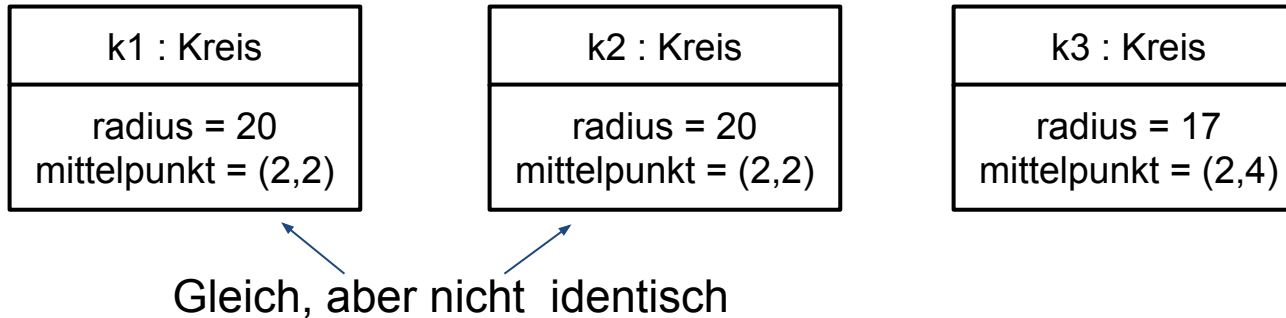
- Ein Objekt besitzt einen Zustand, ein wohldefiniertes Verhalten und eine Identität, die es von allen anderen Objekten unterscheidet
- Zwei Objekte sind gleich, wenn sie dieselben Attributwerte besitzen:

**Gleichheit****Identität**

6.5. Einführung in UML: Objekte

Objektidentität

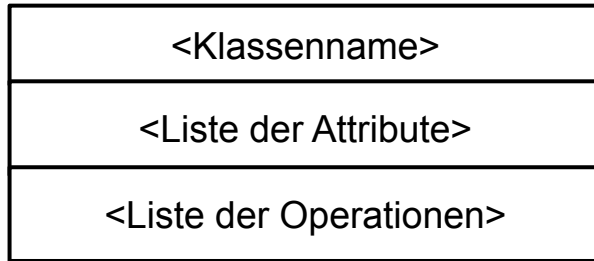
- Jedes Objekt ist per Definition, unabhängig von seinen konkreten Attributwerten, von allen anderen Objekten eindeutig zu unterscheiden.



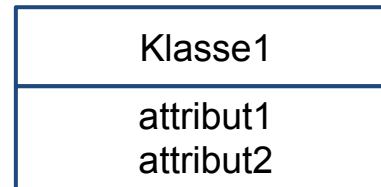
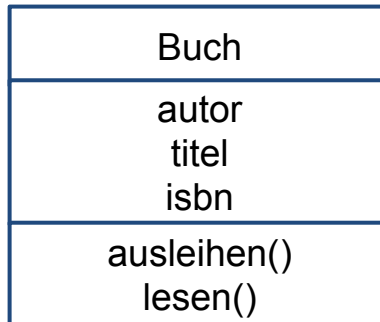
- Zur Laufzeit wahren Speicheradressen die Identität eines Objektes
- In Objektdatenbankmanagementsystem (ODBMS) werden oft künstlich erzeugte Identitätsnummern (sog. OID's) verwendet

6.5. Einführung in UML: Klassen

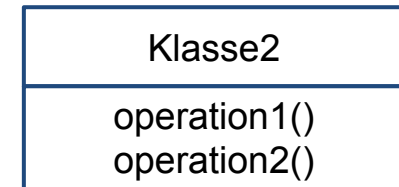
- Allgemeines Schema:



- z.B.:



Klasse ohne
Operationen

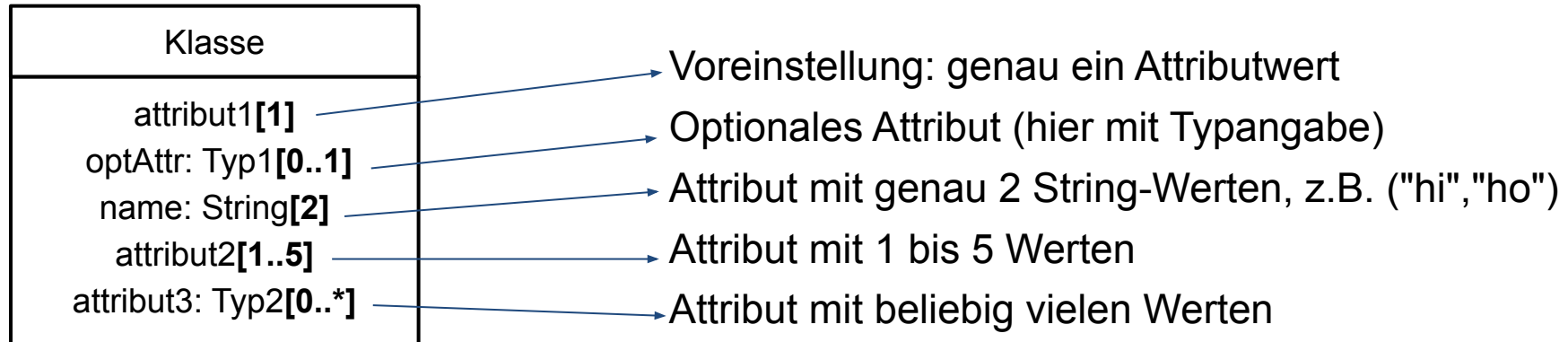


Klasse ohne
Attribute

6.5. Einführung in UML: Attribute

Multiplizitäten

- Attribute können mit einer Multiplizität versehen werden
- Die Multiplizität gibt an, aus wie vielen Werten das Attribut bestehen kann:
[Untergrenze .. Obergrenze (oder * für beliebig viele)]
- z.B.:

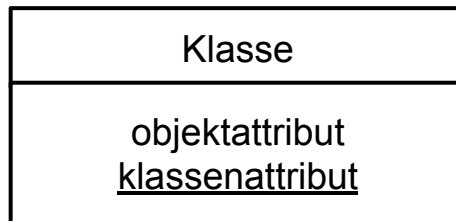




6.5. Einführung in UML: Attribute

Klassenattribute

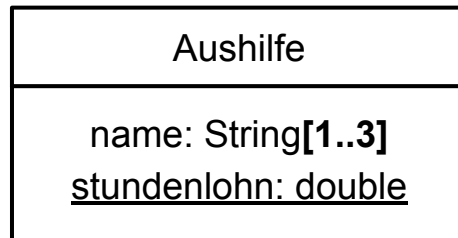
- Klassenattribute sind Attribute, für die nur ein einziger Attributwert für alle Objekte der Klasse existiert:
 - sie werden daher der Klasse zugeordnet, nicht den Objekten
 - sie existieren auch, wenn es (noch) kein Objekt der Klasse gibt
 - sie stellen oft auch Eigenschaften der Klasse selbst dar
- Klassenattribute werden durch Unterstreichen gekennzeichnet:



6.5. Einführung in UML: Attribute

In Java:

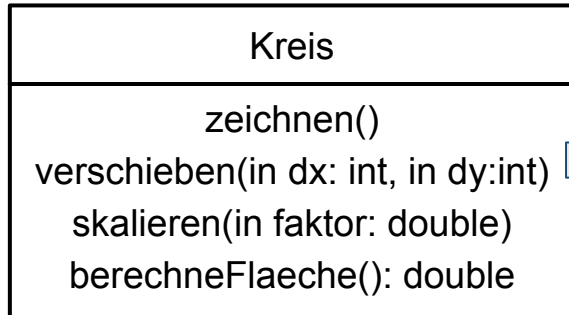
- Attribute mit Multiplizität ungleich 1 werden in Java durch Arrays (Felder) dargestellt
- in Java erfolgt bei der Deklaration eines Feldes keine Angabe, wieviel Elemente es enthalten kann
- Klassenattribute werden durch das vorgestellte Schlüsselwort `static` gekennzeichnet



```
class Aushilfe {  
    String[] name;  
    static double stundenlohn;  
}
```

6.5. Einführung in UML: Operationen

- einfache Operationen und Parameter:



- In UML ist die Angabe einer Parameterliste optional
- Wenn eine Liste angegeben wird, muss sie mindestens die Namen der Parameter enthalten

```
class Kreis {  
    void zeichnen() {  
        // Code der Operation 'zeichnen'  
    }  
    void verschieben(int dx, int dy) {  
        // Code der Operation 'verschieben'  
    }  
    void skalieren(double faktor) {  
        // Code der Operation 'skalieren'  
    }  
    double berechneFlaeche() {  
        // Code der Operation 'berechneFlaeche'  
    }  
}
```

6.5. Einführung in UML: Operationen

- Operation: ausführbare Tätigkeit, die von einem Objekt über eine Botschaft angefordert werden kann
 - alle Objekte einer Klasse haben dieselben Operationen
 - Operationen können direkt auf die Attributwerte eines jeden Objekts der Klasse zugreifen
 - Synonyme: Services, Methoden, Funktionen, Prozeduren
- Drei Arten von Operationen:
 - (Objekt-)Operationen: werden immer auf ein einzelnes (bereits existierendes) Objekt angewandt
 - Konstruktoroperationen: erzeugen ein neues Objekt und initialisieren seine Attribute
 - Klassenoperationen

6.5. Einführung in UML: Operationen

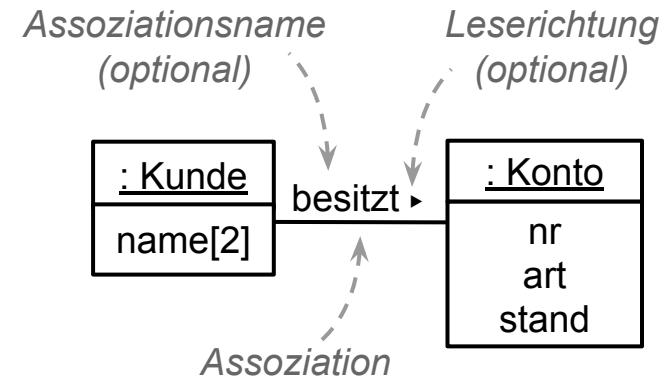
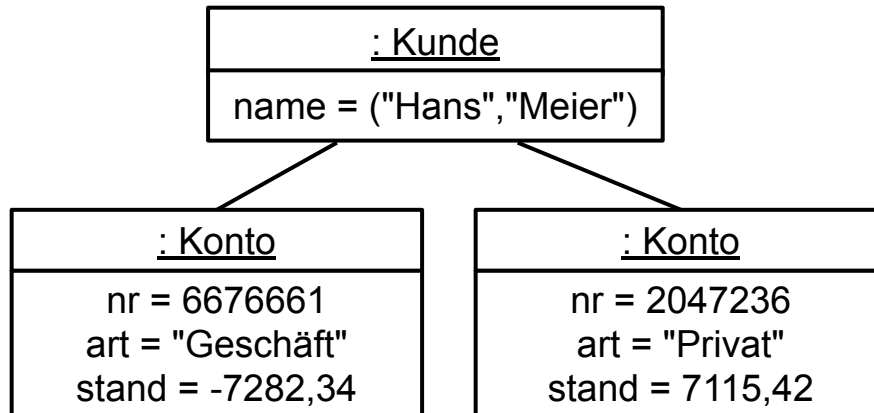
Klassenoperationen

- Klassenoperationen sind der Klasse zugeordnet und werden nicht auf ein einzelnes Objekt angewendet
- Sie werden durch Unterstreichen kenntlich gemacht
- In der OOA werden Klassenoperationen in zwei Fällen benutzt:
 - Manipulation von Klassenattributen ohne Beteiligung eines Objekts, z.B. `erhöheStundenlohn()`
 - Operation bezieht sich auf alle oder mehrere Objekte der Klasse
 - nutzt Objektverwaltung aus
 - z.B. `druckeListe()`:

Aushilfe
name <u>stundenlohn</u> stundenzahl
<u>erhöheStundenlohn()</u> <u>druckeListe()</u>

6.5. Einführung in UML: Assoziationen

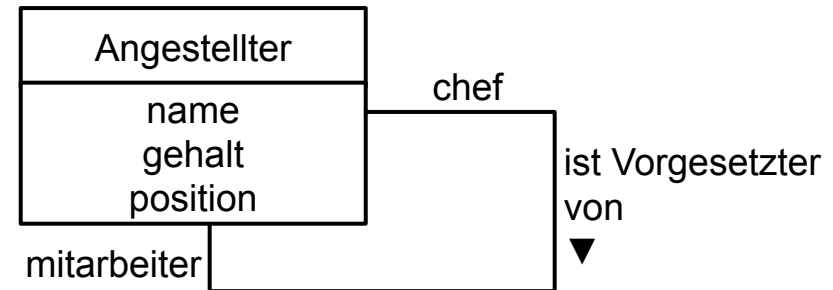
- Zwischen Objekten können Objektbeziehungen bestehen
- Assoziationen beschreiben gleichartige Objektbeziehungen zwischen Klassen
- Objektbeziehung ist Instanz einer Assoziation



6.5. Einführung in UML: Assoziationen

Reflexive Assoziationen, Rollennamen

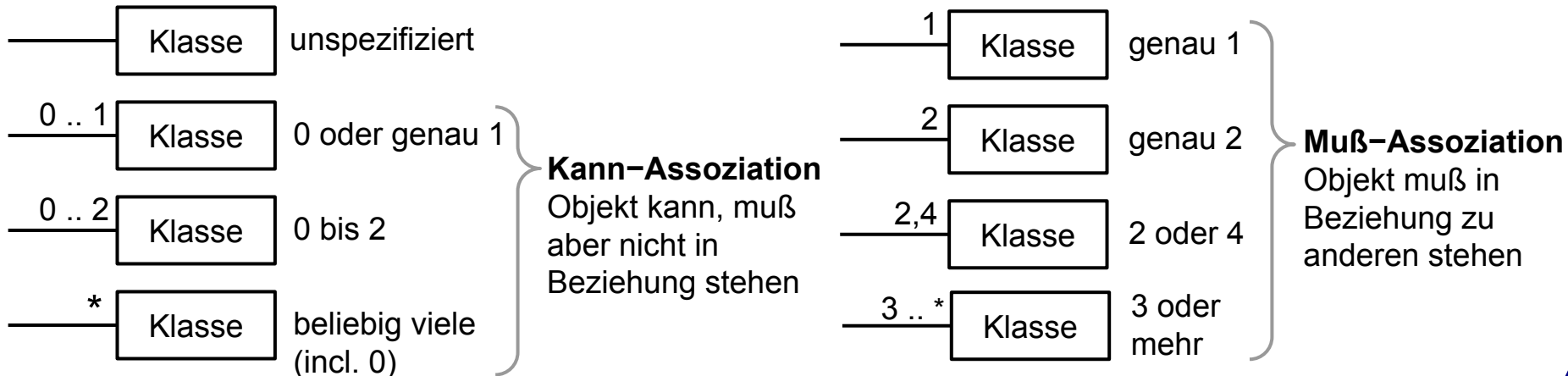
- An einer Objektbeziehung können auch zwei Objekte derselben Klasse beteiligt sein → führt zu reflexiver Assoziation zwischen Klassen
- Neben Assoziationsnamen auch Rollennamen möglich
 - Assoziationsname: Bedeutung der Assoziation
 - Rollename: Bedeutung einer Klasse in der Assoziation
- Beispiel reflexive Assoziation:
 - Assoziationsname:
ist Vorgesetzter von
 - Rollename:
chef, mitarbeiter



6.5. Einführung in UML: Assoziationen

Multiplizität von Assoziationen

- Assoziation sagt zunächst nur, dass ein Objekt andere Objekte kennen kann
- Multiplizität legt fest, wieviele Objekte ein Objekt kennen kann (oder muss)
- Die Multiplizität wird am Ende der Assoziations-Linie notiert:



6.5. Einführung in UML: Assoziationen

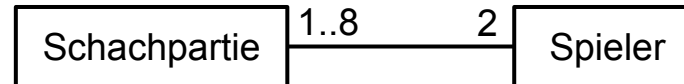
Multiplizität: Beispiele

• Beispiel Bank:



- ein Kunde muss mindestens ein Konto besitzen
- ein Konto gehört zu genau einem Kunden
- wenn der Kunde gelöscht wird, muss auch das Konto gelöscht werden
- ein Kunde kann beliebig viele Depots besitzen

• Beispiel Simultan-Schach:

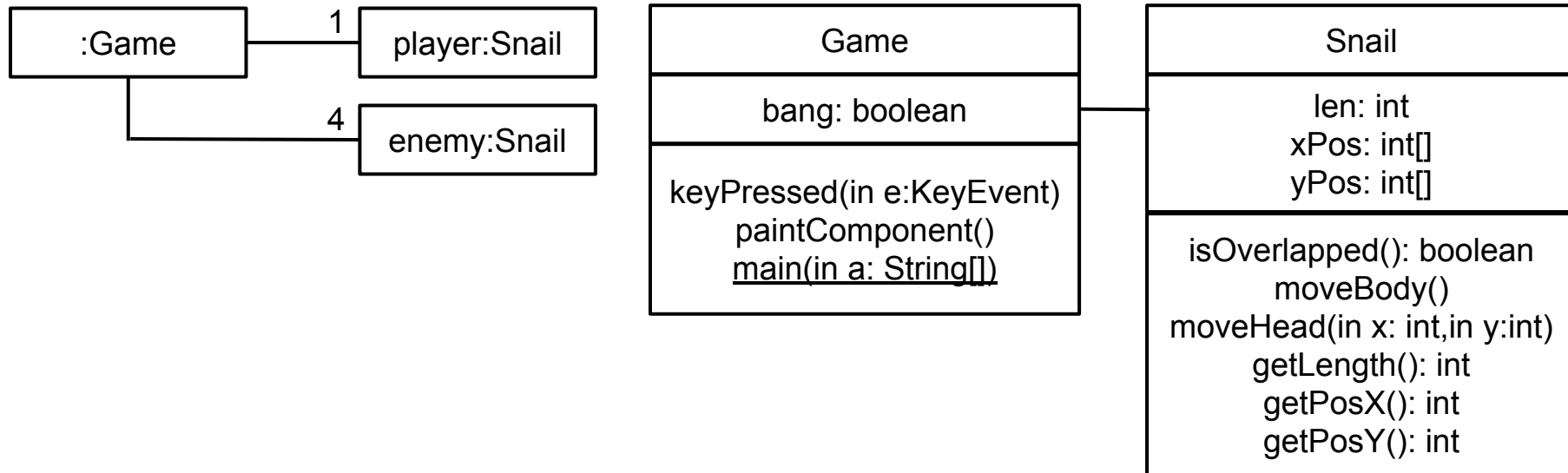


- jede Schachpartie wird von zwei Spielern gespielt
- ein Spieler spielt 1 bis 8 Partien gleichzeitig

6.5. Einführung in UML: Assoziationen

Multiplizität: Beispiele

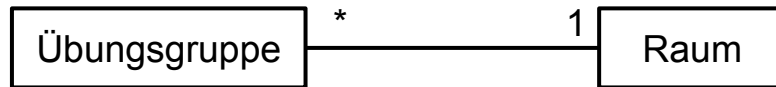
- Beispiel SnailGamev1:



6.5. Einführung in UML: Assoziationen

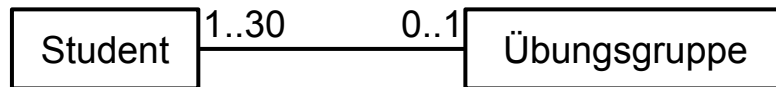
Umsetzung von Assoziationen in Java

- Muß-Assoziation



```
class Übungsgruppe {
    Raum übungsraum;
    // ...
}
class Raum {
    // ...
}
```

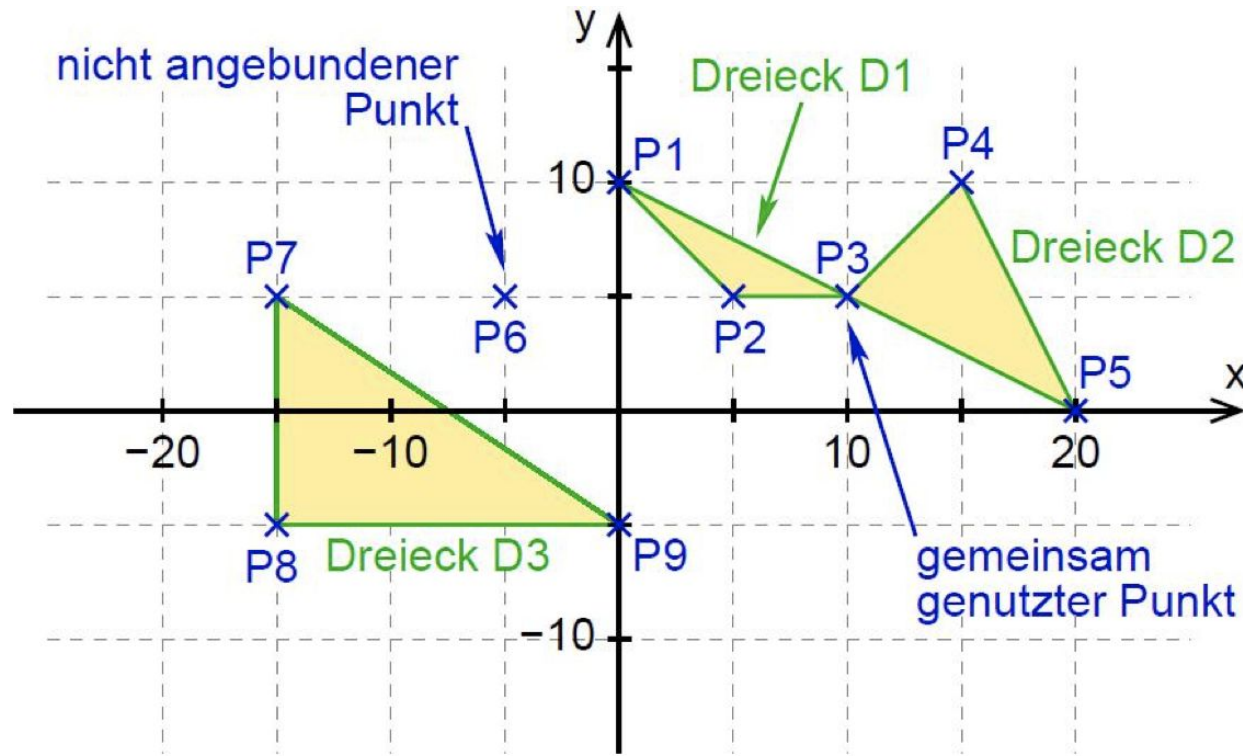
- Kann-Assoziation



```
class Student {
    Übungsgruppe gruppe;
    // ...
}
class Übungsgruppe {
    Student[] teilnehmer;
    // ...
}
```

6.5. Einführung in UML: Assoziationen

Beispiel

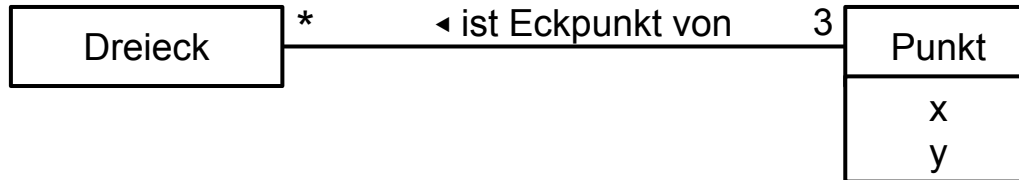


6.5. Einführung in UML: Assoziationen

Beispiel

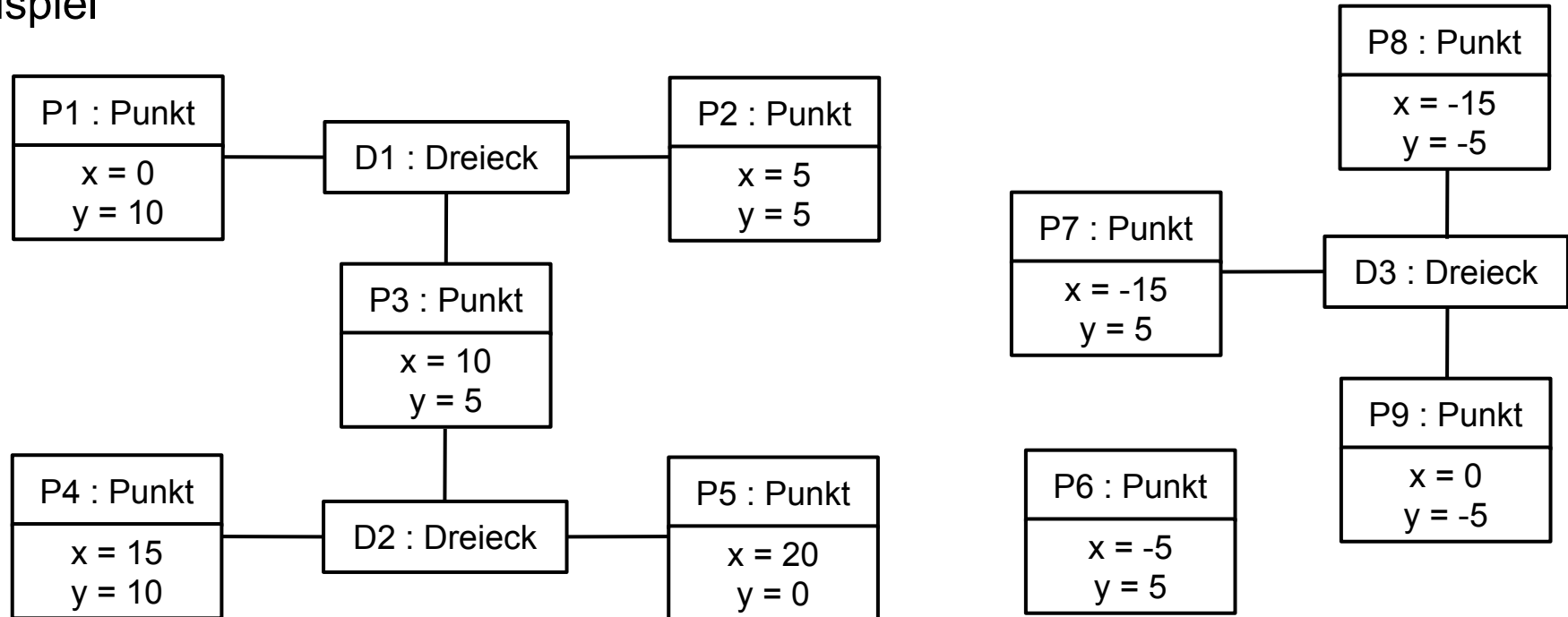
Klassendiagramm

- jedes Dreieck steht mit 3 Punkten in Verbindung
- ein Punkt ist Teil von beliebig vielen Dreiecken



6.5. Einführung in UML: Assoziationen

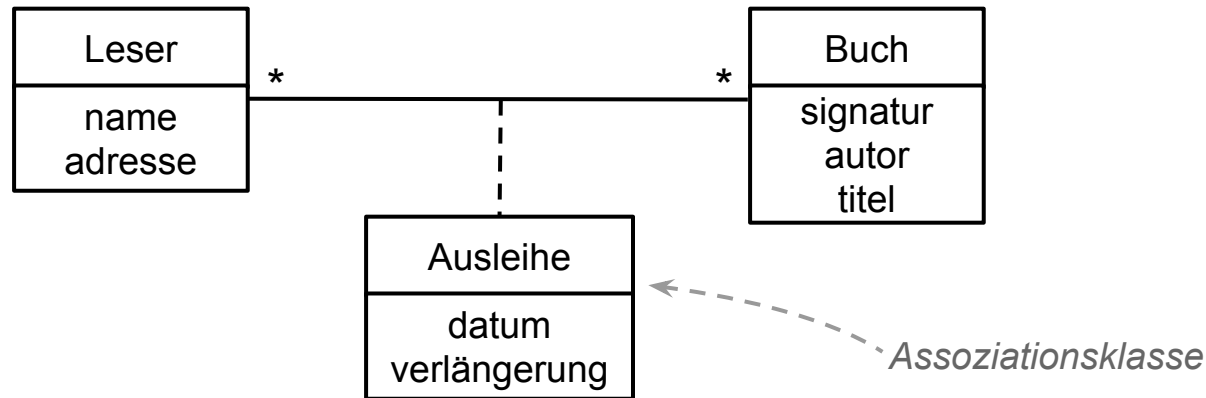
Beispiel



6.5. Einführung in UML: Assoziationen

Assoziationsklassen

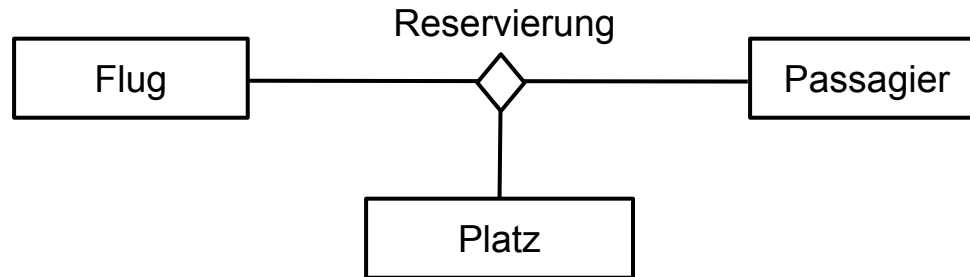
- Manchmal hat auch eine Assoziation Eigenschaften und Verhalten
- Dann: Assoziation wird explizit als Klasse modelliert
- Beispiel:



6.5. Einführung in UML: Assoziationen

Mehrstellige (n-äre) Assoziationen

- Assoziationen sind auch zwischen mehr als zwei Klassen möglich
- Darstellung am Beispiel einer ternären (dreistelligen) Assoziation:
eine Reservierung ist eine Beziehung zwischen Passagier, Flug und Platz



6.5. Einführung in UML: Assoziationen

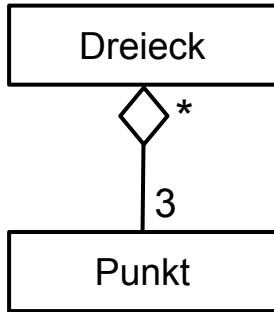
Aggregation und Komposition

- Häufige Abstraktion im täglichen Leben: Teile/Ganzes-Beziehung
 - "besteht aus" bzw. "ist Teil von"
 - z.B.: "Ein Auto besteht aus einer Karosserie, 4 Rädern, ..."
- Aggregation: Teile existieren selbständig und können (gleichzeitig) zu mehreren Aggregat-Objekten gehören
- Komposition: starke Form der Aggregation
 - Teil-Objekt gehört zu genau einem Komposit-Objekt
 - es kann nicht Teil verschiedener Komposit-Objekte sein
 - es kann nicht ohne sein Komposit-Objekt existieren
 - Beim Erzeugen (Löschen) des Komposit-Objekts werden auch seine Teil-Objekte erzeugt (gelöscht)

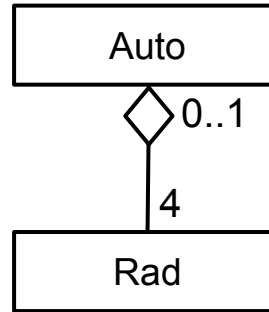
6.5. Einführung in UML: Assoziationen

Aggregation und Komposition

- Darstellung Aggregation:

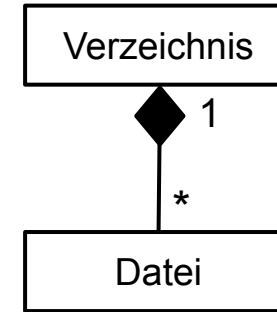


Ein Dreieck besteht immer aus 3 Punkten. Ein Punkt ist Teil von beliebig vielen (incl. 0) Dreiecken.



Ein Auto besteht u.a. aus 4 Rädern. Ein Rad ist Teil von höchstens einem Auto (es gibt auch Räder ohne Auto).

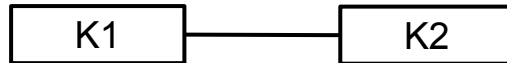
- Darstellung Komposition:



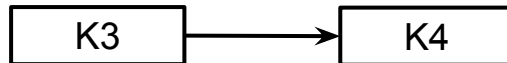
Ein Verzeichnis besteht aus beliebig vielen Dateien. Dateien stehen immer in einem Verzeichnis. Wird dieses gelöscht, so auch alle enthaltenen Dateien.

6.5. Einführung in UML: Assoziationen und Navigierbarkeit

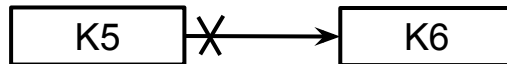
- Im Entwurf wird zusätzlich die Navigierbarkeit modelliert:
- Assoziation von A nach B navigierbar => Objekte von A können auf Objekte von B zugreifen (aber nicht notwendigerweise umgekehrt)
- Darstellung in UML:



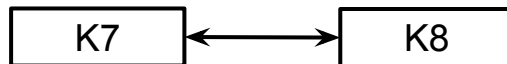
Navigierbarkeit ist unspezifiziert



K3-Objekte können auf K4-Objekte zugreifen, keine Aussage über umgekehrte Richtung



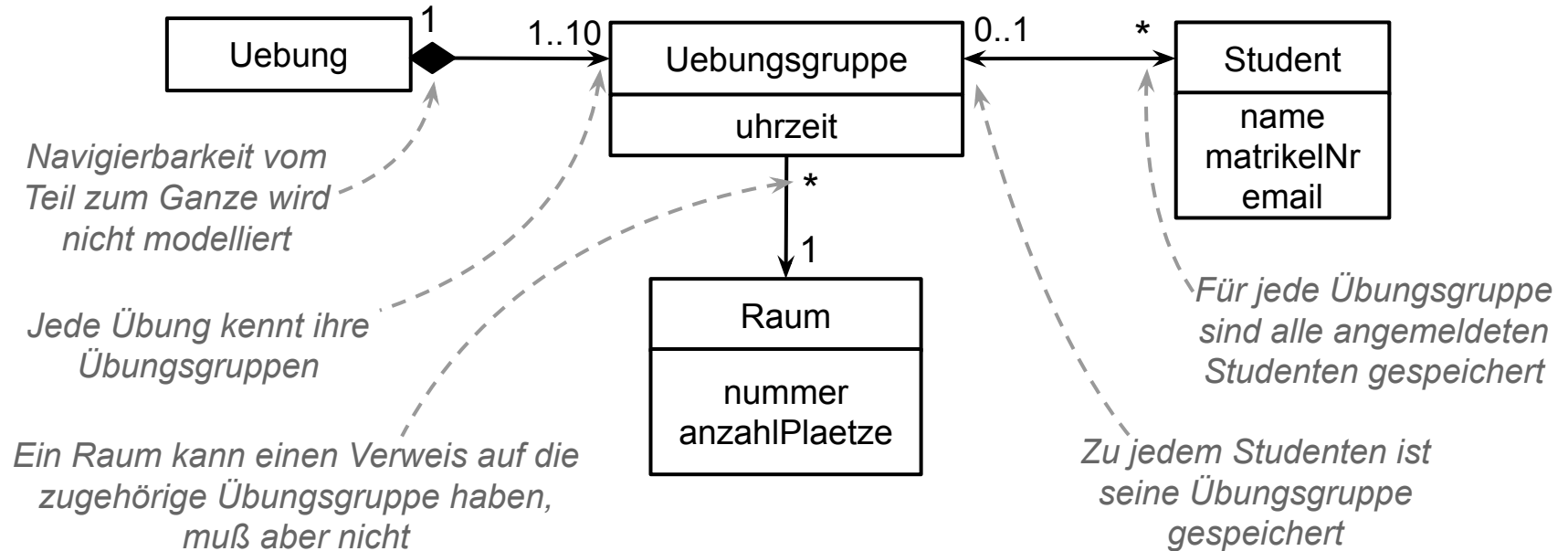
K5-Objekte können auf K6-Objekte zugreifen, aber nicht umgekehrt



Bidirektionale Assoziation: K7-Objekte können auf K8-Objekte zugreifen und umgekehrt

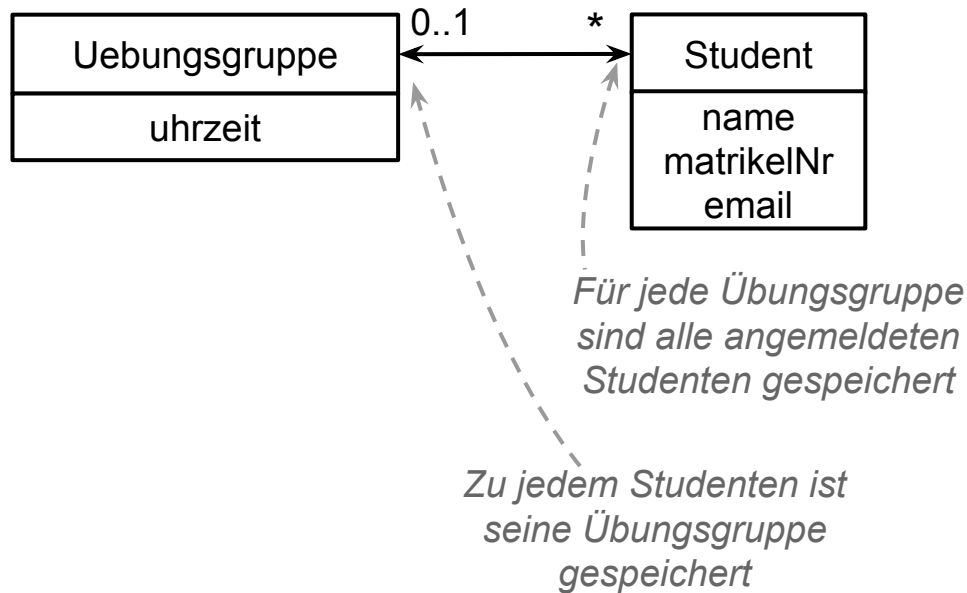
6.5. Einführung in UML: Assoziationen und Navigierbarkeit

- Beispiel:



6.5. Einführung in UML: Assoziationen und Navigierbarkeit

- Beispiel:



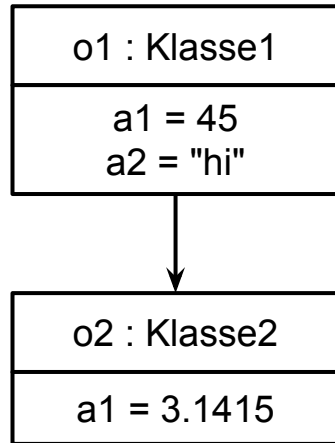
```
class Uebungsgruppe {  
    Student[] teilnehmer;  
}  
class Student {  
    Uebungsgruppe gruppe;  
}  
// ...  
Student student = new Student();  
// ...  
if (student.gruppe==null) {  
    // ...  
} else {  
    // ...  
}
```

6.5. Einführung in UML: Assoziationen und Navigierbarkeit

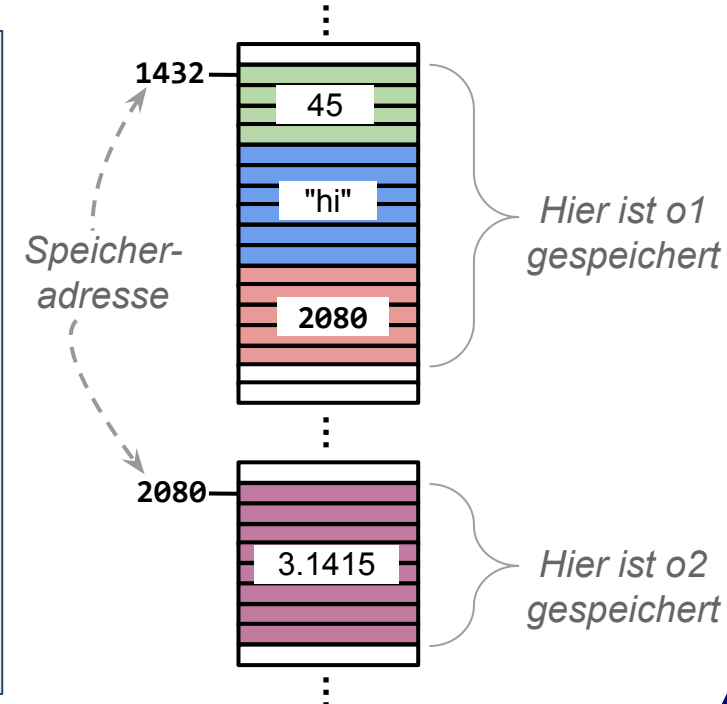
- Was bedeutet die Navigierbarkeit für die Realisierung von Assoziationen?
- $K1 \rightarrow K2$ bedeutet, dass das K1-Objekt das K2-Objekt (bzw. die K2-Objekte) "kennen" muss: das K1-Objekt muss eine **Referenz** (auch Verweis, Zeiger) auf das K2-Objekt speichern
- In der Programmierung ist eine Referenz ein spezieller Wert, über den ein Objekt eindeutig angesprochen werden kann, z.B. Adresse des Objekts im Speicher des Rechners
- Referenzen können wie normale Werte benutzt werden:
 - Speicherung in Attributen
 - Übergabe als Parameter / Ergebnis von Operationen

6.5. Einführung in UML: Assoziationen und Navigierbarkeit

● Beispiel:

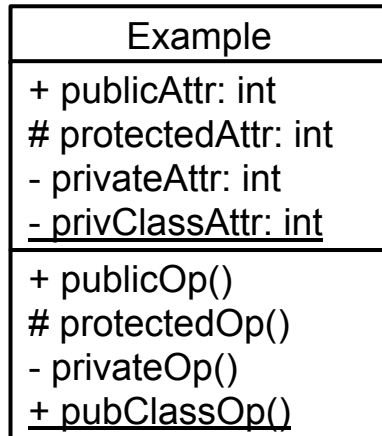


```
class Klasse1 {
    private int a1 = 45;
    private String a2 = "hi";
    public Klasse2 k2;
}
class Klasse2 {
    private double a1;
    public Klasse2(double a1) {
        this.a1 = a1;
    }
}
// ...
Klasse1 o1 = new Klasse1();
Klasse2 o2 = new Klasse2(3.1415);
o1.k2 = o2;
```



6.5. Einführung in UML: Sichtbarkeit

- **public** (öffentlich): sichtbar für alle Klassen (auf Attribut kann von allen Klassen aus zugegriffen werden, Operation kann von allen Klassen aufgerufen werden)
- **private** (privat): sichtbar nur innerhalb der Klasse (kein Zugriff/Aufruf durch andere Klassen möglich)
- **protected** (geschützt): sichtbar nur innerhalb der Klasse und ihren Unterklassen

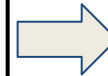
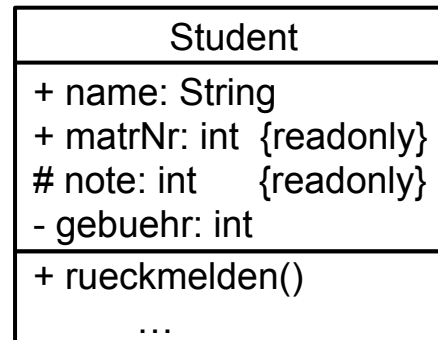


```
class Example {  
    public int publicAttr;  
    protected int protectedAttr;  
    private int privateAttr;  
    private static int privClassAttr;  
  
    public void publicOp() { /*...*/ }  
    protected void protectedOp() { /*...*/ }  
    private void privateOp() { /*...*/ }  
    public static void pubClassOp() { /*...*/ }  
}
```


6.5. Einführung in UML: Sichtbarkeit

Hinweise zu Get- und Set-Methoden

- Get- und Set-Methoden (Namenskonvention: getXxx(), setXxx()) werden i.d.R. nicht im Klassendiagramm dargestellt
- Die Sichtbarkeit des Attributs im Klassendiagramm bestimmt die Sichtbarkeit der Get- und Set-Methoden im Java-Code, im Java-Code ist das Attribut selbst immer private
- Bei readOnly-Attributen: Set-Methode ist private bzw. kann auch ganz fehlen



```
class Student {
    private String name;
    private int matrNr;
    private int note;
    private int gebuehr;
    // Hier keine Get- oder Set- Methoden
    // für gebuehr, da privates Attribute

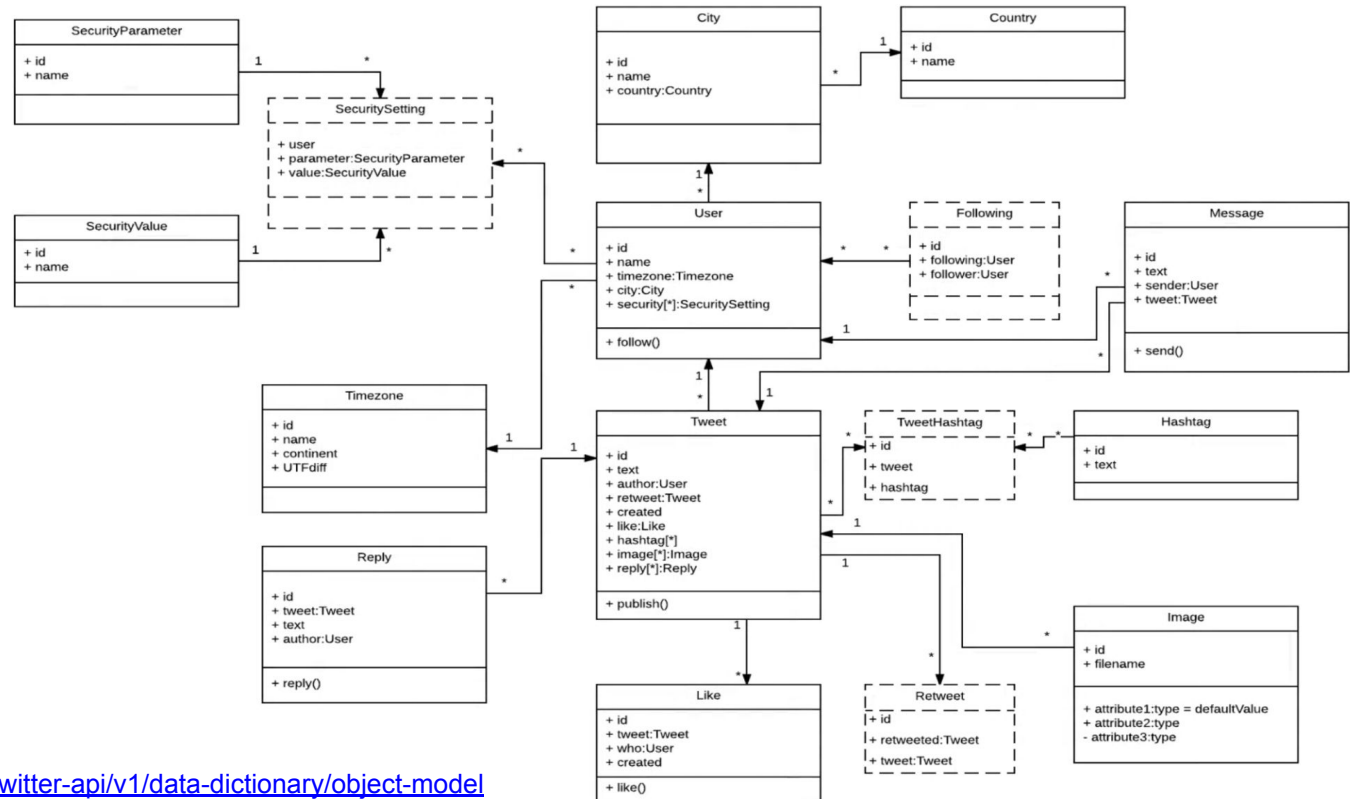
    public String getName() { /*...*/ }
    public void setName(String n) { /*...*/ }

    public int getMatrNr() { /*...*/ }
    // Hier kein setMatrNr: matrNr wird
    // bei Objekterzeugung direkt gesetzt

    protected int getNote() { /*...*/ }
    private void setNote(int n) { /*...*/ }
}
```

6.6. Beispiel: Twitter

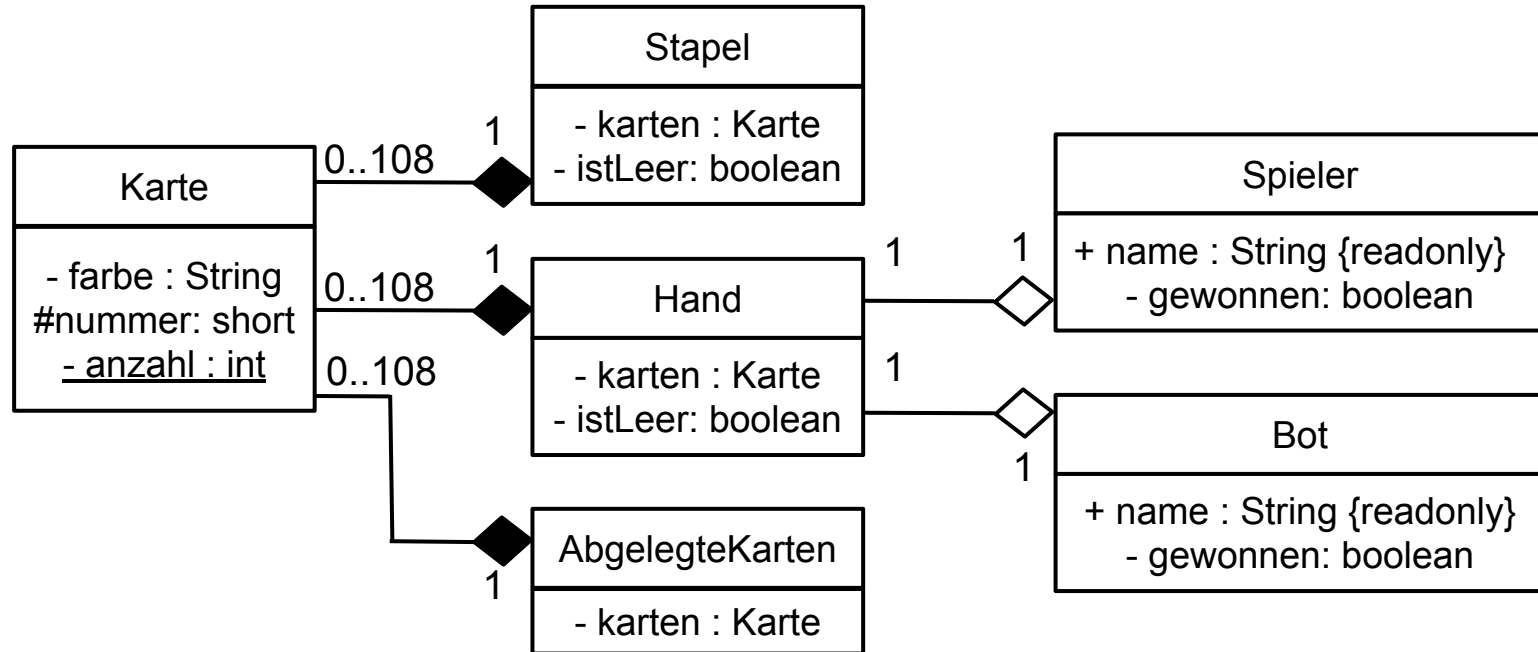
(<https://www.devcamp.com/trails/uml-foundations/campsites/class-diagrams/guides/4496d36d-ab10-475b-94b8-7a33c89bed4e>)



details:

<https://developer.twitter.com/en/docs/twitter-api/v1/data-dictionary/object-model>

6.7. Beispiel UML → Java: Uno (🌶️🌶️)



6.7. Beispiel Java → UML (🌶️🌶️🌶️)

```
class MeineKlasse {  
    private String name;  
    protected int nr;  
    private static boolean markiert;  
    public MeineKlasse k;  
    public MeineKlasse(String a, int b) {  
        name = a; nr = b; markiert = false;  
        k = new MeineKlasse("meine Klasse", 12);  
    }  
    public String getName() { return name; }  
    protected void setMarkiert(boolean m) { markiert = m; }  
    public static void main(String[] s) {  
        MeineKlasse k = new MeineKlasse("Klasse", 127);  
    }  
}
```

Objektorientierte und Formale Programmierung

-- Java Grundlagen --

7. Vererbung und Polymorphie

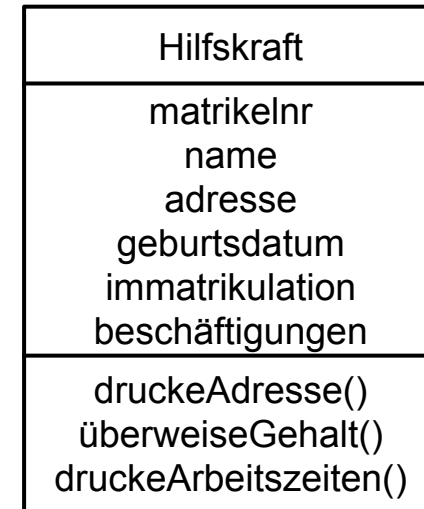
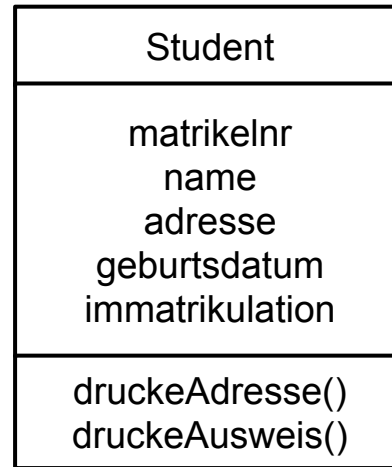
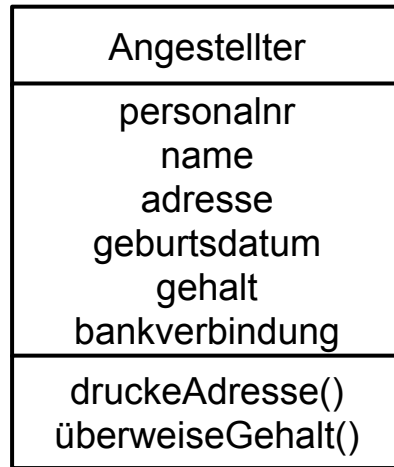
7.1. Generalisierung in UML und Java

- Generalisierung: Beziehung zwischen einer allgemeineren Klasse (Basisklasse, Oberklasse) und einer spezialisierteren Klasse (Unterklasse)
 - die spezialisierte Klasse ist konsistent mit der Basisklasse, enthält aber zusätzliche Attribute, Operationen und / oder Assoziationen
 - ein Objekt der Unterklasse kann überall da verwendet werden, wo ein Objekt der Oberklasse erlaubt ist
- Nicht nur: Zusammenfassung gemeinsamer Eigenschaften und Verhaltensweisen,
sondern immer auch: Generalisierung im Wortsinn
 - jedes Objekt der Unterklasse ist ein Objekt der Oberklasse
- Generalisierung führt zu einer Klassenhierarchie

7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

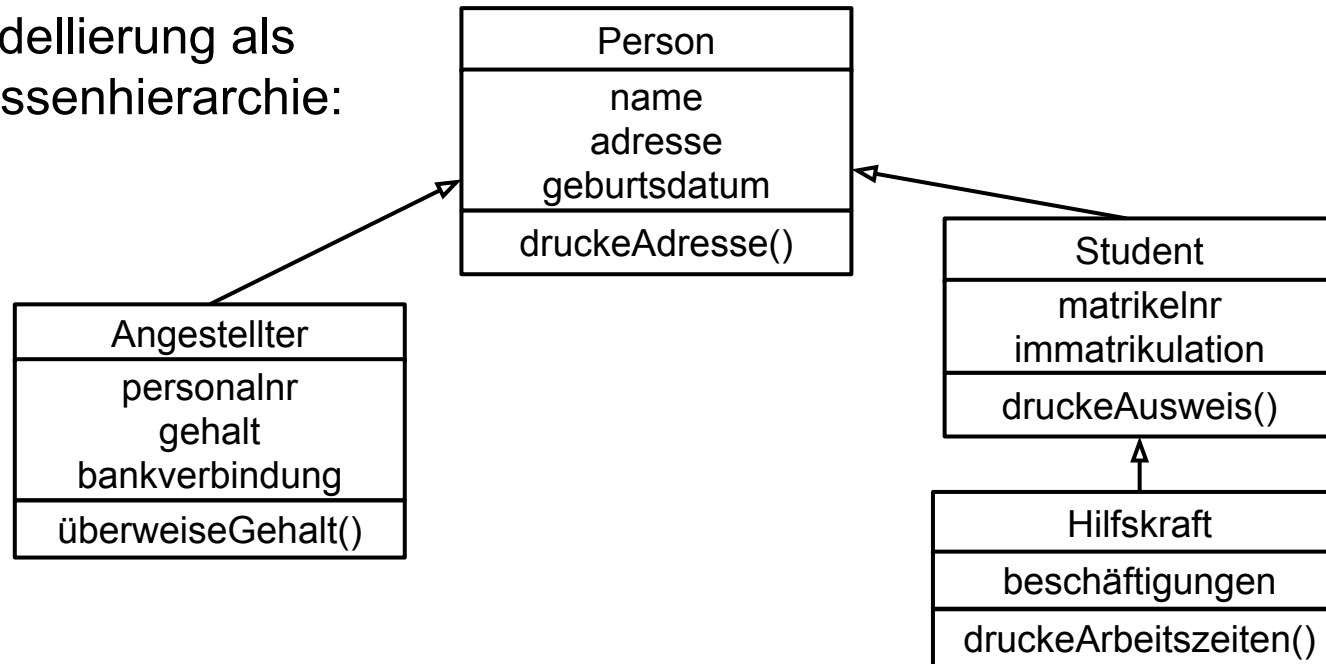
- Modellierung als unabhängige Klassen:



7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

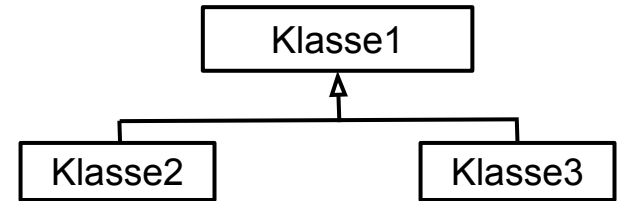
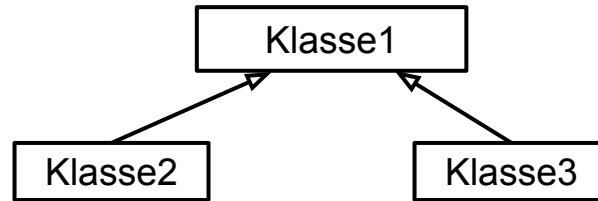
- Modellierung als Klassenhierarchie:



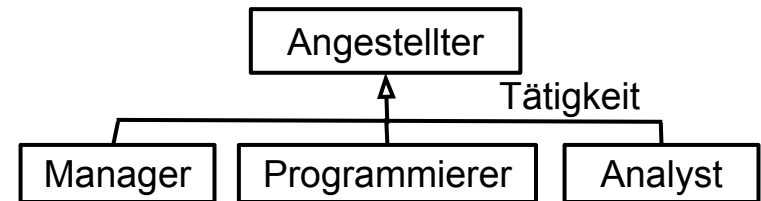
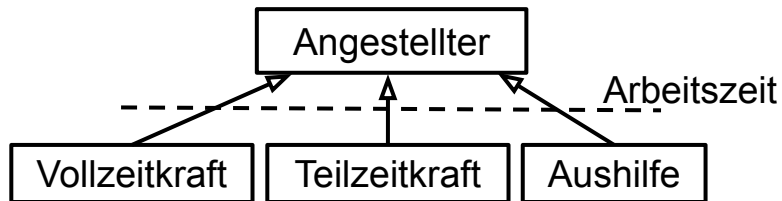
7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

Darstellung:



- Ein zusätzlicher Diskriminator (Generalisierungsmenge) kann das Kriterium angeben, nach dem klassifiziert wird:





7.1. Generalisierung in UML und Java

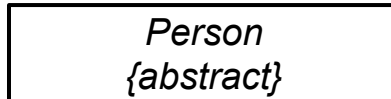
Abstrakte Klassen und Operationen

- Im vorherigen Beispiel wurden im Modell nur Personen betrachtet, die entweder Angestellter, Student oder Hilfskraft sind
- Die neue Basisklasse "Person" wird daher als *abstrakte* Klasse modelliert
- von einer abstrakten Klasse können keine Instanzen (Objekte) erzeugt werden
- Darstellung:

- Klassenname in Kursivschrift:



- Oder für handschriftliche Diagramme:

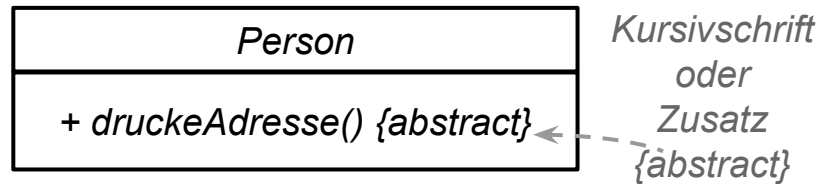


```
abstract class Person {
    String name;
    String adresse;
    Date geburtsdatum;
    public void druckeAdresse() {
        //...
    };
}
```

7.1. Generalisierung in UML und Java

Abstrakte Klassen und Operationen

- Eine abstrakte Operation einer Klasse wird von der Klasse nur deklariert, nicht aber implementiert
 - die Klasse legt nur Signatur und Ergebnistyp fest
 - die Implementierung muss in einer Unterklasse durch überschreiben der ererbten Operation erfolgen
- Abstrakte Operationen dürfen nur in abstrakten Klassen auftreten
- Darstellung:



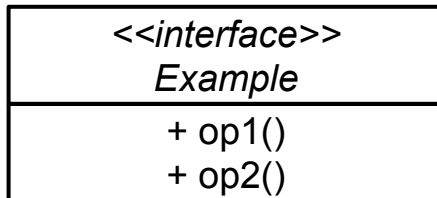
```
abstract class Person {  
    String name;  
    String adresse;  
    Date geburtsdatum;  
    public abstract void druckeAdresse();  
}
```



7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Eine (abstrakte) Klasse mit abstrakten Operationen
- Beschreibt eine Menge von Signaturen von Operationen
- Java erlaubt in Schnittstellen nur *öffentliche, unveränderliche* und *initialisierte* Klassenattribute: `public static final int M_PI = 3;`,
`public`, `static` und `final` können dann auch entfallen: `int M_PI = 3;`
- Die Operationen einer Schnittstelle sind immer abstrakt und öffentlich
`public` kann in Java auch entfallen
- Darstellung:



```
interface Example {  
    public void op1();  
    public void op2();  
}
```

```
interface Example {  
    void op1();  
    void op2();  
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Schnittstellen definieren "Dienstleistungen" für aufrufende Klassen, sagen aber nichts über deren Implementierung aus
- funktionale Abstraktionen, die festlegen was implementiert werden soll, aber nicht wie
- Schnittstellen realisieren damit das Geheimnisprinzip in der stärksten Form:
 - Java-Klassen verhindern über Sichtbarkeiten zwar den Zugriff auf Interna der Klasse, ein Programmierer kann diese aber trotzdem im Java-Code der Klasse lesen
 - der Java-Code einer Schnittstelle enthält nur die öffentlich sichtbaren Definitionen, nicht die Implementierung

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Schnittstellen sind von ihrer Struktur und Verwendung her praktisch identisch mit abstrakten Klassen:
 - sie können wie abstrakte Klassen nicht instanziiert werden
 - Referenzen auf Schnittstellen und auch abstrakte Klassen sind aber möglich, sie können auf Objekte zeigen, die die Schnittstelle implementieren
- Klassen können von Schnittstellen "erben"
 - "vererbt" werden nur die Signaturen der Operationen
 - die Klassen müssen diese Operationen selbst implementieren
 - man spricht in diesem Fall von einer Implementierungs-Beziehung statt von Generalisierung

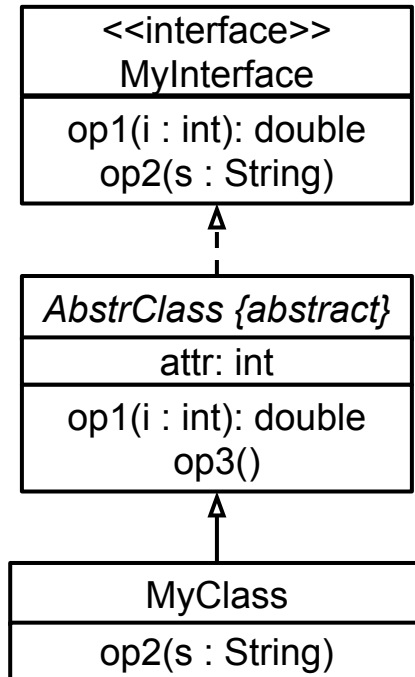
7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Die Implementierungs-Beziehung **implements** besagt, dass eine Klasse die Operationen einer Schnittstelle implementiert
 - die Klasse erbt die abstrakten Operationen, d.h. deren Signaturen incl. Ergebnistyp: Diese müssen dann geeignet überschrieben werden
 - werden nicht alle Operationen überschrieben (d.h. implementiert), so bleibt die erbende Klasse abstrakt
 - die überschreibenden Operationen müssen öffentlich sein
 - die Klasse kann zusätzlich weitere Operationen und Attribute definieren
- Eine Klasse kann mehrere Schnittstellen implementieren (eine Art Mehrfachvererbung, die auch in Java erlaubt ist)

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen): Die Implementierungs-Beziehung **implements**



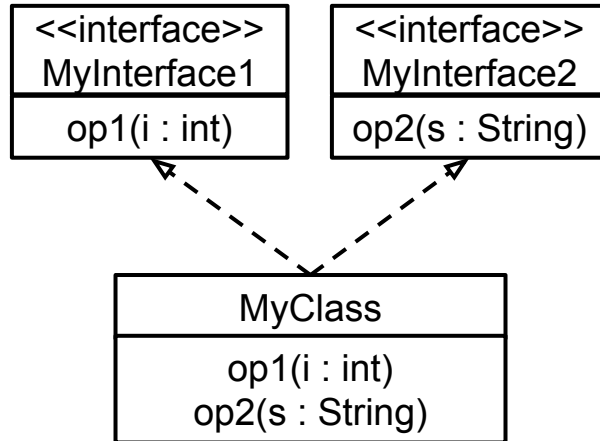
```
interface MyInterface {
    public double op1(int i);
    public void op2(String s);
}

abstract class AbstrClass implements MyInterface {
    protected int attr;
    public double op1(int i) { /* ... */ }
    public void op3() { /* ... */ }
}

class MyClass extends AbstrClass {
    public void op2(String s) { /* ... */ }
}
```


7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen): Implementierung mehrerer Schnittstellen



```
interface MyInterface1 {
    public void op1(int i);
}

interface MyInterface2 {
    public void op2(String s);
}

class MyClass implements MyInterface1, MyInterface2 {
    public void op1(int i) { /* ... */ }
    public void op2(String s) { /* ... */ }
}
```

7.1. Generalisierung in UML und Java

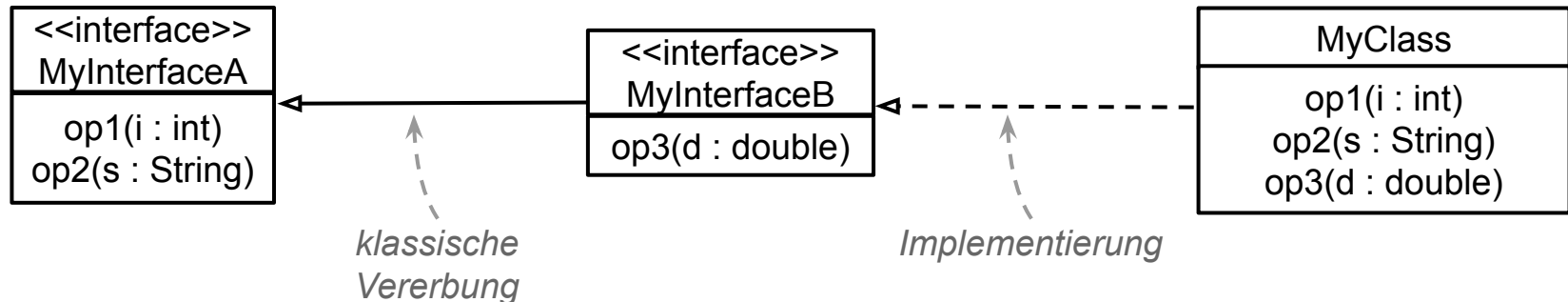
Beispiel Schnittstelle: Interface.java (🌶️🌶️)

```
/** Schreiben Sie ein Java-Programm, um eine Schnittstelle Figur  
mit der Methode getFlaeche() zu erstellen. Erstellen Sie drei  
Klassen, Rechteck, Kreis und Dreieck, die die Figur-  
Schnittstelle implementieren. Implementieren Sie dann die  
Methode getFlaeche() für jede der drei Klassen.  
*/
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen): Vererbung

- Schnittstellen können von anderen Schnittstellen erben (analog zu Klassen)
- Darstellung in UML und Java wie bei normaler Vererbung
- Die implementierende Klasse muss die Operationen "ihrer" Schnittstelle und aller Ober-Schnittstellen implementieren:



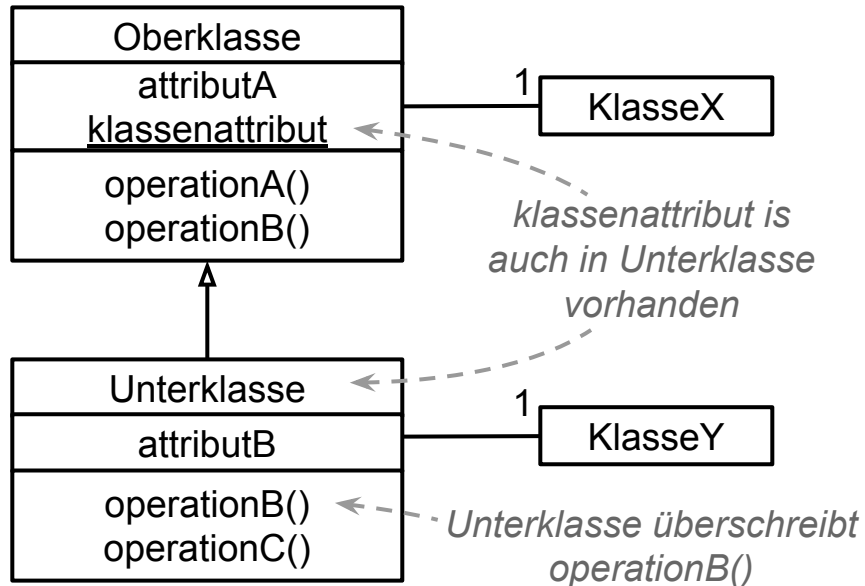
7.1. Generalisierung in UML und Java

Vererbung

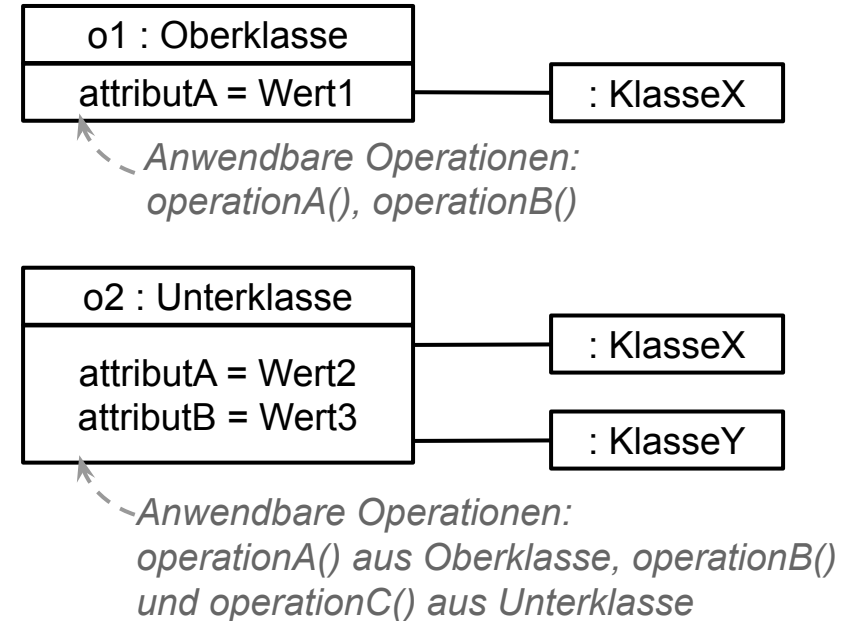
- Eine Unterklasse übernimmt (erbt) von ihren Oberklassen
 - alle *Attribute* und *Klassenattribute*, auch deren Anfangswert wenn definiert
 - alle *Operationen* und *Klassenoperationen*, d.h. alle Operationen einer Oberklasse können auch auf ein Objekt der Unterklasse angewendet werden
 - alle Assoziationen
- Die Unterklasse kann zusätzliche Attribute, Operationen und Assoziationen hinzufügen, aber ererbte *nicht löschen*
- Die Unterklasse kann das Verhalten *neu definieren*, indem sie Operationen der Oberklasse *überschreibt* (d.h. eine Operation gleichen Namens neu definiert)

7.1. Generalisierung in UML und Java

Vererbung: Beispiel
Klassendiagramm:



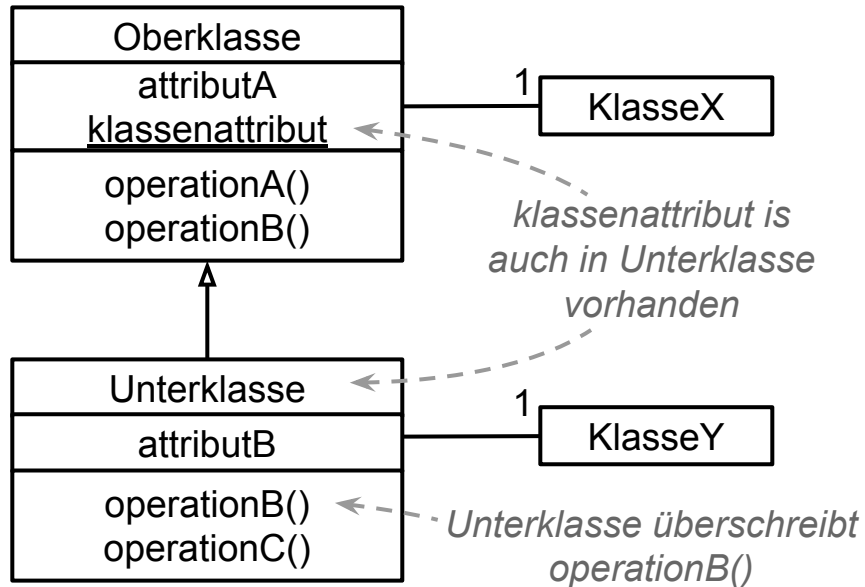
Objektdiagramm:



7.1. Generalisierung in UML und Java

Vererbung: Beispiel

Klassendiagramm:



in Java:

```
class Oberklasse {
    int attributA;
    static double klassenattribut;
    KlasseX kx;
    void operationA() { // ... }
    int operationB() { return 4; }
}

class Unterklasse extends Oberklasse {
    char attributB;
    KlasseY ky;
    int operationB() { return 5; }
    void operationC() { // ... }
}
```

7.1. Generalisierung in UML und Java

Vererbung: Beispiel
in Java:

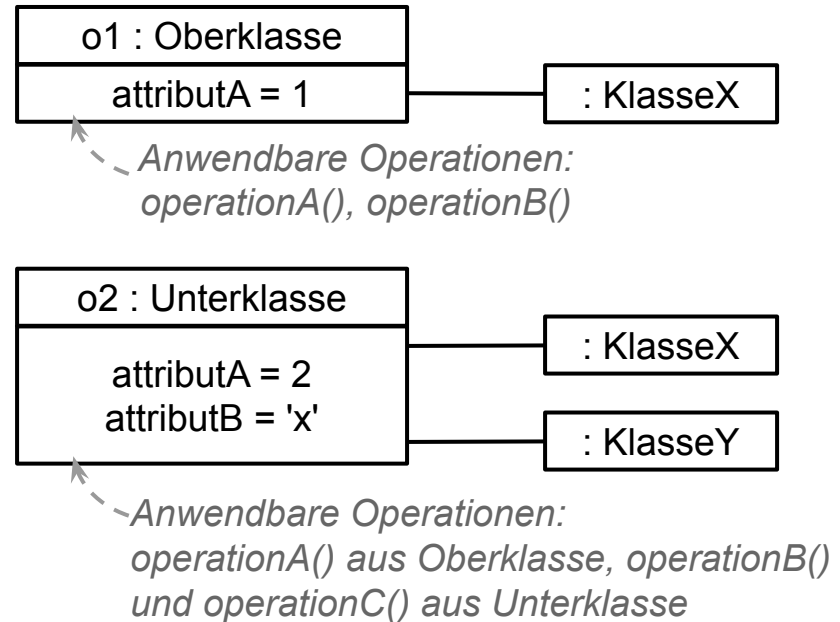
```
KlasseX x = new KlasseX();
KlasseY y = new KlasseY();

Oberklasse.klassenattribut = 1.234;

Oberklasse o1 = new Oberklasse();
o1.attributA = 1; o1.kx = x;
o1.operationA();
o1.operationB(); // returns 4

Unterklasse o2 = new Unterklasse();
o2.attributA = 2; o2.attributB = 'x';
o2.kx = x; o2.ky = y;
o2.operationA(); o2.operationC();
o2.operationB(); // returns 5
```

Objektdiagramm:



7.1. Generalisierung in UML und Java

Vererbung: Überschreiben von Methoden

Beim Überschreiben einer vererbten Methode müssen Signatur und Ergebnistyp *exakt* übereinstimmen:

// Richtig:

```
class Ober {  
    void op(int p) { /* ... */ }  
}  
  
class Unter extends Ober {  
    void op(int p) { /* ... */ }  
}
```

// Falsch:

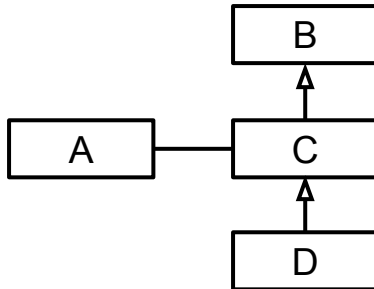
```
class Unter1 extends Ober {  
    // neue Operation (kein Überschreiben):  
    void op(double p) { /* ... */ }  
}  
  
class Unter1 extends Ober {  
    // Fehler:  
    int op(int p) { /* ... */ }  
}
```


7.1. Generalisierung in UML und Java

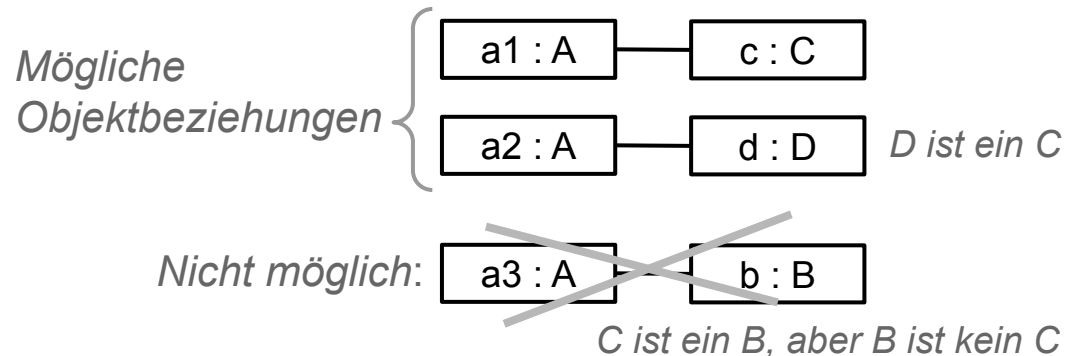
Assoziationen und Generalisierung

- Assoziationsbeziehungen werden ebenfalls vererbt: wenn es eine Assoziation zwischen A und B gibt, kann ein Objekt von A auch mit einem Objekt einer Unterklasse von B in Beziehung stehen
- z.B.:

Klassendiagramm:



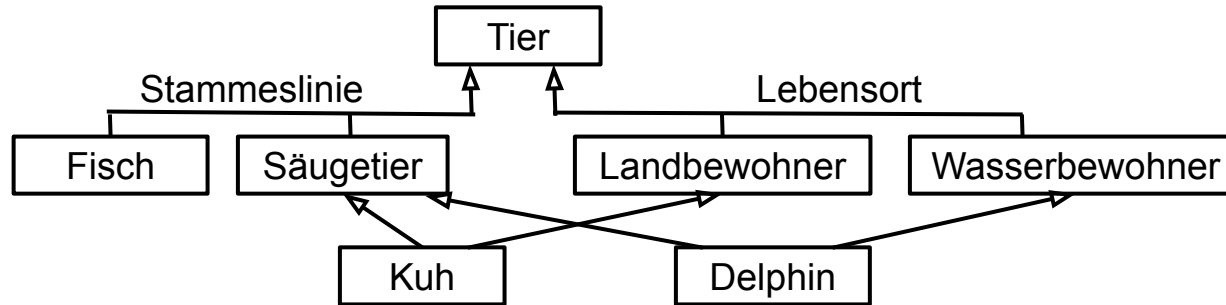
Objektdiagramm:



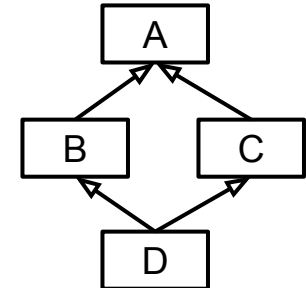
7.1. Generalisierung in UML und Java

Mehrfachvererbung

- Eine Klasse kann in UML auch von mehreren direkten Basisklassen erben:



- Konzept wird nicht von Java unterstützt
 - Probleme, wenn z.B. Oberklassen verschiedene, aber gleichnamige Attribute / Operationen besitzen
- Diamond-Problem entsteht durch Mehrfachvererbung in der objektorientierten Programmierung



7.1. Generalisierung in UML und Java

Diskussion: Vorteile und Probleme

- Bessere Strukturierung des Modell-Universums
- Aufbauend auf vorhandenen Klassen können ähnliche Klassen mit wenig Aufwand erstellt werden
- Einfache Änderbarkeit: Änderung von Attributen / Operationen der Basisklasse wirkt sich automatisch auf die Unterklassen aus
 - dies kann aber auch unerwünscht sein
- Klassen sind schwieriger zu verstehen / zu verwenden
 - auch alle Oberklassen müssen verstanden werden
- Gefahr, überflüssige Klassenhierarchien zu bilden
- Fazit: Generalisierung mit Bedacht verwenden

Beispiel: SnailGame v2 (moodle)

```
class Game extends JPanel implements KeyListener {
    private Snail player;
    private Snail[] enemy;
    private boolean bang = false;

    public int xSize = 800;
    public int ySize = 400;

    public Game() { /* ... */ }
    public void keyPressed(KeyEvent e) {...}
    public void keyReleased(KeyEvent e) {}
    public void keyTyped(KeyEvent e) {}
    public void paintComponent( Graphics g ) {...}
    public static void main(String[] a) {...}
}
```

```
interface KeyListener {
    public void keyPressed(KeyEvent e);
    public void keyReleased(KeyEvent e);
    public void keyTyped(KeyEvent e);
}
```

```
public class JPanel extends JComponent
implements Accessible {
    public void paintComponent( Graphics g )
    {...}
}
```



Beispiel: SnailGame v2 (moodle)

```
class Game extends JPanel implements KeyListener {  
    private Snail player;  
    private Snail[] enemy;  
    private boolean bang = false;  
  
    public int xSize = 800;  
    public int ySize = 400;  
  
    public Game() { /* ... */ }  
    public void keyPressed(KeyEvent e) {...}  
    public void keyReleased(KeyEvent e) {}  
    public void keyTyped(KeyEvent e) {}  
    public void paintComponent( Graphics g )  
    public static void main(String[] a) {...}  
}
```

```
class Snail {  
    private short len = 7;  
    private int[][] pos = null;  
    public static short segSize = 20;  
  
    public Snail(short len) { /* ... */ }  
    public Snail(short len, int x, int y) {...}  
    public boolean isOverlapped(Snail snail) {...}  
    public void moveBody() { /* ... */ }  
    public void moveHead(int x, int y) { /* ... */ }  
    public int getLength() { /* ... */ }  
    public int getPosX(int i) { /* ... */ }  
    public int getPosY(int i) { /* ... */ }  
    private int getPosXY(int i) { /* ... */ }  
}
```

7.2. Konstruktoren und Vererbung

- Die Konstruktoren einer Klasse werden nicht an die Unterklassen vererbt
- In einem Konstruktor kann mittels `super([<Parameterliste>])` ein Konstruktor der Oberklasse aufgerufen werden:

```
class Shape {  
    Shape(int color) { /*...*/ }  
    // ...  
}  
class Circle extends Shape {  
    Circle(double[] center, double radius, int color) {  
        super(color);    // ruft Konstruktor Shape(int color),  
        // ...           // muss erste Anweisung sein!  
    }  
    // ...  
}
```



7.2. Konstruktoren und Vererbung

- Vor der Ausführung eines Unterklassen-Konstruktors wird immer ein Konstruktor der Oberklasse ausgeführt
- falls kein expliziter Aufruf mit **super** erfolgt, wird der Default-Konstruktor der Oberklasse ausgeführt:

```
class A {  
    A() {  
        // Rumpf A ...  
    }  
}
```

```
class B extends A {  
    B(int i) {  
        // Rumpf B ...  
    }  
}
```

```
class C extends B {  
    C() { super(9);  
        // Rumpf C ...  
    }  
}
```

- Reihenfolge der *Konstruktoraufrufe* bei new C(): **C() → B(9) → A()**
- Reihenfolge der *Abarbeitung der Konstruktor-Rümpfe*: **A → B → C**

7.2. Konstruktoren und Vererbung

Die "Referenzvariable" **super**

- In Instanzmethoden abgeleiteter Klassen gibt es eine spezielle "Referenzvariable" **super**
- Sie erlaubt u.a. den Zugriff auf überschriebene Methoden der Basisklasse:

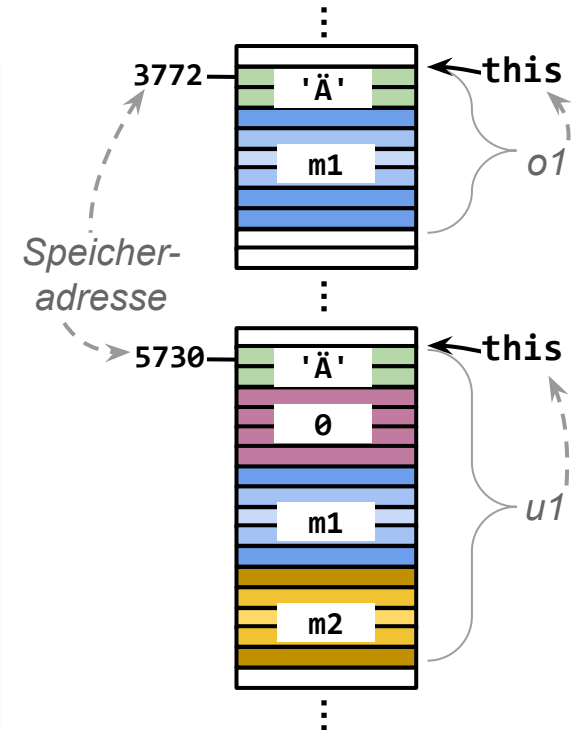
```
class Ober {  
    int op(int i) { /* ... */ }  
}  
class Unter extends Ober {  
    int op(int i) { return super.op(i)/2; /* Ruft op(i) in Klasse Ober */ }  
}
```

- Wie **this** muß auch **super** nicht deklariert werden
- Im Gegensatz zu **this** ist **super** aber keine Referenz auf ein reales Objekt

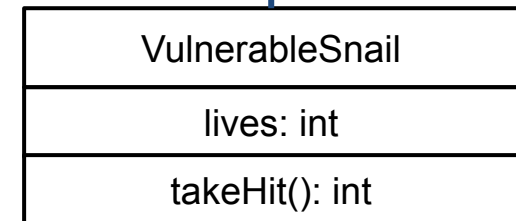
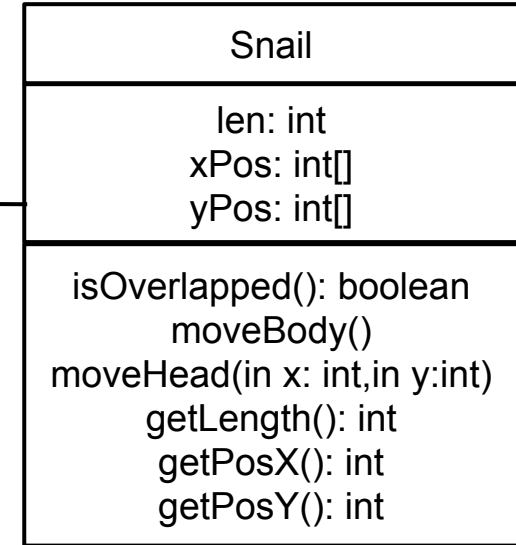
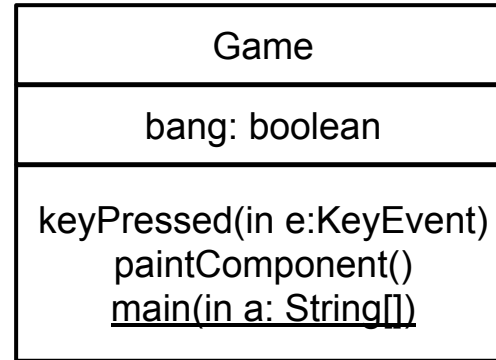
7.2. Konstruktoren und Vererbung

Beispiel `this` und `super`

```
class Ober {  
    protected char a1 = 'Ä';  
    public boolean m1(char a1) { return this.a1 == a1; }  
    public Ober() { a1 = '?'; }  
}  
  
class Unter extends Ober {  
    private int a2 = 0;  
    public boolean m1(char a1) { return a2 == (int) a1; }  
    public void m2() { this.a2 = (int) super.a1; }  
    public Unter() { super(); a2 = (int) '?'; }  
}  
  
// ...  
Ober o1 = new Ober();  
Unter u1 = new Unter();  
u1.m1('A');
```



Beispiel: SnailGame v2 ([github](#))



```
// Implementiere eine neue Klasse für eine Schnecke, die  
// 3 Leben hat, die nach jeder Kollision herunterzählen:  
class VulnerableSnail extends Snail {  
    private int lives = 3;  
    // Konstruktor mit super  
    public int takeHit() { /* ... */ }  
}
```

7.3. Polymorphie

- griech. "Vielgestaltigkeit"
- Eigenschaft eines Bezeichners (Operation, Funktion, Variable, ...), je nach Umgebung verschiedene Wirkung zu zeigen
- Verschiedene Arten der Polymorphie:
 - Überladen von Bezeichnern (z.B. '+' für `int`, `double`, `String`)
 - parametrisierbare Datentypen / Klassen (Typ als Parameter)
 - polymorphe *Funktionen* können Ergebnisse unterschiedlichen Typs liefern
 - polymorphe *Variable* können je nach Umgebung verschiedenartige Größen bezeichnen
- Referenzvariablen sind polymorphe Variablen
 - können auf Objekte unterschiedlicher Klassen verweisen

7.3. Polymorphie

```
import java.util.Random;

abstract class Tier {
    protected String gattung;
    public Tier(String gattung) { this.gattung = gattung; }
    public void print() {
        System.out.println("Tier der Gattung " + gattung + ".");
    }
}

class Hund extends Tier {
    protected String name, rasse;
    public Hund(String aName, String aRasse) {
        super("Hund"); name = aName; rasse = aRasse; }
    public void print() {
        System.out.println("Ich bin " + name + ", der " + rasse + ".");
    }
}
```

7.3. Polymorphie

```
class Katze extends Tier {  
    protected String name;  
    public Katze(String n) { super("Katze"); name = n; }  
    public void print() {  
        System.out.print("Ich bin " + name + ". "); super.print(); }  
}  
public class DemoPolymorphie {  
    public static void main(String[] args) {  
        Tier[] tiere = new Tier[3];  
        tiere[0] = new Hund("Waldi", "Dackel"); // tiere[] enthaelt verweise  
        tiere[1] = new Hund("Hasso", "Boxer"); // auf Objekte verschiedener  
        tiere[2] = new Katze("Chanty"); // Klassen (Hund, Katze)  
        Random rnd = new Random();  
        for (int i=0; i<5; i++) {  
            Tier t = tiere[rnd.nextInt(3)]; // t ist eine polymorphe Variable  
            t.print(); // print richtet sich nach dem  
                        // Objekt, auf das t verweist  
        }  
    }  
}
```

7.3. Polymorphie

Mögliche Ausgabe des Beispiels:

```
Ich bin Chanty.  
Ich bin ein Tier der Gattung Katze.  
Ich bin Hasso, der Boxer.  
Ich bin Waldi, der Dackel.  
Ich bin Chanty.  
Ich bin ein Tier der Gattung Katze.  
Ich bin Waldi, der Dackel.
```

7.3. Polymorphie

Binden von Bezeichnern

Bezeichner können zu unterschiedlichen Zeiten an Objekte, Datentypen oder Datenstrukturen gebunden werden:

- zur Übersetzungszeit:
statische Bindung, frühe Bindung
- zur Laufzeit:
dynamische Bindung, späte Bindung

```
// statische / fruehe Bindung:
```

```
Student t = new Student();  
System.out.println( t.getMatrNr() );
```

```
// dynamische / spaete Bindung:
```

```
// Array tiere[] enthaelt verweise:  
Tier[] tiere = new Tier[3];  
tiere[0] = new Hund("Waldi", "Dackel");  
tiere[1] = new Katze("Chanti");  
// t ist eine polymorphe Variable:  
Tier t = tiere[0];  
t.print(); // print -> Hund  
t = tiere[1];  
t.print(); // print -> Katze
```

7.3. Polymorphie

Binden von Bezeichnern

- **Methoden** werden in Java **dynamisch** gebunden
 - abhängig vom Objekt, für das die Methode aufgerufen wird (Polymorphie)
 - Merkregel: jedes Objekt kennt seine Methoden
- **Klassenmethoden** und **Attribute** werden **statisch** gebunden
- abhängig vom Typ der Referenzvariable, die für den Zugriff benutzt wird

7.3. Polymorphie -- Beispiel

```
interface I {  
    public int op(int i);  
}  
  
class Ober implements I {  
    public int attr = 1;  
    public static int kattr = 10;  
    public int op(int i) { return i+1; }  
    public static int kop() { return 4; }  
}  
  
class Unter extends Ober {  
    public int attr = 2;  
    public static int kattr = 20;  
    public int op(int i) { return i+2; }  
    public int op(double d) { return 7; }  
    public static int kop() { return 8; }  
}
```

```
I[] i = { new Ober(), new Unter() };  
i[0].op(1);      // = 2  
i[1].op(1);      // = 3  
  
int z;  
Ober o = new Ober();  
Unter u = new Unter();  
  
z = i[0].op(3);  // = 4  
z = i[1].op(4);  // = 6  
z = o.op(5.1);   // FEHLER  
z = u.op(6.2);   // = 7  
z = o.attr;      // = 1  
z = u.attr;      // = 2  
z = o.kop();     // = 4  
z = u.kop();     // = 8  
z = o.kattr;     // = 10  
z = u.kattr;     // = 20
```

7.3. Polymorphie

Konversion von Objekttypen

- Eine neue Klasse definiert in Java auch einen neuen Typ
 - z.B.: `Tier einTier;`
`Hund fido = new Hund("Fido", "Spaniel");`
`einTier` und `fido` sind Variablen verschiedenen Typs
- Java ist *typsicher*: Zuweisungen und Operationen sind nur mit Variablen bzw. Ausdrücken des korrekten Typs erlaubt
- Wir können aber:
`einTier = fido;`
schreiben, weil Java eine implizite Typkonversion von einer Klasse zu einer Oberklasse (bzw. implementierten Schnittstelle) macht

7.3. Polymorphie

Konversion von Objekttypen

- Explizite Typkonversionen (mit `einTier` und `fido` aus der letzte Folie):

```
einTier = (Tier)fido;           // korrekt, unnötig
fido = (Hund)einTier;          // korrekt
Student s = (Student)einTier;  // Compilerfehler
Katze pucki = (Katze)einTier;  // Laufzeitfehler
```

- Typkonversion bewegt sich in der Klassenhierarchie immer nur
 - aufwärts (implizite oder explizite Konversion), oder
 - abwärts (nur explizite Konversion)
- Test des Objekttyps über Operator `instanceof` möglich:

```
if (einTier instanceof Katze)
    Katze pucki = (Katze)einTier; // OK, wird nicht ausgeführt
```

7.3. Polymorphie

Beispiel -- Verteilung

```
import java.util.Random;

class Verteilung {
    protected int yLen = 24;
    protected int xLen = 80;
    protected int runs = 120;
    protected boolean[][] display = null;
    protected Random r;

    public Verteilung(int height, int width, int runs) {
        r = new Random();
        yLen = height; xLen = width; this.runs = runs;
        display = new boolean[xLen][yLen];
        for (int x = 0; x < xLen; x++)
            for (int y = 0; y < yLen; y++)
                display[x][y] = false;
    }
}
```

7.3. Polymorphie

Beispiel -- Verteilung

```
public void print() {  
    for (int y = 0; y < yLen; y++) {  
        System.out.print("|");  
        for (int x = 0; x < xLen; x++)  
            System.out.print( (display[x][y] == true)?"\u2589":" ");  
        System.out.println("|");  
    }  
}  
  
public static void main(String[] args) {  
    Verteilung v;  
    if ( (args.length > 0) && (args[0].equals("u")) ) {  
        v = new Uniform(24, 100, 240);  
    } else {  
        v = new Gauss(24, 100, 240);  
    }  
    v.print();  
}
```

7.3. Polymorphie

Beispiel -- Verteilung

```
class Uniform extends Verteilung {  
    public Uniform(int height, int width, int runs) {  
        super(height, width, runs);  
        for (int i = 0; i < runs; i++ ) {  
            int c = r.nextInt(height);  
            int j = 0;  
            while (display[j][c]) { j++; }  
            display[j][c] = true;  
        }  
    }  
}
```

7.3. Polymorphie

Beispiel -- Verteilung

```
class Gauss extends Verteilung {  
    public Gauss(int height, int width, int runs) {  
        super(height, width, runs);  
        for (int i = 0; i < runs; i++) {  
            int c = (int)( yLen/2 + r.nextGaussian() );  
            int j = 0;  
            while (display[j][c]) { j++; }  
            display[j][c] = true;  
        }  
    }  
}
```



7.4. Die universelle Klasse Object

- Java definiert eine Klasse **Object**, die an der Spitze jeder Klassenhierarchie steht
- Alle Klassen, die nicht explizit von einer anderen Klasse abgeleitet werden, sind (implizit) von **Object** abgeleitet: `public class Tier {...}` heißt damit `public class Tier extends Object { ...}`
- Die Klasse **Object** definiert 9 Methoden, die alle Klassen erben (direkt oder indirekt):

```
protected Object clone()  
Class<?> getClass()  
void notifyAll()
```

```
boolean equals(Object obj)  
int hashCode()  
void wait()
```

```
protected void finalize()  
void notify()  
String toString()
```

- diese Methoden können wie alle anderen auch überschrieben werden



7.4. Die universelle Klasse Object

- **public String toString()**
 - gibt eine textuelle Darstellung des **Object** zurück
 - Implementierung in **Object**: "Klassenname@Adresse"
- **public boolean equals(Object obj)**
 - stellt fest, ob zwei Objekte gleich sind
 - Implementierung in **Object**: `return this == obj;`
- **protected Object clone()**
 - erzeugt eine Kopie des Objekts
 - Implementierung in **Object** kopiert alle Instanzvariablen
- **protected void finalize()**
 - aufgerufen, bevor *Garbage Collector* **Object** löscht
 - Implementierung in **Object** tut nichts

```
class MyClass {  
    protected void finalize() {  
        System.out.println("Bye!");  
    }  
  
    public static void main(String[] s) {  
        MyClass me = new MyClass();  
        MyClass you = new MyClass();  
  
        // test toString:  
        System.out.println(me.toString());  
  
        // test equals:  
        boolean eq = me.equals(you);  
        System.out.println(eq);  
  
        // test finalize:  
        me = null;  
        you = null;  
        System.gc(); // garbage collector  
    }  
}
```

7.4. Die universelle Klasse Object

- `System.out.println()` kann beliebige Objekte ausdrucken, z.B.:

```
class Hund extends Tier {  
    // ...  
    public String toString() {  
        return "Ich bin " + name + ", der " + rasse + ".";  
    }  
}
```

```
Hund waldi = new Hund ("Waldi", "Dackel");  
System.out.println(waldi);
```

- weil es so implementiert wurde:

```
public void println(Object obj) {  
    String str = obj.toString(); // Polymorphie  
    println(str);               // Aufruf der überladenen Methode  
}
```

Objektorientierte und Formale Programmierung

-- Java Grundlagen --

8. Ausnahmebehandlung

Exceptions (Ausnahmen) signalisieren Fehler zur Laufzeit eines Programms

```
class ExceptTest {  
    public static void main(String[] args) {  
        int a = (int)(10*Math.random());  
        int b = (int)(10*Math.random());  
        System.out.println("a/b="+a/b);  
    }  
}
```

```
$ javac ExceptTest.java; java ExceptTest  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at ExceptTest.main (ExceptTest.java:5)
```

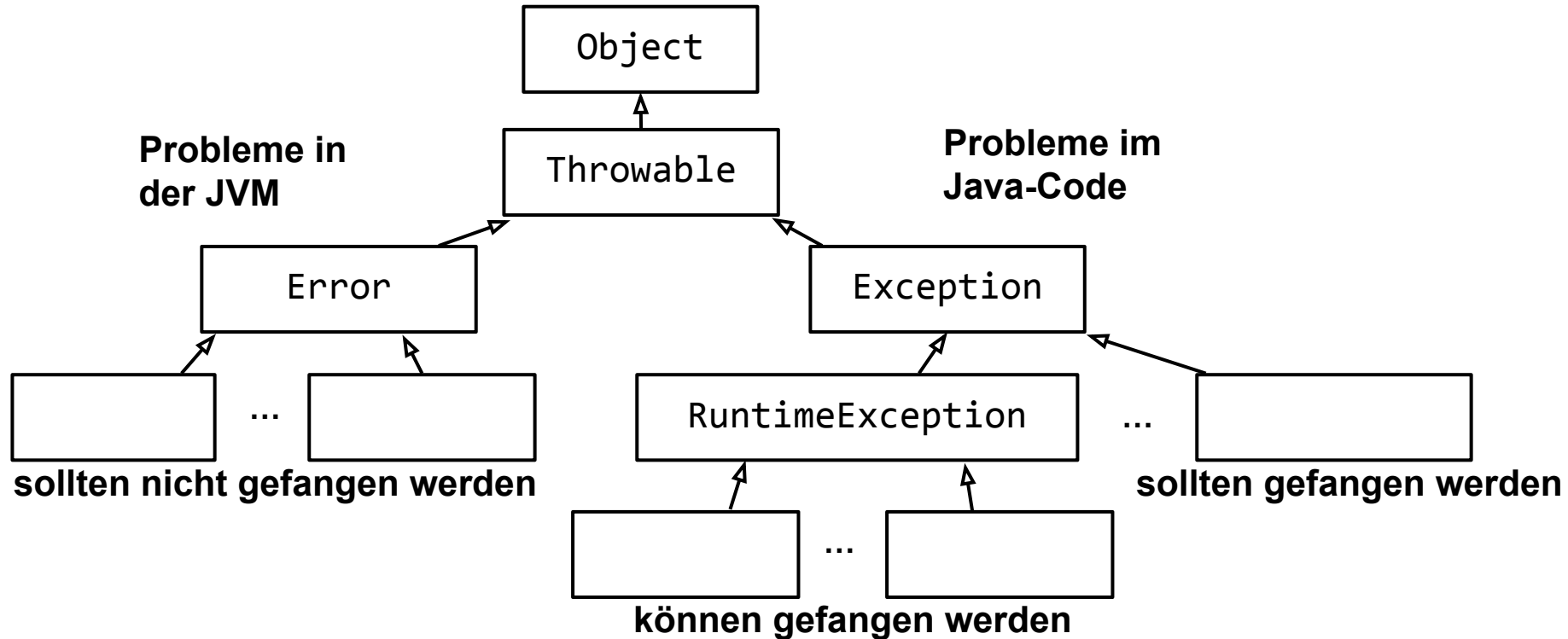
Exceptions (Ausnahmen) signalisieren Fehler zur Laufzeit eines Programms

```
class ExceptTest {  
    public static void main(String[] args) {  
        int a = (int)(10*Math.random());  
        int b = (int)(10*Math.random());  
        try {  
            System.out.println("a/b=" + (a/b));  
        }  
        catch (ArithmeticException e) {  
            System.out.println("a/b=?  -- Es gab ein Problem: " + e.getMessage());  
        }  
    }  
}
```

```
$ javac ExceptTest.java; java ExceptTest  
a/b=?  -- Es gab ein Problem: / by zero
```

- Exceptions (Ausnahmen) signalisieren Fehler zur Laufzeit eines Programms
- Sie können
 - implizit durch Java-Anweisungen (z.B. `x/0`, `a[-1]`)
 - explizit durch die Anweisung **throw** ausgelöst ("geworfen") werden
- Exceptions sind Ausnahmesituationen – sie sollten außerhalb des normalen Programmcodes behandelt ("gefangen") werden
 - höhere Übersichtlichkeit des Codes
- In Java sind Exceptions Objekte, die in einer Fehlersituation dynamisch erzeugt werden
 - ihre Attribute beschreiben den Fehler genauer

Exceptions: Objekthierarchie



Unterklasse RuntimeException

- **RuntimeExceptions** werden meist durch Fehler im Programmcode verursacht, sie müssen daher nicht behandelt werden ("Unchecked" von Java Compiler)
- Beispiele (siehe auch [Dokumentation zum Paket java.lang](#)):

ArithmeticException: z.B. `1/0` (ganzzahlig)

IndexOutOfBoundsException: z.B. `array[-1]`

NegativeArraySizeException: z.B. `new double[-5]`

NullPointerException:

z.B. `Hund meinHund = null; meinHund.print();`

ClassCastException:

z.B. `Tier t = new Hund("fido"); Katze k = (Katze)t;`

Definition eigener Exceptions → [github](#)

```
public class MyExcept extends Exception {  
    private int elNr = -1;  
  
    public MyExcept() {} // diese beiden Konstruktoren  
    public MyExcept(String s) { super(s); } // werden immer definiert  
    // extra Konstruktor mit Zusatzinformation zum Fehler:  
    public MyExcept(String s, int elNr) { super(s); this.elNr = elNr; }  
  
    public int getElementNr() { return elNr; }  
  
    public String toString() {  
        // getMessage() ist Methode der Oberklasse Throwable, liefert den String  
        // zurueck, der dem Konstruktor uebergeben wurde:  
        return "Eigener Fehler im Element " + elNr + ": " + getMessage();  
    }  
}
```

Definition eigener Exceptions

```
public class Vector {  
    double[] vector;  
    // ...  
  
    public void invert() throws MyExcept {  
        for (int i = 0; i < vector.length; i++) {  
            if (vector[i] == 0.0) {  
                throw new MyException("Divide by 0.", i);  
            }  
            vector[i] = 1.0 / vector[i];  
        }  
    }  
}
```

Werfen (**throw**) von Exceptions

- Exceptions können im Programm explizit durch die Anweisung **throw** ausgelöst werden:

```
public static double invert(double x) {  
    if (x == 0.0) throw new ArithmeticException("Divide by 0.");  
    return 1.0 / x;  
}
```

- throw** kann auch in einem catch-Block verwendet werden:

```
catch (Exception e) {  
    // ... Lokale Fehlerbehandlung ...  
    throw e; // Exception an Aufrufer weitergeben  
}
```

Behandlung von Exceptions

- Einfachster Fall: Exception wird nicht behandelt
 - Methode bricht beim Auftreten der Exception sofort ab
 - Exception wird an Aufrufer der Methode weitergegeben
 - Wenn die Exception nicht spätestens in der main-Methode gefangen wird → Abbruch des Programms
 - Jede Methode muss deklarieren, welche Exceptions sie werfen kann (ausgenommen **Error** und **RuntimeException**):

```
void anmelden(...) throws AnmeldungException
```

oder

```
public static void main(...) throws AnmeldungException, OtherException
```

Behandlung von Exceptions

- **try - catch - Block:**

```
try {  
    int z = zahlen[index];  
    int kehrwert = 1 / z;  
    // ...  
}  
catch (IndexOutOfBoundsException e) {  
    System.out.println("Unzulässiger Index");  
}  
catch (ArithmeticException e) {  
    System.out.println("Fehler: " + e.getMessage());  
}  
catch (Exception e) {  
    System.out.println(e);  
}
```

Behandlung von Exceptions

- Wenn Exception im **try**-Block auftritt:
 - **try**-Block wird verlassen
 - Erster "passender" **catch**-Block wird ausgeführt, Ausführung wird nach dem letzten catch-Block fortgesetzt
 - falls kein passender catch-Block vorhanden:
Abbruch der Methode, Weitergeben der Exception an Aufrufer
- Der **catch**-Block ist "passend" wenn das erzeugte Exception-Objekt an den Parameter des catch-Blocks zugewiesen werden kann.
Die erzeugte Exception ist identisch mit der spezifizierten Exception-Klasse oder ist eine Unterklasse davon

finally-Block

- Nach dem letzten **catch**-Block kann noch ein **finally**-Block angefügt werden
 - auch **try** - **finally** (ohne **catch**) ist erlaubt
- Die Anweisungen dieses Blocks werden immer nach Verlassen des **try**-Blocks ausgeführt, egal ob
 - der **try**-Block normal beendet wird
 - der Block (und die Methode) durch **return** verlassen wird
 - eine **Exception** auftritt und durch **catch** gefangen wird
 - hier: **finally** nach **catch** ausgeführt
 - eine **Exception** auftritt und an den Aufrufer weitergegeben wird
- Anwendung: "Aufräumarbeiten"
 - z.B. Löschen temporärer Dateien, Schließen von Fenstern, ...



Beispiel -- Wordle: Laden von Wörterdatei

```
int len = 0;
try { // load 5-letter words from a predefined file
    words = new String[WORDS_MAX];
    Scanner scanner = new Scanner(new File("words.txt"));
    while ( (scanner.hasNextLine()) && (len < WORDS_MAX) ) {
        words[len] = scanner.nextLine();
        len++;
    }
    scanner.close();
}
catch (java.io.IOException e) { // use backup dictionary
    len = 3;
    words = new String[len];
    words[0] = "worse"; words[1] = "worth"; words[2] = "words";
}
```


Beispiel -- Wordle: Laden von Wörterdatei (🌶️🌶️🌶️)

```
/** WordleClass: Implementieren Sie das Wordle-Spiel mit Wörterliste:
 */
import java.util.Scanner; // scan strings in der Konsole
import java.util.Random; // Random Number Generator
import java.io.File;

class WordleClass {
    /*...*/
    private String dialog( String hint ) { /*...*/ } // private Methode
    private void loadWords(String filename) { /*...*/ } //
    public void run() { /*...*/ } // public Methode

    public static void main( String[] str ) { // main ist eine Klassenmethode
        WordleClass w = new WordleClass();
        w.run();
    }
}
```

Beispiel -- WordleClass Lösung (1/4) → [github](#)

```
/** WordleClass: das komplette Wordle-Spiel mit Wörterliste
 */

import java.util.Scanner; // scan strings in der Konsole
import java.util.Random; // Random Number Generator
import java.io.File;

class WordleClass {
    private String loesung = "weary"; // default Loesunswort
    private String hint = "-----"; // Hinweis mit gefundenen Buchstaben
    private String[] words = null; // Wörterarray
    private static final int MAX_WORDS = 10000; // Maximum # Wörter

    public static String dialog( String hint ) { // Zeige Hinweis, Eingabe
        System.out.print( hint + ", what is your guess? ");
        Scanner scan = new Scanner(System.in);
        return scan.next();
    }
}
```

Beispiel -- WordleClass Lösung (2/4) → [github](#)

```
private void loadWords(String filename) {  
    int len = 0;  
    try { // lade Wörter aus filename (z.B. "words.txt"):  
        words = new String[MAX_WORDS];  
        Scanner scanner = new Scanner(new File(filename));  
        while ( (scanner.hasNextLine()) && (len<MAX_WORDS) ) {  
            words[len] = scanner.nextLine();  
            len++;  
        }  
        scanner.close();  
    }  
    catch (java.io.IOException e) { // backup Wörter falls Datei nicht besteht  
        len = 3;  
        words = new String[len];  
        words[0] = "worse";  
        words[1] = "worth";  
        words[2] = "words";  
    }  
}
```

Beispiel -- WordleClass Lösung (3/4) → [github](#)

```
public void run() {  
    String in;           // Eingabe Benutzer  
    char[] hintChar;     // Array von char fuer hint zu aendern  
    loadWords("words.txt");  
  
    do {  
        in = dialog( hint );  
        hintChar = hint.toCharArray();  
        if ( in.length() >= loesung.length() ) {  
            for (int i = 0; i < loesung.length(); i++ ) { // Kontrolliere einzelne  
                if ( in.charAt(i) == loesung.charAt(i) ) { // Buchstaben von in  
                    hintChar[i] = in.charAt(i);           // und zeige sie in hintChar  
                } else {                                   // wenn richtig  
                    boolean found = false;  
                    for (int j = 0; j < loesung.length(); j++ ) { // suche Buchstaben an  
                        if ( in.charAt(i) == loesung.charAt(j) ) { // andere Stellen in loesung  
                            found = true;  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

Beispiel -- WordleClass Lösung (4/4) → [github](#)

```
        hintChar[i] = (found?'?':'-');
    }
}
} else {
    System.out.println("word too short.");
}
hint = new String(hintChar);
} while ( ! in.equals( loesung ) );
}

public static void main( String[] str ) { // main ist eine Klassenmethode
    WordleClass w = new WordleClass();
    w.run();
}
}
```

Objektorientierte und Formale Programmierung

-- Java Grundlagen --

9. Dateien, Ströme und Serialisierung

9.1. Pakete der Java-Klassenbibliothek

- Die Sprache Java wird von einer (standardisierten) Klassenbibliothek ergänzt, mit tausenden Klassen in hunderten Paketen
- Häufig genutzte Pakete (importieren mit **import**, z.B.: `import java.awt.*;`):
 - **java.lang**: Klassen, die zum Kern der Sprache Java gehören (z.B.: **String**, **StringBuffer**, **Object**, **System**). Diese müssen nicht explizit importiert werden
 - **java.io**: Ein- und Ausgabe (Konsole und Dateien)
 - **java.util**: nützliche Hilfsklassen
u.a. Java Collection Framework (Container-Klassen)
 - **java.awt**: Elemente für Bedienoberflächen
 - **javax.swing**: verbesserte Bibliothek für Bedienoberflächen
 - **java.net**: Netzkommunikation
 - **java.sql**: Datenbank-Anbindung
 - **java.beans**: Komponentenmodell

9.2. Die Datenstruktur "Datei" (file)

- Eine Datei ist eine nach bestimmten Gesichtspunkten zusammengestellte Menge von Daten:

Datensatz 1				
Datensatz 2				
Datensatz 3				
...				

- Sie besteht aus einer Folge gleichartig aufgebauter Datensätze
 - ein Datensatz besteht aus mehreren Feldern unterschiedlichen Typs
 - die Anzahl der Datensätze muss nicht festgelegt werden
- Dateien werden i.d.R. dauerhaft auf Hintergrundspeichern gespeichert

9.2. Die Datenstruktur "Datei" (file)

Gängige Datei-Organisationen: Sequentielle Datei

- Daten sind fortlaufend gespeichert und können nur in dieser Reihenfolge gelesen werden
- es gibt ein **Dateifenster**, das bei jedem Lesen bzw. Schreiben um eine Position (einen Datensatz) weiter rückt
- es kann nur jeweils der Datensatz im Dateifenster gelesen bzw. geschrieben werden
- das Dateifenster kann zum Teil auch direkt positioniert werden

Adleman	1978	RSA
Diffie	1976	Key Exchange
Rivest	1978	RSA
Shamir	1978	RSA



9.2. Die Datenstruktur "Datei" (file)

Gängige Datei-Organisationen: Direkte Datei

- Zugriff auf Datensätze erfolgt über einen Schlüssel, aus dem direkt die Position in der Datei bestimmt wird

"Diffie"



Tabelle oder (Hash)-Algorithmus

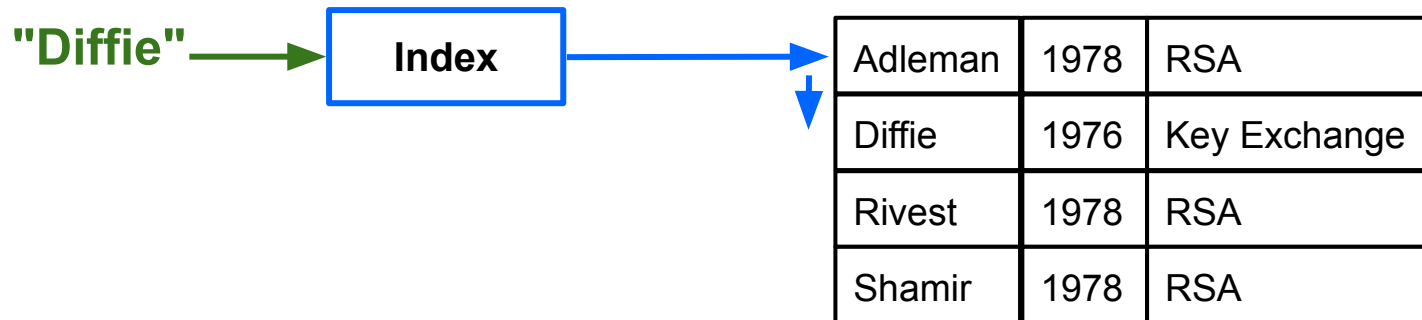


Adleman	1978	RSA
Diffie	1976	Key Exchange
Rivest	1978	RSA
Shamir	1978	RSA

9.2. Die Datenstruktur "Datei" (file)

Gängige Datei-Organisationen: Indexsequentielle Datei, Mischform

- Nutzung einer Tabelle (Index), die für einen Schlüssel in die Nähe des Datensatzes führt. Von dort aus sequentielle Suche



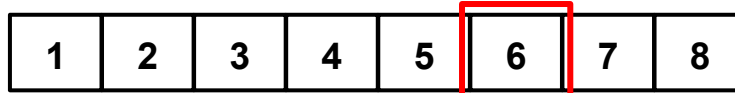
9.2. Die Datenstruktur "Datei" (file)

Das Dateimodell von Java

- Eine Datei in Java ist eine (unstrukturierte) Folge von Bytes. z.B. Textdatei: Folge von 8-Bit-Zeichen
- Nach dem Öffnen einer Datei verweist ein *Dateizeiger* auf das nächste zu lesende bzw. zu schreibende Byte
- Lese- und Schreiboperationen kopieren einen Datenblock aus der Datei bzw. in die Datei, der Dateizeiger wird entsprechend weitergeschoben
- Lesen über das Dateiende hinaus (End-of-file, EOF) ist nicht möglich
- Schreiben über das Dateiende führt zum *Anfügen* an die Datei
- Der Dateizeiger kann auch explizit positioniert werden

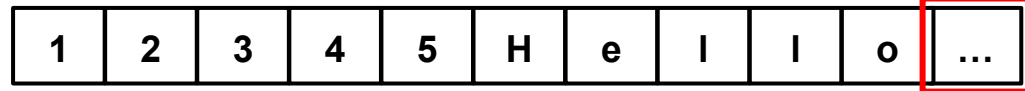
9.2. Die Datenstruktur "Datei" (file)

Das Dateimodell von Java, Beispiel: Schreiben in eine (Text-)Datei

Datei
(vorher)

↑ Dateizeiger

Datenblock

Schreibe Datenblock
in DateiDatei
(nachher)

↑ Dateizeiger

9.2. Die Datenstruktur "Datei" (file)

Grundoperationen auf Dateien

- öffnen (**open**) einer durch ihren Namen gegebenen Datei
 - zum Lesen: Dateizeiger wird auf Anfang positioniert
 - zum Schreiben: Dateizeiger wird auf Anfang bzw. Ende positioniert (überschreiben der Datei bzw. Anfügen)
 - i.d.R. wird beim Öffnen auch ein Dateipuffer eingerichtet
 - speichert einen Teil der Datei im Hauptspeicher zwischen
 - verhindert, dass jede Datei-Operation sofort auf dem langsamen Hintergrundspeicher ausgeführt werden muss
- Schließen (**close**) einer geöffneten Datei
 - Dateien sollten nach Verwendung immer geschlossen werden
 - sonst evtl. Datenverlust: Zurückschreiben des Dateipuffers
 - nach dem Schließen sind keine Operationen mehr zulässig

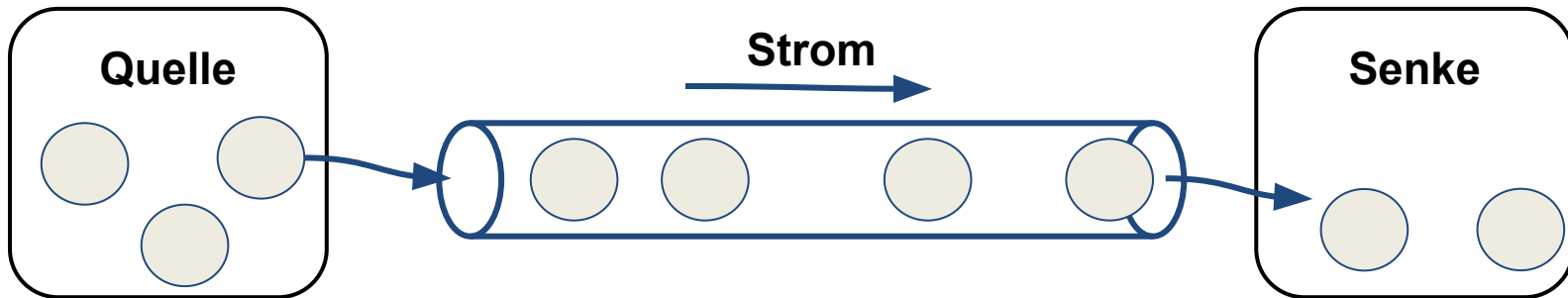
9.2. Die Datenstruktur "Datei" (file)

Grundoperationen auf Dateien

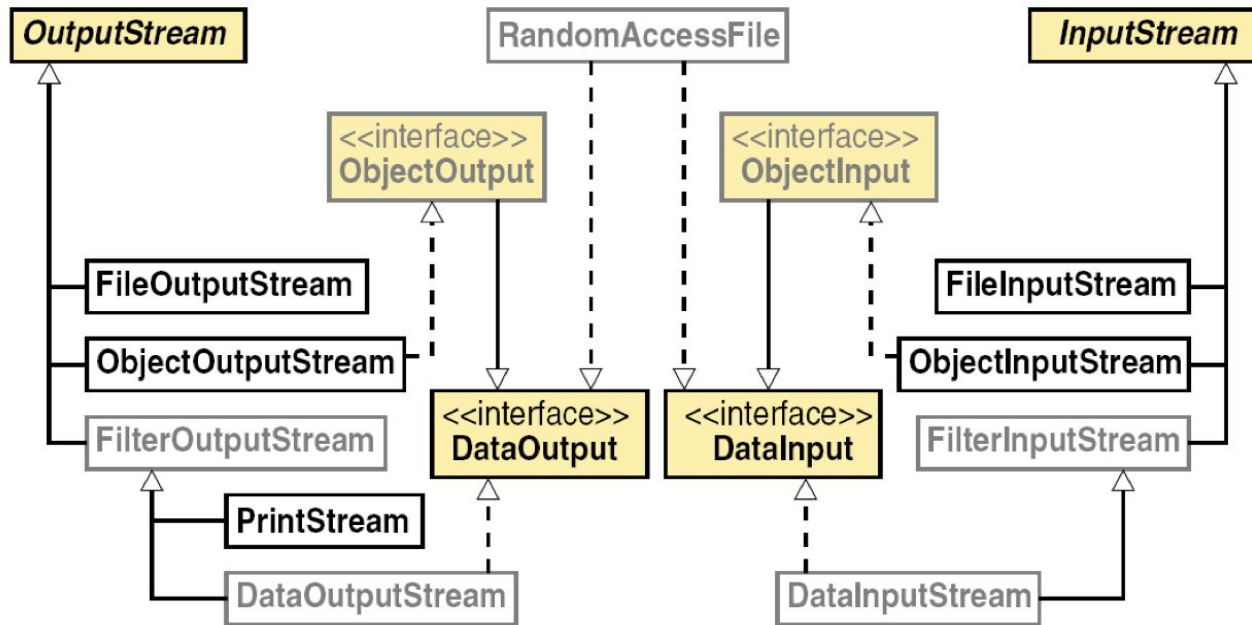
- **lesen (read)** eines Datenblocks
 - die Daten ab dem Dateizeiger werden in eine Variable (z.B. Byte-Array) kopiert
 - Dateizeiger wird entsprechend weiterbewegt
- **schreiben (write)** eines Datenblocks
 - Inhalt einer Variable (z.B. Byte-Array) wird ab dem Dateizeiger in die Datei kopiert (ggf. angefügt)
 - Dateizeiger wird entsprechend weiterbewegt
- **flush**: Leeren des Dateipuffers
 - Inhalt des Dateipuffers wird in die Datei zurückgeschrieben
- **seek**: explizites Positionieren des Dateizeigers
 - ermöglicht wahlfreien Zugriff auf die Datei

9.3. Ein- und Ausgabe mit Strömen (Streams)

- In Java erfolgt jede Ein-/Ausgabe (auch in Dateien) über Ströme. Sie stellen die Schnittstelle des Programms nach außen dar
- Ein Strom ist eine geordnete Folge von Daten mit einer Quelle und einer Senke
 - Ströme sind i.d.R. unidirektional (entweder Ein- oder Ausgabe)
 - ein Strom puffert die Daten so lange, bis sie von der Senke entnommen werden (Warteschlange)



9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Strom-Klassen / Schnittstellen im Paket `java.io`

9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Strom-Klassen / Schnittstellen im Paket `java.io`

- Abstrakte Basisklassen: **`InputStream`**, **`OutputStream`**
allgemeine Ströme für Ein- bzw. Ausgabe
- Dateiströme: **`FileInputStream`**, **`FileOutputStream`**
spezielle Ströme für die Ein-/Ausgabe auf Dateien
- Serialisierung mittels **`ObjectInputStream`**, **`ObjectOutputStream`**
- Bidirektionaler Dateistrom: **`RandomAccessFile`**
ermöglicht zusätzlich Positionieren des Dateizeigers
- Filterströme: **`FilterInputStream`**, **`FilterOutputStream`**
 - erhalten Daten von einem anderen Strom und filtern diese bzw. geben gefilterte Daten an einen anderen Strom weiter
 - Filterung: z.B. Umwandlung von Datentypen in Byteströme

9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Strom-Klassen / Schnittstellen im Paket `java.io`

- Schnittstellen **`DataInput`**, **`DataOutput`**
 - definieren Operationen zur Ein-/Ausgabe von einfachen Datentypen (**`int`**, **`double`**, ...) und **`Strings`**
 - implementiert von den Filterströmen **`DataInputStream`**, **`DataOutputStream`**
- Schnittstellen **`ObjectInput`**, **`ObjectOutput`**
 - definieren Operationen zur Ein-/Ausgabe von Objekten
 - implementiert von den Strömen **`ObjectInputStream`**, **`ObjectOutputStream`**
- Strom zur formatierten Text-Ausgabe von Daten: **`PrintStream`**

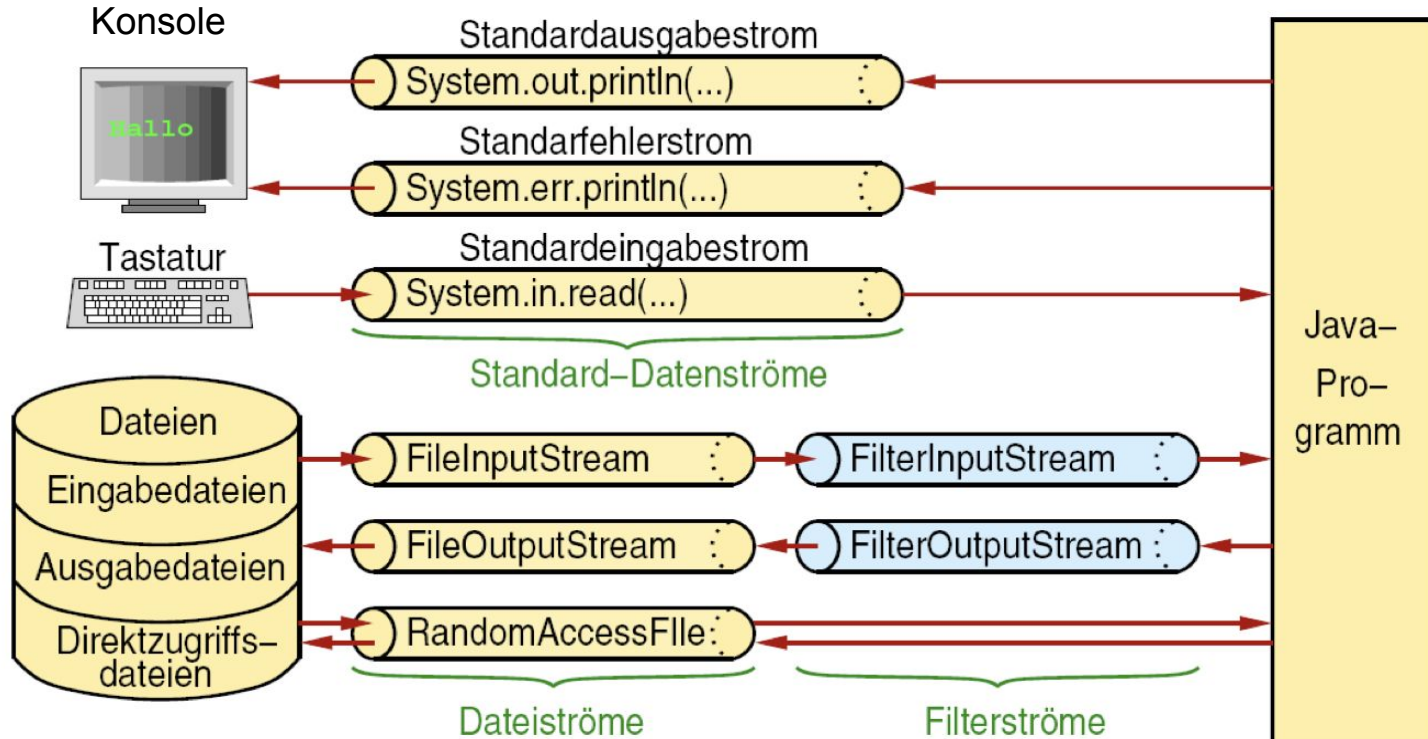
9.3. Ein- und Ausgabe mit Strömen (Streams)

Standard-Datenströme

Java definiert drei Standard-Datenströme für die Ein-/Ausgabe von / zur Konsole:

- **InputStream** `System.in` zum Einlesen von Zeichen von der Tastatur, z.B. `char ch = (char) System.in.read();`
- **PrintStream** `System.out` zur Ausgabe von Zeichen auf den Bildschirm, z.B. `System.out.println("Hallo");`
- **PrintStream** `System.err` zur Ausgabe von Zeichen auf den Bildschirm, speziell für Fehlermeldungen

9.3. Ein- und Ausgabe mit Strömen (Streams)



9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Operationen der Klasse `InputStream`

- **`abstract int read() throws IOException`**
 - liest ein Byte (0 ... 255) aus dem Strom
 - blockiert, falls keine Eingabe verfügbar ist
 - am Stromende (z.B. Dateiende) wird -1 zurückgegeben
- **`int read(byte[] buf) throws IOException`**
 - liest bis zu `buf.length` Bytes aus dem Strom
 - blockiert, bis eine Eingabe verfügbar ist
 - Ergebnis: Zahl der gelesenen Bytes bzw. -1 am Stromende
- **`void close() throws IOException`**
 - schließt den Strom: Freigabe belegter Ressourcen

9.3. Ein- und Ausgabe mit Strömen (Streams)

Beispiel: Bytes im Eingabestrom zählen (🌶️) → [github](#)

```
import java.io.*;

public class Count {
    // IOException wird nicht gefangen, dies muß deklariert werden:
    public static void main(String[] args) throws IOException {
        int count = 0;
        // Zeichen einlesen mit System.in.read(), bis Stromende (Strg-D):

        String msg = "  Eingabe hatte " + count + " Bytes\n";
        // Nur zur Demonstration. Ausgabe als Bytearray via write():
        System.out.write(msg.getBytes());
    }
}
```

9.3. Ein- und Ausgabe mit Strömen (Streams)

Wichtige Operationen der Klasse **OutputStream**

- **abstract void write(int b) throws IOException**
 - schreibt das Byte b (0 ... 255) in den Strom
 - (nur die unteren 8 Bit von b sind relevant)
- **void write(byte[] buf) throws IOException**
 - schreibt die Bytes aus buf in den Strom
- **void flush() throws IOException**
 - leert den Puffer des Stroms
 - alle noch im Puffer stehenden Bytes werden z.B. auf den Bildschirm oder in die Datei geschrieben
- **void close() throws IOException**
 - schließt den Strom: Freigabe belegter Ressourcen

9.3. Ein- und Ausgabe mit Strömen (Streams)

Dateiströme

- **`FileInputStream(String path)` throws `FileNotFoundException`**
öffnet Datei mit angegebenem Namen zum Lesen
- **`FileOutputStream(String path)` throws `FileNotFoundException`**
 - öffnet Datei mit angegebenem Namen zum Schreiben
 - Datei wird ggf. neu erzeugt
- Operationen werden von `InputStream` bzw. `OutputStream` geerbt und teilweise mit neuen Implementierungen überschrieben

9.3. Ein- und Ausgabe mit Strömen (Streams)

Dateiströme Beispiel: Datei Kopieren (1/2) () → [github](#)

```
class Copy {  
    public static void copyFile(String from, String to) throws IOException {  
        // Ein- und Ausgabedateien öffnen:  
        FileInputStream in = new FileInputStream(from);  
        FileOutputStream out = new FileOutputStream(to);  
        // Datei byteweise kopieren mit read() und write():  
  
        // Dateien schließen:  
        in.close(); out.close();  
    }  
}
```

9.3. Ein- und Ausgabe mit Strömen (Streams)

Dateiströme Beispiel: Datei Kopieren (2/2) () → [github](#)

```
// Aufruf: java Copy <Eingabedatei> <Ausgabedatei>
public static void main(String[] args) {
    if (args.length != 2) {
        System.err.println("Programm benötigt 2 Argumente: " +
                           "<Eingabedatei> <Ausgabedatei> ");
        return;
    }
    try {
        copyFile( args[0], args[1] );
    }
    catch (IOException e) {
        System.err.println("Fehler beim Kopieren: " + e );
    }
}
```

9.3. Ein- und Ausgabe mit Strömen (Streams)

Beispiel Dateien / Strömen: TextLogger.java () → [github](#)

```
import java.io.*;
class TextLogger {
    public static void main(String[] args) {
        // IOException soll in main gefangen werden:

        FileOutputStream out = new FileOutputStream("log.txt"); // Ausgabe
        // Zeichen einlesen bis Stromende (^D), schreibe jedes Zeichen
        // via out in die Ausgabedatei:

        // Datei schließen:

    }
}
```

9.4. Serialisierung von Objekten

- Ziel: einmal erzeugte Objekte sollen auch über das Ende des Programms hinaus gespeichert bleiben ("Lightweight Persistence")
- Persistenz: Langfristige Speicherung von Objekten mit ihren Zuständen und Beziehungen, so dass ein analoger Zustand im Arbeitsspeicher wiederhergestellt werden kann
- Serialisierung: Umwandlung des Zustands eines Objekts in einen *Byte-Strom* bzw. umgekehrt (Deserialisierung)
 - der Byte-Strom lässt sich dann in eine Datei ausgeben bzw. von dort wieder einlesen
 - das Objekt kann dabei Referenzen auf Arrays und andere Objekte enthalten, die automatisch mit serialisiert werden

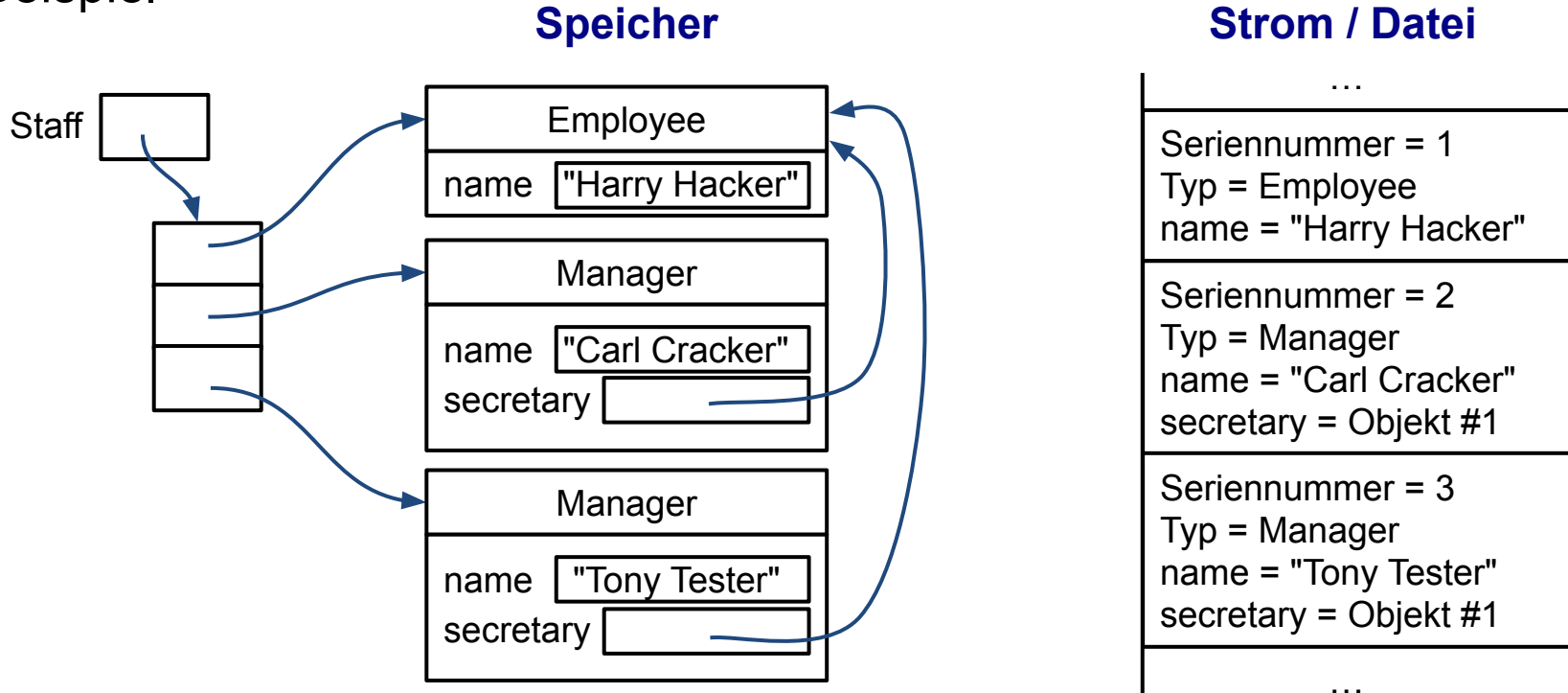
9.4. Serialisierung von Objekten

Was geschieht bei der Serialisierung eines Objekts / Arrays?

1. Erzeuge eine eindeutige Seriennummer für das Objekt und schreibe diese in den Strom
2. Schreibe Information zur Klasse in den Strom, u.a. Klassenname, Attributnamen und -typen
3. Für alle Attribute des Objektes (bzw. Elemente des Arrays):
 - falls keine Referenz: schreibe den Wert in den Strom
 - sonst: Z = Ziel der Referenz
 - falls Z noch nicht in diesen Strom serialisiert wurde:
 - serialisiere Z (Rekursion)
 - sonst: schreibe die Seriennummer von Z in den Strom

9.4. Serialisierung von Objekten

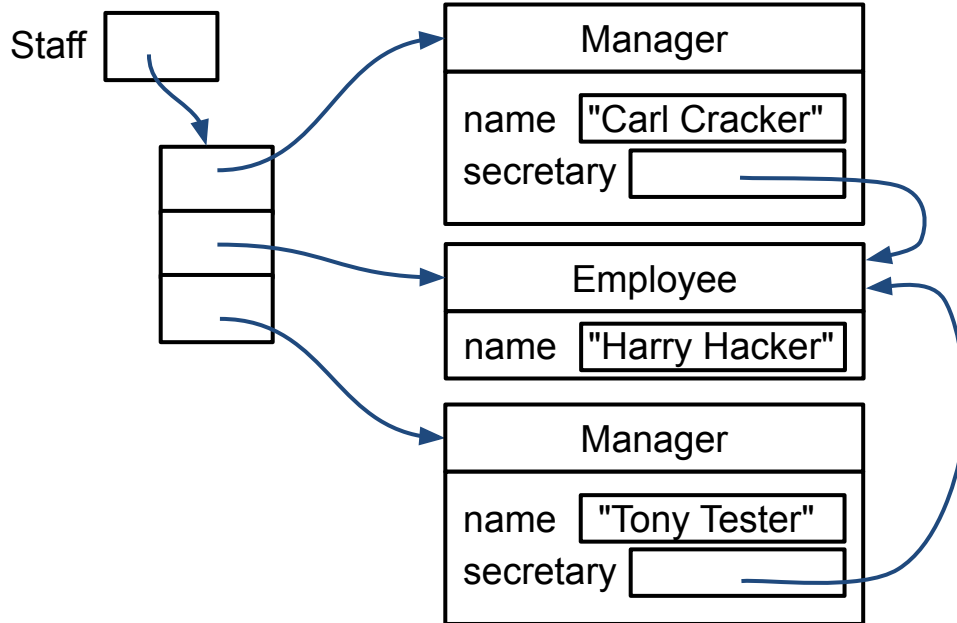
Beispiel



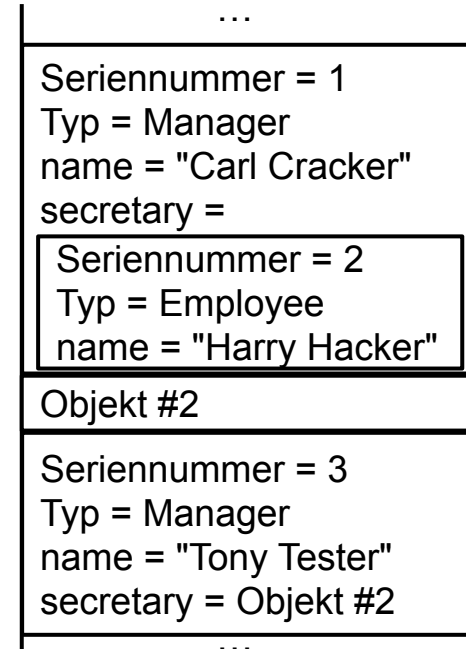
9.4. Serialisierung von Objekten

Beispiel

Speicher



Strom / Datei



9.4. Serialisierung von Objekten

Voraussetzung für die Serialisierbarkeit von Objekten

1. Die Klasse muß die Schnittstelle **Serializable** implementieren
 - **Serializable** besitzt weder Methoden noch Attribute (Interface **Serializable** in **java.io**),
 - die Schnittstelle dient nur der Markierung einer Klasse als serialisierbar (*Marker-Interface*)
2. Zudem müssen alle Referenzen in dem Objekt wieder auf serialisierbare Objekte verweisen:

```
class Person implements Serializable {  
    private String name; // String ist serialisierbar  
    private Address adresse;  
}  
class Address implements Serializable { // ...
```

9.4. Serialisierung von Objekten

Die Klasse **ObjectOutputStream**

- Realisiert die Serialisierung von Objekten
- Konstruktor: **ObjectOutputStream(OutputStream out) throws IOException**
- **void writeObject(Object obj) throws IOException**
serialisiert **obj** in den Ausgabestrom
- **void reset() throws IOException**
löscht alle Information darüber, welche Objekte bereits in den Strom geschrieben wurden.
Nachfolgendes **writeObject()** schreibt Objekte erneut in den Strom
- zusätzlich: alle Methoden der Schnittstelle **DataOutput**

9.4. Serialisierung von Objekten

Was schreibt `writeObject(obj)` in den Ausgabestrom?

- Falls **obj** noch nicht in den Strom geschrieben wurde:
 - neben **obj** werden auch alle von **obj** aus erreichbaren Objekte serialisiert
 - es wird also immer ein ganzer Objekt-Graph serialisiert
 - **obj** heißt Wurzelobjekt des Objekt-Graphen
 - die Referenzen zwischen den Objekten werden bei der Deserialisierung automatisch wiederhergestellt
- Falls **obj** bereits in den Strom geschrieben wurde (und kein `reset()` ausgeführt wurde), wird nur ein "Verweis" (Seriennummer) auf das schon im Strom befindliche Objekt geschrieben

9.4. Serialisierung von Objekten

Die Klasse **ObjectInputStream**

- Konstruktor: **ObjectInputStream(InputStream in) throws IOException**
- **Object readObject() throws IOException**
liest das nächste Objekt aus dem Eingabestrom
 - falls Objekt bereits vorher gelesen wurde (ohne **reset()**):
Ergebnis ist Referenz auf das schon existierende Objekt
 - sonst: Objekt und alle in Beziehung stehenden Objekte lesen
 - Objekte werden neu erzeugt, besitzen denselben Zustand und dieselben Beziehungen wie die geschriebenen Objekte
 - Ergebnis ist Referenz auf das Wurzelobjekt
 - i.d.R. explizite Typkonversion des Ergebnisses notwendig
- zusätzlich: alle Methoden der Schnittstelle **DataInput**

9.4. Serialisierung von Objekten

Die Schnittstellen `DataOutput` und `DataInput`

- Einige Methoden von `DataOutput`:
 - **`void writeInt(int v) throws IOException`**
schreibt ganze Zahl in den Strom (in Binärform: 4 Bytes)
 - **`void writeDouble(double v) throws IOException`**
schreibt Gleitkomma-Zahl (in Binärform: 8 Bytes)
- Einige Methoden von `DataInput`:
 - **`int readInt() throws IOException`**
 - **`double readDouble() throws IOException`**
 - bei Leseversuch am Dateiende: **`EOFException`**

9.4. Serialisierung von Objekten

Beispiel: Studentendatei (1/3)

```
import java.io.*;
class Name implements Serializable {
    String name;
    String vorname;
    public Name(String n, String vn) { name = n; vorname = vn; }
}

class Student implements Serializable {
    Name name;
    int matrNr;
    double note;
    public Student(String n, String vn, int mn) {
        name = new Name(n, vn); matrNr = mn;
    }
    public void setNote(double note) { this.note = note; }
```

9.4. Serialisierung von Objekten

Beispiel: Studentendatei (2/3)

```
public static void main(String[] args) {  
    ObjectOutputStream oos = null;  
    try {  
        Student s = new Student("Hugo", "Test", 12345678);  
        oos = new ObjectOutputStream( new FileOutputStream("out.ser") );  
        oos.writeObject(s); // Schreibe Objekt s  
        s.setNote(3.7);  
        oos.reset(); // sonst wird nur eine weitere Referenz geschrieben  
        oos.writeObject(s); // Schreibe Objekt s nochmal  
    }  
    catch (/*...*/) {/*...*/}  
    finally { /*...*/ oos.close(); }
```

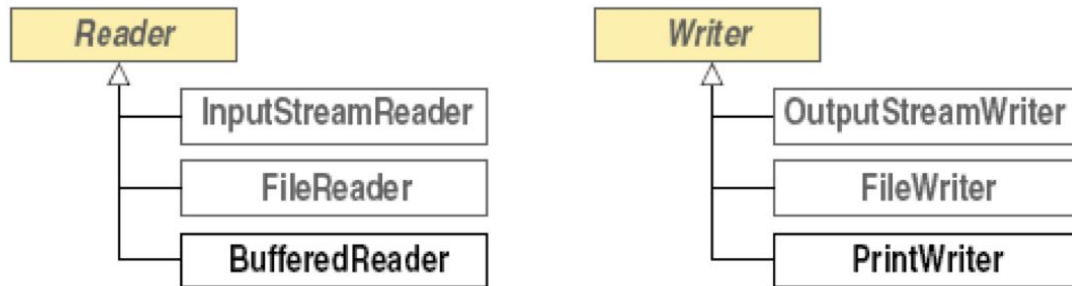
9.4. Serialisierung von Objekten

Beispiel: Studentendatei (3/3)

```
ObjectInputStream ois = null;
try {
    ois = new ObjectInputStream( new FileInputStream("out.ser") );
    // Objekte von Datei einlesen und Typ umwandeln
    Student s1 = (Student) ois.readObject();
    Student s2 = (Student) ois.readObject();
    System.out.println(s1);
    System.out.println(s2);
    System.out.println(s1 == s2); // Was wird hier ausgegeben?
}
catch (/*...*/) { /*...*/ }
finally { /*...*/ ois.close(); }
}
```


9.5. Formatierte Text-Ein-/Ausgabe

- Java-Ströme arbeiten byte-orientiert, nicht zeichen-orientiert
- d.h. Daten werden im Strom binär übertragen, nicht als Text
- Für die text-basierte Ein-/Ausgabe stellt Java zusätzliche Klassen zur Verfügung, u.a.: **Reader** und **Writer** stellen Basis-Methoden für die zeichenweise Ein-/Ausgabe zur Verfügung



9.5. Formatierte Text-Ein-/Ausgabe

Beispiel (1/4)

```
import java.io.*;
class Student {
    String name, vorname;
    int matrNr;    double note;

    public Student(String name, String vorname, int matrNr) {
        this.name=name;    this.vorname = vorname;    this.matrNr = matrNr;
    }
    public Student(BufferedReader reader) throws IOException {
        try {
            String line = reader.readLine();
            String[] fields = line.split(",");
            name = fields[0];    vorname = fields[1];
            matrNr = Integer.parseInt(fields[2]);
            note = Double.parseDouble(fields[3]);
        }
    }
}
```

9.5. Formatierte Text-Ein-/Ausgabe

Beispiel (2/4)

```
    catch (NullPointerException e) {  
        throw new IOException("Unerwartetes Dateiende");  
    }  
    catch (NumberFormatException e) {  
        throw new IOException("Falsches Elementformat");  
    }  
    catch (IndexOutOfBoundsException e) {  
        throw new IOException("Zu wenig Datenelemente");  
    }  
}  
  
public void writeToStream(PrintWriter pw) {  
    pw.println(name + "," + vorname + "," + matrNr + "," + note);  
    pw.flush();  
}
```

9.5. Formatierte Text-Ein-/Ausgabe

Beispiel (3/4)

```
public static void main(String[] args) {  
    PrintWriter pw = null;  
    try {  
        Student s = new Student("Hugo", "Test", 12345678);  
        s.writeToStream(new PrintWriter(System.out));  
        pw = new PrintWriter( new FileWriter("out.txt") );  
        s.writeToStream(pw);  
    }  
    catch (FileNotFoundException e) { /*...*/ }  
    catch (IOException e) { /*...*/ }  
    finally {  
        if (pw != null) pw.close();    // Keine IOException  
    }  
}
```

9.5. Formatierte Text-Ein-/Ausgabe

Beispiel (4/4)

```
BufferedReader reader = null;
try {
    reader = new BufferedReader( new FileReader("out.txt") );
    Student s1 = new Student(reader);
    System.out.println("Name,Vorname,matrNr,Note:");
    Student s3 = new Student(new BufferedReader(
                                new InputStreamReader(System.in) ) );

    System.out.println(s1);
    System.out.println(s3);
}
catch (FileNotFoundException e) { /*...*/ }
catch (IOException e) { /*...*/ }
finally {
    try { reader.close(); } catch (Exception e) { /*...*/ }
}
```

Objektorientierte und Formale Programmierung

-- Java Grundlagen --

10. Threads und Sockets

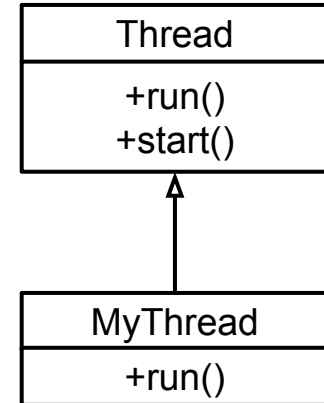
10.1. Threads

- Unsere bisherigen Programme arbeiteten rein sequentiell, die Anweisungen wurden eine nach der anderen ausgeführt
- Manchmal sollte ein Programm aber auch nebenläufig arbeiten können, d.h. mehrere Dinge (scheinbar) gleichzeitig tun, z.B.:
 - Ausgabe von Bild und Ton einer Multimedia-Anwendung
 - gleichzeitige Darstellung mehrerer Animationen
 - Bearbeitung längerer Aufgaben (z.B. Drucken) im "Hintergrund", während mit dem Programm weitergearbeitet wird
- Ein Thread ist eine Aktivität (d.h. Ausführung von Programmcode), die nebenläufig zu anderen Aktivitäten ausgeführt wird. Alle Threads einer Programmausführung arbeiten dabei auf denselben Daten

10.1. Threads

Threads in Java

- Ein Java-Programm startet immer mit genau einem Thread (main-Thread), der die Methode **main()** abarbeitet
- Für weitere Threads steht die Klasse **Thread** zur Verfügung, von dieser Klasse muß eine Unterklasse definiert werden
- Die wichtigsten Operationen von Thread sind:
 - **void run()**: wird beim Start des Threads ausgeführt
 - muß in der Unterklasse überschrieben werden mit dem Code, den der Thread ausführen soll
 - der Thread endet, wenn **run()** zurückkehrt
 - **void start()**: startet den Thread
 - **start()** kehrt sofort zum Aufrufer zurück
 - der Thread führt nebenläufig seine **run()**-Methode aus





10.1. Threads

Beispiel WorkerThread (🌶️) : Berechnung im Hintergrund

```
class WorkerThread {  
    public static void main(String[] s) {  
        // Erstellen Sie ein Array von Threads, die gleichzeitig ausgeführt werden:  
        Thread t = new WorkThread(); // Erzeuge ein neues Thread-Objekt  
        t.start(); // Führe die run-Methode des Objekts in einem neuen Thread aus  
    }  
}  
  
class WorkThread extends Thread {  
    public void run() { // wird nebenläufig zum Aufrufer ausgeführt  
        System.out.println("Working ...\n");  
        double v = 1.000000001;  
        for (double i=0; i<5000000000.0; i++) { v *= v; } // komplexe Berechnung  
        System.out.println("Done!\n");  
    }  
}
```

10.1. Threads

Synchronisation von Threads

- Eine Hintergrund-Berechnung wie im Beispiel ist nur möglich, wenn das Ergebnis nicht zum Weiterarbeiten benötigt wird
- Andernfalls kann mit der Methode `join()` auf das Ende des Threads gewartet werden, wenn das Ergebnis gebraucht wird
Dieses kann / muß in Attributen des Thread-Objekts gespeichert werden
- In vielen Fällen ist auch eine weitergehende Synchronisation der Threads erforderlich:
 - wechselseitiger Ausschluß von Methoden:
verhindert gleichzeitige Ausführung durch mehrere Threads
 - Warten auf Ereignisse, die andere Threads auslösen



10.1. Threads

Beispiel WorkerThreadJoin (🌶️🌶️)

```
class WorkerThreadJoin {  
    public static void main(String[] s) {  
        // Erstellen Sie ein Array von Threads, die gleichzeitig ausgeführt werden  
        // und benutzen Sie join() für jedes Thread:  
  
    }  
}  
  
class WorkThread extends Thread {  
    public void run() { // wird nebenläufig zum Aufrufer ausgeführt  
        System.out.println("Working ...\n");  
        double v = 1.000000001;  
        for (double i=0; i<5000000000.0; i++) { v *= v; } // komplexe Berechnung  
        System.out.println("Done!\n");  
    }  
}
```

10.1. Threads

Beispiel: Bankkonto (1/2)

```
class Konto {  
    public Konto(double saldo) { this.saldo = saldo; }  
    public double getSaldo() { return saldo; }  
    public boolean abheben(double betrag) {  
        double neuerSaldo = getSaldo() - betrag;  
        boolean ok = true;  
        if (neuerSaldo < 0) {  
            // Bei Ueberziehung: Anfrage an Schufa (über Netzwerk)  
            ok = frageSchufa(neuerSaldo); // kann dauern ...  
        }  
        if (ok)  
            saldo = neuerSaldo; // Buchung durchführen  
        return ok;  
    }  
    private double saldo = 0.0;  
}
```

10.1. Threads

Beispiel: Bankkonto (2/2)

```
class Banking extends Thread {
    Konto konto; double betrag;    // Eingabedaten für den Thread
    Banking(Konto k, double b) { konto = k; betrag = b; }
    public void run() {
        konto.abheben(betrag);
        System.out.println("Kontostand: " + konto.getSaldo());
    }
}

class Bankkonto {
    public static void main(String args[]) {
        Konto konto = new Konto(10);    // Konto mit 10 EUR
        for (int i=0; i<3; i++) {    // dreimal 10 EUR abheben
            Banking t = new Banking(konto, 10); t.start();
        }
    }
}
```



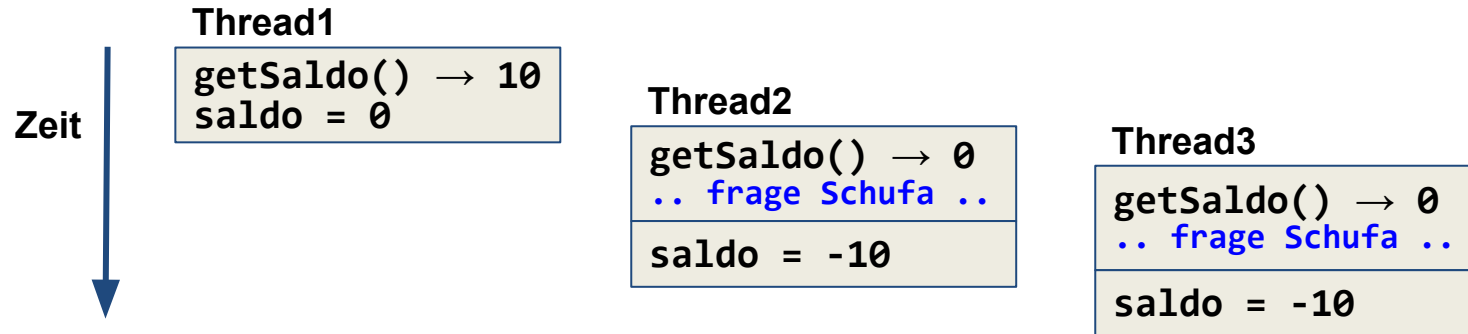
10.1. Threads

Beispiel: Bankkonto Ausgabe

```
Kontostand: 0.0  
Kontostand: -10.0  
Kontostand: -10.0
```

Warum hier zweimal derselbe Kontostand?

Problem: zeitliche Verzahnung in **abheben()**:



Lösung: wechselseitiger Ausschluß für **abheben()**

10.1. Threads

Synchronisation mit Monitor

- Kritischer Bereich ist ein Programmteil, der nur von einem Prozess zur gleichen Zeit durchlaufen werden darf
- Monitor dient zur Kapselung eines kritischen Bereichs und zur Synchronisation nebenläufiger Prozesse
- Sperre („Lock“)
 - Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen zurückgenommen
 - Sperre beim Eintritt in den Monitor von anderem Prozess gesetzt, so muss der aktuelle Prozess warten
 - Freigabe der Sperre beim Verlassen des Monitors

10.1. Threads

Synchronisation mit Monitor

- Nutzung des Monitors mit Hilfe des Schlüsselworts `synchronized`
 - Schutz einer kompletten Methode durch Sperre mit `this`-Pointer
 - Schutz eines Codeabschnitts durch Angabe einer Objektvariable

```
public synchronized void method() {  
    // anweisungen  
}
```

```
synchronized(einObjekt) {  
    // anweisungen  
}
```

```
public void method() {  
    synchronized(this) {  
        // anweisungen  
    }  
}
```


10.1. Threads

Beispiel Monitor: Hochzählen eines gemeinsamen Zählers

```
public class Listing extends Thread {
    static int cnt = 0;
    public static void main(String[] args) {
        Thread t1 = new Listing();
        Thread t2 = new Listing();
        t1.start();
        t2.start();
    }
    public void run() {
        while (true) System.out.print(cnt++ " ");
    }
}
```

Mögliche Ausgabe: 1 2 3 4 5 7 8 9 10 6 11 12 ...

Operation `System.out.print(cnt++);` ist nicht atomar
Unterbrechung während der Ausführung möglich



10.1. Threads

Beispiel Monitor: Hochzählen eines gemeinsamen Zählers

```
public class SyncListing extends Thread {
    static int cnt = 0;
    public static void main(String[] args) {
        Thread t1 = new SyncListing();
        Thread t2 = new SyncListing();
        t1.start();
        t2.start();
    }
    public void run() {
        while (true) {
            synchronized (getClass()) { // getClass() gibt die Laufzeitklasse
                System.out.print(cnt++ + " "); // dieses Objekts zurück.
            }
        }
    }
}
```

10.1. Threads

Anwendung von synchronized auf eine Methode

- Zugriff auf ein Objekt selbst wird synchronisiert
- Mehr als ein Thread wird das Objekt zur gleichen Zeit verwenden
- Beispiel „Zählerobjekt“:
 - Kapselung des Zählers
 - Auf Anforderung aktuellen Zählerstand liefern und internen Zähler inkrementieren

10.1. Threads

Beispiel (1/3)

```
class Counter {  
    int cnt;  
    public Counter(int cnt) {    this.cnt = cnt; }  
    public int nextNumber() {  
        int ret = cnt;  
        // zeitaufwändige Berechnung um langwierige Operation zu simulieren:  
        double x = 1.0, y, z;  
        for (int i= 0; i < 9000000; ++i) {  
            x = Math.sin((x*i%35)*1.13);  
            y = Math.log(x+10.0);  
            z = Math.sqrt(x+y);  
        }  
        cnt++;  
        return ret;  
    }  
}
```

10.1. Threads

Beispiel (2/3)

```
public class Listing extends Thread {
    private String name; private Counter counter;
    public Listing(String name, Counter counter) {
        this.name = name; this.counter = counter;
    }
    public static void main(String[] args) {
        Thread[] t = new Thread[5];
        Counter cnt = new Counter(10);
        for (int i = 0; i < 5; ++i) {
            t[i] = new Listing("Thread-" + i, cnt); t[i].start();
        }
    }
    public void run() {
        while (true)
            System.out.println(counter.nextNumber() + " for " + name);
    }
}
```

10.1. Threads

Beispiel (3/3)

- Ergebnis: doppelte Zahlenwerte:
- Markierung der Methode `nextNumber()` als `synchronized` macht diese zu einem Monitor.
Code in der Methode wird als atomares Programmfragment behandelt. Unterbrechung des kritischen Abschnitts durch anderen Thread ist nicht möglich.

```
public synchronized int nextNumber() { // ...
```

```
10 for Thread-2  
11 for Thread-4  
10 for Thread-0  
10 for Thread-1  
11 for Thread-2  
11 for Thread-3  
12 for Thread-4  
13 for Thread-0  
14 for Thread-1  
15 for Thread-2  
16 for Thread-3  
...
```

10.1. Threads

wait und **notify**

- **wait** und **notify** sind Synchronisationsprimitive der Klasse **Object**
Das Objekt besitzt Warteliste von Threads, die unterbrochen wurden und auf ein Ereignis warten, um fortgesetzt zu werden. **wait** und **notify** dürfen nur innerhalb eines synchronized-Blocks aufgerufen werden
- Aufruf von **wait**
 - Nimmt bereits gewährten Sperren zurück
 - Stellt den Prozess, der den Aufruf von wait verursachte, in die Warteliste des Objekts
- Aufruf von **notify**
 - entfernt einen (beliebigen) Prozess aus der Warteliste des Objekts
 - Stellt die aufgehobenen Sperren wieder her

10.1. Threads

Beispiel Producer/Consumer für Fließkommazahlen (1/3)

```
class Producer extends Thread {  
    private Vector v;  
    public Producer(Vector v) { this.v = v; }  
    public void run() {  
        String s;  
        while (true) {  
            synchronized (v) {  
                s = "Wert "+Math.random();  
                v.addElement(s);  
                System.out.println("Produzent erzeugt "+s);  
                v.notify();  
            }  
            try {  
                Thread.sleep((int)(100*Math.random()));  
            } catch (InterruptedException e) { }  
        }  
    }  
}
```


10.1. Threads

Beispiel Producer/Consumer für Fließkommazahlen (2/3)

```
class Consumer extends Thread {  
    private Vector v;  
    public Consumer(Vector v) { this.v = v; }  
    public void run() {  
        while (true) {  
            synchronized (v) {  
                if (v.size() < 1)  
                    try { v.wait(); } catch (InterruptedException e) { }  
                System.out.print("Konsument fand " + (String)v.elementAt(0));  
                v.removeElementAt(0);  
                System.out.println(" (verbleiben: " + v.size() + ")");  
            }  
            try {  
                Thread.sleep((int)(100*Math.random()));  
            } catch (InterruptedException e) { }  
        }  
    }  
}
```

10.1. Threads

Beispiel Producer/Consumer für Fließkommazahlen (3/3)

```
public class ProdConListing {  
    public static void main(String[] args) {  
        Vector v = new Vector();  
        Producer p = new Producer(v);  
        Consumer c = new Consumer(v);  
        p.start();  
        c.start();  
    }  
}
```

Beispielausgabe

```
Produzent erzeugte Wert 0.6548096532111007  
Konsument fand Wert 0.6548096532111007 (verbleiben: 0)  
Produzent erzeugte Wert 0.6965546173953919  
Produzent erzeugte Wert 0.6990053250441516  
Produzent erzeugte Wert 0.9874467815778902  
Konsument fand Wert 0.6965546173953919 (verbleiben: 2)  
Produzent erzeugte Wert 0.019655027417308846  
Konsument fand Wert 0.6990053250441516 (verbleiben: 2)
```

10.2. Sockets Adressierung

- Zur Adressierung von Rechnern im Netz wird die Klasse **InetAddress** des Pakets **java.net** verwendet
- Ein **InetAddress**-Objekt enthält sowohl eine IP-Adresse als auch den symbolischen Namen des jeweiligen Rechners
- Die beiden Bestandteile können mit den Methoden **getHostName** und **getHostAddress** abgefragt werden.
- Mit Hilfe von **getAddress** kann die IP-Adresse auch direkt als **byte**-Array mit vier Elementen beschafft werden
 - **String** **getHostName()**
 - **String** **getHostAddress()**
 - **byte[]** **getAddress()**

10.2. Sockets Adressierung

- Statische Methoden zum Erzeugen eines `InetAddress`-Objekts:
 - `public static InetAddress getByName(String host) throws UnknownHostException`
 - `public static InetAddress getLocalHost() throws UnknownHostException`
- `getByName` erwartet einen String mit der IP-Adresse oder dem Namen des Hosts als Argument
- `getLocalHost` liefert ein `InetAddress`-Objekt für den eigenen Rechner
- `UnknownHostException` wenn die Adresse nicht ermittelt werden kann (z.B. kein DNS-Server)

10.2. Sockets

Beispiel

```
import java.net.*;
public class Lookup {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java Lookup <host>");
            System.exit(1);
        }
        try { // Get requested address
            InetAddress addr = InetAddress.getByName(args[0]);
            System.out.println(addr.getHostName());
            System.out.println(addr.getHostAddress());
        }
        catch (UnknownHostException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```

10.2. Sockets

Aufbau einer einfachen Socket-Verbindung

- Socket bezeichnet eine streambasierte Programmierschnittstelle zur Kommunikation zweier Rechner in einem TCP/IP-Netz
- Das Übertragen von Daten über eine Socket-Verbindung ähnelt dem Zugriff auf eine Datei:
 - Aufbau der Verbindung
 - Daten gelesen und/oder geschrieben.
 - Abbauen der Verbindung
- Erzeugen eines Sockets:
 - `public Socket(String host, int port) throws UnknownHostException, IOException`
 - `public Socket(InetAddress address, int port) throws IOException`

10.2. Sockets

Aufbau einer einfachen Socket-Verbindung

- Nachdem die Socket-Verbindung erfolgreich aufgebaut wurde, kann mit den beiden Methoden `getInputStream` und `getOutputStream` je ein Stream zum Empfangen und Versenden von Daten beschafft werden:
 - `public InputStream getInputStream() throws IOException`
 - `public OutputStream getOutputStream() throws IOException`
- Streams können direkt verwendet werden oder mit `Filterstreams` in bequemer zu verwendenden Streamtyp geschachtelt werden.
- Nach Ende der Kommunikation werden Eingabe- und Ausgabestreams als auch der Socket selbst mit `close` geschlossen

10.2. Sockets

Beispiel einer Socket-Verbindung

Verbindung zum DayTime-Service auf Port 13:

```
public class DayTime {  
    public static void main(String[] args) {  
        String hostname = "time.nist.gov";  
        try {  
            Socket sock = new Socket(hostname, 13);  
            InputStream in = sock.getInputStream();  
            int len;  
            byte[] b = new byte[100];  
            while ((len = in.read(b)) != -1) { // Lese Zeit als Serie von Bytes von  
                System.out.write(b, 0, len); // time.nist.gov, Port 13  
            }  
            in.close();  
            sock.close();  
        } catch (IOException e) { System.err.println(e.toString()); }  
    }  
}
```


10.2. Sockets

Beispiel einer Socket-Verbindung

Echo / Lesen und Schreiben von Daten (1/3)

```
class EchoClient {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage: java EchoClient <host>");
            System.exit(1);
        }
        try {
            Socket sock = new Socket(args[0], 7);
            InputStream in = sock.getInputStream();
            OutputStream out = sock.getOutputStream();
            sock.setSoTimeout(300); // Timeout setzen
            OutputThread th = new OutputThread(in); // Ausgabethread erzeugen
            th.start();
        }
    }
}
```

10.2. Sockets

Beispiel einer Socket-Verbindung

Echo / Lesen und Schreiben von Daten (2/3)

```
// Schleife für Benutzereingaben
BufferedReader conin = new BufferedReader(
    new InputStreamReader(System.in) );

String line = "";
while (true) {
    line = conin.readLine();    // Eingabezeile lesen
    if (line.equalsIgnoreCase("QUIT"))
        break;
    out.write(line.getBytes()); // Eingabezeile an ECHO-Server
    out.write('\r');
    out.write('\n');
    th.yield();    // Ausgabe abwarten
}
```

10.2. Sockets

Beispiel einer Socket-Verbindung

Echo / Lesen und Schreiben von Daten (3/3)

```
// Programm beenden
System.out.println("terminating output thread...");
th.requestStop();
th.yield();
try { Thread.sleep(1000); } catch (InterruptedException e) { }
in.close();
out.close();
sock.close();
} catch (IOException e) {
    System.err.println(e.toString());
    System.exit(1);
}
} // end main
}
```

10.2. Sockets

Beispiel einer Socket-Verbindung

OutputThread (1/2)

```
class OutputThread extends Thread {  
    InputStream in;  
    boolean stoprequested;  
  
    public OutputThread(InputStream in) {  
        super();  
        this.in = in;  
        stoprequested = false;  
    }  
  
    public synchronized void requestStop() {  
        stoprequested = true;  
    }  
}
```

10.2. Sockets

Beispiel einer Socket-Verbindung

OutputThread (2/2)

```
public void run() {  
    int len;  
    byte[] b = new byte[100];  
    try {  
        while (!stoprequested) {  
            try {  
                if ((len = in.read(b)) == -1) break;  
                System.out.write(b, 0, len);  
            } catch (InterruptedException e) {  
                // nochmal versuchen  
            }  
        }  
    } catch (IOException e) {  
        System.err.println("OutputThread: " + e.toString());  
    }  
}
```

10.2. Sockets

Server Sockets

- Klasse **ServerSocket** bietet Funktionen um auf einen eingehenden Verbindungswunsch zu warten und nach erfolgreichem Verbindungsaufbau einen Socket zur Kommunikation mit dem Client zurückzugeben (`import java.net.serversocket;`)
 - `public ServerSocket(int port) throws IOException`
 - `public Socket accept() throws IOException`
- Konstruktor erzeugt einen **ServerSocket** für einen bestimmten Port, also einen bestimmten Typ von Server-Anwendung
- Anschließend wird die Methode `accept` aufgerufen, um auf einen eingehenden Verbindungswunsch zu warten
- `accept` blockiert so lange, bis sich ein Client bei der Server-Anwendung anmeldet
- Ist der Verbindungsaufbau erfolgreich, liefert `accept` ein Socket-Objekt, das wie bei einer Client-Anwendung zur Kommunikation mit der Gegenseite verwendet werden kann

10.2. Sockets

Beispiel Server Socket: Echo Server

```
class SimpleEchoServer {
    public static void main(String[] args) {
        try {
            ServerSocket echod = new ServerSocket(7);
            Socket socket = echod.accept(); // Warte auf Verbindung (Port 7)
            InputStream in = socket.getInputStream();
            OutputStream out = socket.getOutputStream();
            int c;
            while ((c = in.read()) != -1) {
                out.write((char)c);
                System.out.print((char)c);
            }
            socket.close(); // Verbindung beenden
            echod.close();
        } catch (IOException e) { /*...*/ }
    }
}
```

10.2. Sockets

Verbindungen zu mehreren Clients

Für jeden Client soll ein eigener Thread angelegt werden

- Hauptprogramm erzeugt **ServerSocket**
- Warten auf Verbindungswunsch in einer Schleife mit **accept**
- Nach dem Verbindungsaufbau erfolgt die weitere Bearbeitung in einem neuen Thread mit dem Verbindungs-Socket als Argument
 - Thread erledigt die gesamte Kommunikation mit dem Client
 - Beendet der Client die Verbindung, wird auch der zugehörige Thread beendet

10.2. Sockets

Beispiel Server für mehrere Clients (1/3)

```
class EchoServer {
    public static void main(String[] args) {
        int cnt = 0;
        try {
            System.out.println("Warte auf Verbindungen auf Port 7...");
            ServerSocket echod = new ServerSocket(7);
            while (true) { // accept-Aufruf wird in eine Schleife verpackt und für
                Socket socket = echod.accept(); // jeden einkommenden Client-Request
                (new EchoClientThread(++cnt, socket)).start(); // wird ein eigener
                                                                // Thread EchoClientThread erzeugt
            }
        } catch (IOException e) {
            System.err.println(e.toString());
            System.exit(1);
        }
    }
}
```

10.2. Sockets

Beispiel Server für mehrere Clients (2/3)

```
class EchoClientThread extends Thread {  
    private int name;  
    private Socket socket;  
  
    public EchoClientThread(int name, Socket socket) {  
        this.name = name;  
        this.socket = socket;  
    }  
    //...
```

10.2. Sockets

Beispiel Server für mehrere Clients (3/3)

```
public void run() {  
    String msg = "EchoServer: Verbindung " + name;  
    System.out.println(msg + " hergestellt");  
    try {  
        InputStream in = socket.getInputStream();  
        OutputStream out = socket.getOutputStream();  
        out.write((msg + "\r\n").getBytes());  
        int c;  
        while ((c = in.read()) != -1) {  
            out.write((char)c);  
            System.out.print((char)c);  
        }  
        System.out.println("Verbindung " + name + " wird beendet");  
        socket.close();  
    }  
    catch (IOException e) { System.err.println(e.toString()); }  
}
```

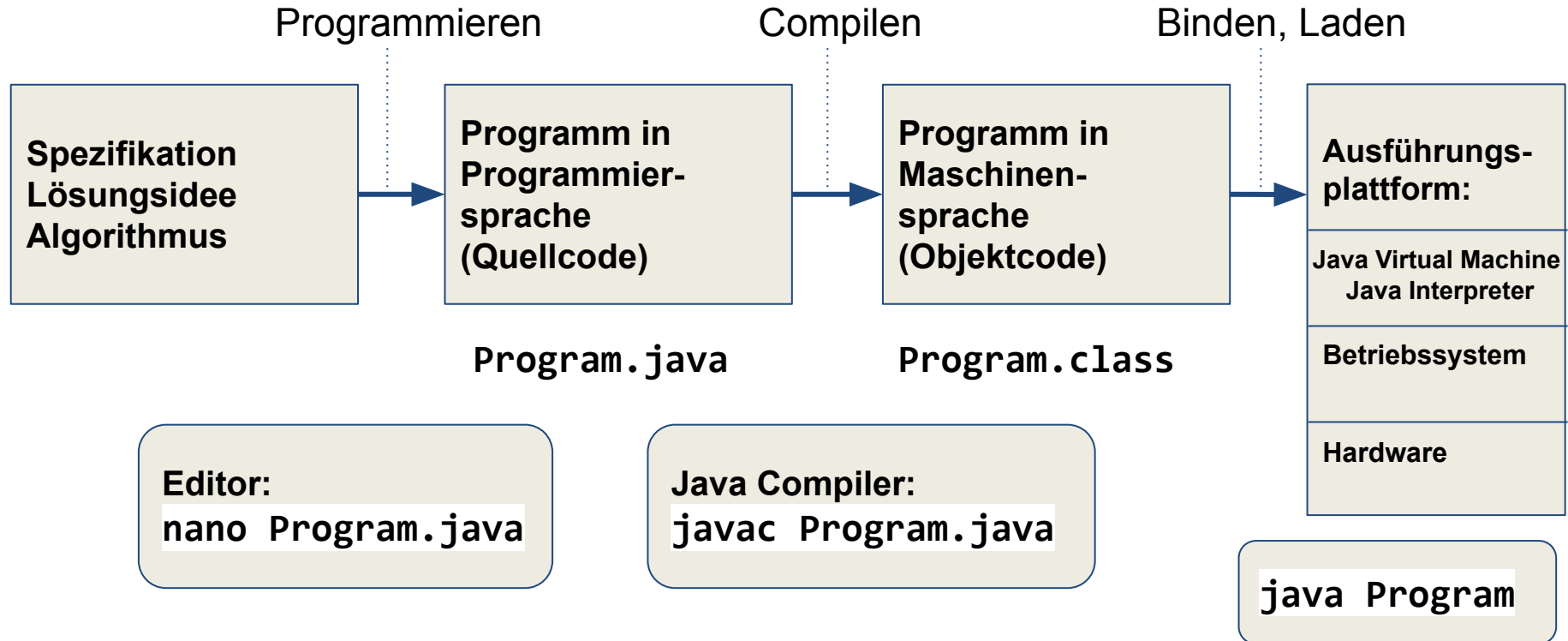
Objekt

erung



Überblick

1. Erstellen von Programmen in Java



2. Syntaktische Grundelemente

- Schlüsselworte:

abstract	const	float	int	protected	throw
boolean	continue	for	interface	public	throws
break	default	future	long	rest	transient
byte	do	generic	native	return	true
byvalue	double	goto	new	short	try
case	else	if	null	static	var
cast	extends	implements	operator	super	void
catch	false	import	outer	switch	volatile
char	final	inner	package	synchronized	while
class	finally	instanceof	private	this	

Ablaufkontrolle	Konstante	andere (Deklarationen)
Datentypen	Objekt-Orientierung	reserviert für Erweiterungen

- Konstanten: `2002` `-2L` `0xFABEL` `2.1` `0.1E-23` `.1e+19` `'a'` `'\n'`
`true` `false` `null` `"Hallo"` ...
- Operatoren: `+` `-` `*` `/` `&` `&&` `=` `==` `>=` `*=` `>` `>>` `>>>` ...

- Namen/Identifikatoren:

`Summe`, `getName`, `$all4you`,
`_1_2`, `beliebige_Länge_123`

- bestehen aus beliebige Buchstaben, Ziffern, `'_'` und `'$'`
- beginnen nie mit Ziffern

- Klammern: `()` `[]` `{ }`
- Trennzeichen: `,` `;` `.`
 plus Leerräume, Tabstops
 und Zeilenwechse

3. Datentypen und Variablen

- `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`: `char sym = '?'`;
- Arrays (Felder): `double[] nums = {3.1, 2.2, 1.3}`;
- Klassen als benutzerdefinierte Typen: `String name = new String("me")`;
- Typkonversion:
 - Explizit: `short num2 = (short) sym`;
 - Implizit: `int num3 = num2`;
- Gültigkeitsbereich:

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

```
{    int a = 3, b = 1;
    if (a > 0) {
        int b;    // Fehler: b bereits deklariert
        int c;    /* OK */
    }
    { char c;    /* OK */
      double a; // Fehler: a bereits deklariert
    }
}
```

- Speicher:

`int num3`

--	--	--	--

`double d`

--	--	--	--	--	--	--	--

`char sym`

--	--

4. Anweisungen (1/2)

- Ausdrücke und Zuweisungen, Priorität:
 - Ein Ausdruck berechnet einen Wert: `(x - y)*2.4;` `produkt != 3117;` `number%3;`
 - Eine Zuweisung weist einer Variablen einen Wert zu: `num = 1;` `produkt *= x - y;`
 - Prä-/Postfix In-/Dekrement: `x=1;` `y = ++x;` `//x=y=2` `x=1;` `y = x++;` `//x=2,y=1`
- Anweisungen werden immer mit ';' beendet, Anweisungsfolge wird mit { } zu Block
- Auswahl: `if (zahl==0) signum=0; else if (zahl>0) signum=+1; else signum=-1;`
`signum = (zahl==0) ? 0 : (zahl>0 ? +1 : -1);`

`switch (menuItem) { case 1: /*...*/ break;`
`case 2: /*...*/ break;`
`case 3: case 4: case 5: /*...*/ break;`
`default: /*...*/ break;`
`}`

4. Anweisungen (2/2)

- Wiederholungs-Anweisungen:

- `int i, summe; summe = 0; i = 1; while (i < 10) { summe += i; i++; }`
- `int i, summe; summe = 0; i = 1; do { summe += i; i++; } while (i <= 10);`
- `int i, summe; summe = 0; for (i = 1; i <= 10; i++) { summe += i; }`

- Programmier-Konventionen:

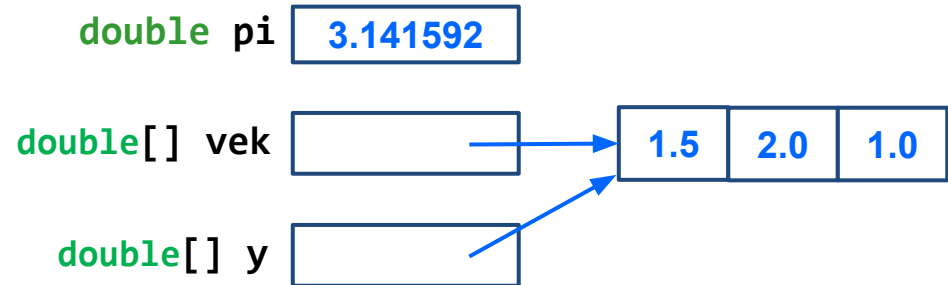
- Aussagekräftige Namen
- Einrückungen (Indentation)
- Anweisungen mit { }
klammern
- Kommentaren

```
int zahl, teiler;
boolean prim;
System.out.println(2);
for ( zahl = 3; zahl < 100; zahl += 2 ) {
    prim = true;
    for ( teiler = 3; teiler < zahl; teiler += 2 ) {
        if ( zahl % teiler == 0 ) {
            prim = false;
            teiler = zahl; // verlasse die for-teiler-Schleife
        }
    } // for-teiler-Schleife
    if ( prim )
        System.out.println(zahl);
} // for-zahl-Schleife
```

5. Arrays und Strings

- Arrays:

```
double pi = 3.141592;  
double[] vek = null;  
vek = new double[3];  
// vek.length == 3  
vek[0] = 1.5;  
vek[1] = 2.0;  
vek[2] = 1.0;  
double[] y = vek;
```



- Strings:

```
String motto = null; motto = "Wir lernen Java!"; // automatisches new  
// motto.length()==16, motto.equals("Wir lernen Python!")==false
```



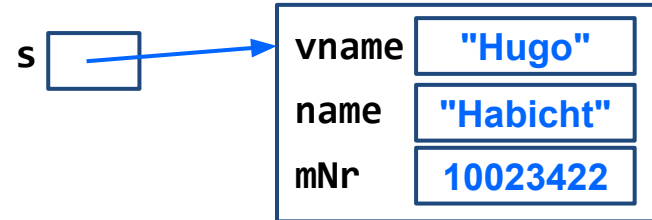
6. Objekte und Methoden (1/5)

- Klassen, Objekte, (Klasse-)Attribute und Methoden, Konstruktoren, Kapselung:

```
class Student {  
    private final static long DEFAULT = 1000000;  
    private static int anzahl = 0;  
    private String vname, name;  
    private long mNr;  
  
    public Student() { mNr = DEFAULT; }  
    public Student(long mNr) { this.mNr = mNr; }  
    public long getMatrNr() { return mNr; }  
    public void setMatrNr(long m) { this.mNr = m; }  
    public static int getAnzahl() { return anzahl; }  
}  
  
// ...  
// Student.anzahl = 0  
Student s = new Student("Hugo", "Habicht", 160232);  
s.setMatrNr(160); // s.mNr = 160; -> Fehler  
// Student.anzahl = 1
```

Student

BASE	1000000
anzahl	0

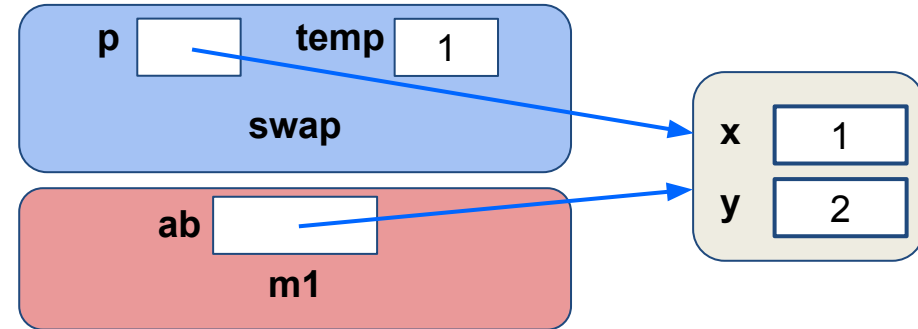
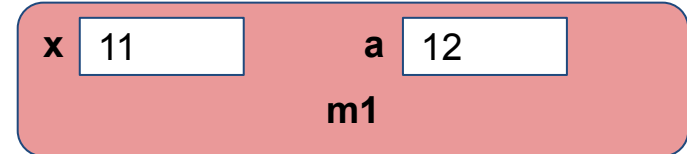
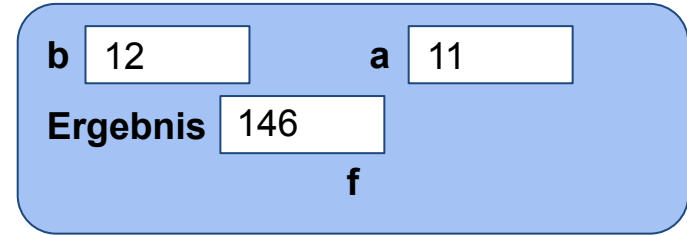


6. Objekte und Methoden (2/5)

• Aufruf von Methoden:

```
int f(int b, int a) {  
    a = 2 * b + a * a;  
    return a + 1;  
}  
void m1() {  
    int x = 10; int a = 12;  
    a = f(a, ++x) - f(a, a + 3);  
}
```

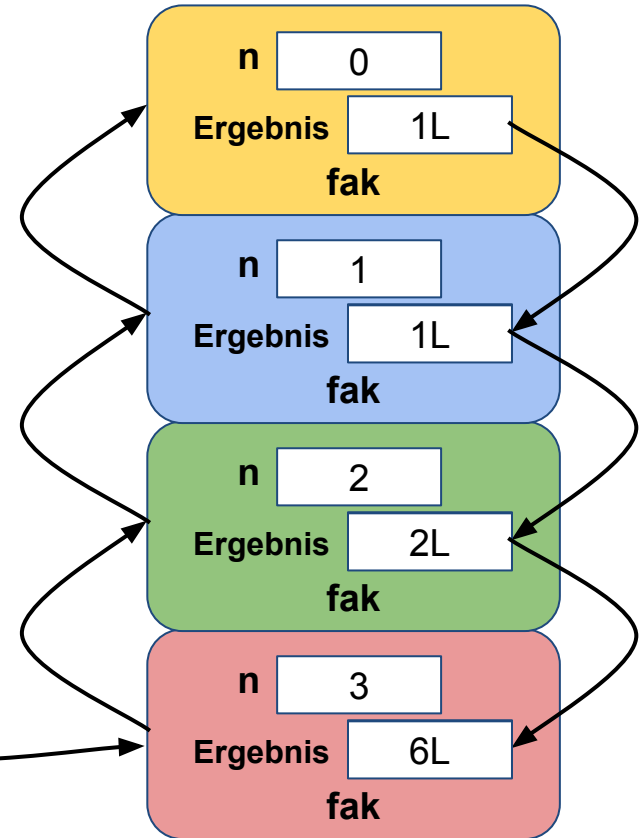
```
void swap(Pair p) {  
    int temp = p.x; p.x = p.y;  
    p.y = temp;  
}  
void m1() {  
    Pair ab = new Pair();  
    ab.x = 1; ab.y = 2; swap(ab);  
}
```



6. Objekte und Methoden (3/5)

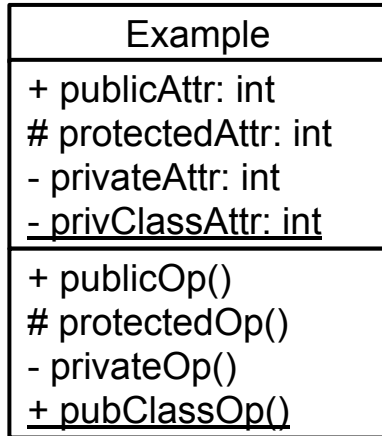
- Aufruf von Methoden, Rekursion:

```
static long fak(long n) {  
    if (n == 0)  
        return 1L;  
    else if (n > 0)  
        return n * fak(n-1);  
}  
long f = fak(3);
```



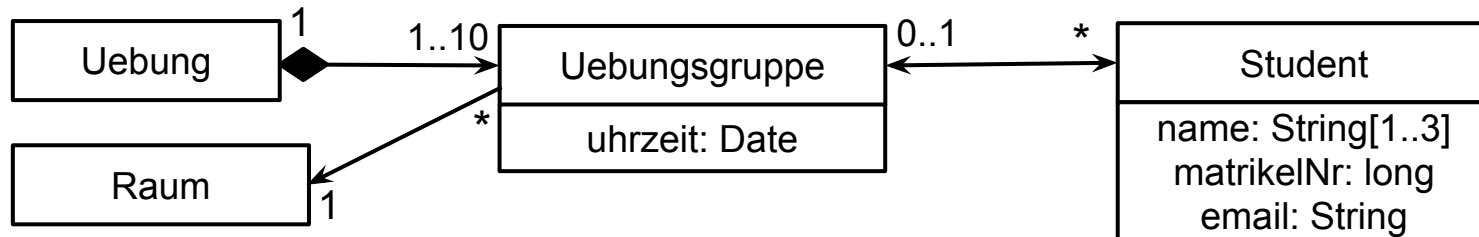
6. Objekte und Methoden (4/5)

- UML Klassendiagramm:



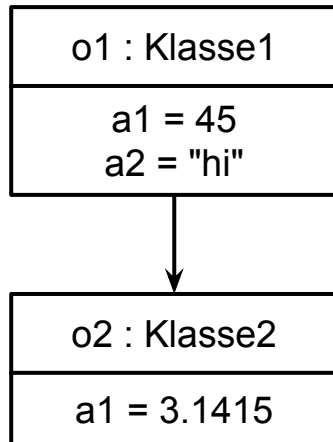
```
class Example {
    private int publicAttr;
    protected int protectedAttr;
    private int privateAttr;
    private static int privClassAttr;

    public void publicOp() { /*...*/ }
    protected void protectedOp() { /*...*/ }
    private void privateOp() { /*...*/ }
    private static void pubClassOp() { /*...*/ }
}
```



6. Objekte und Methoden (5/5)

- UML Objektdiagramm:



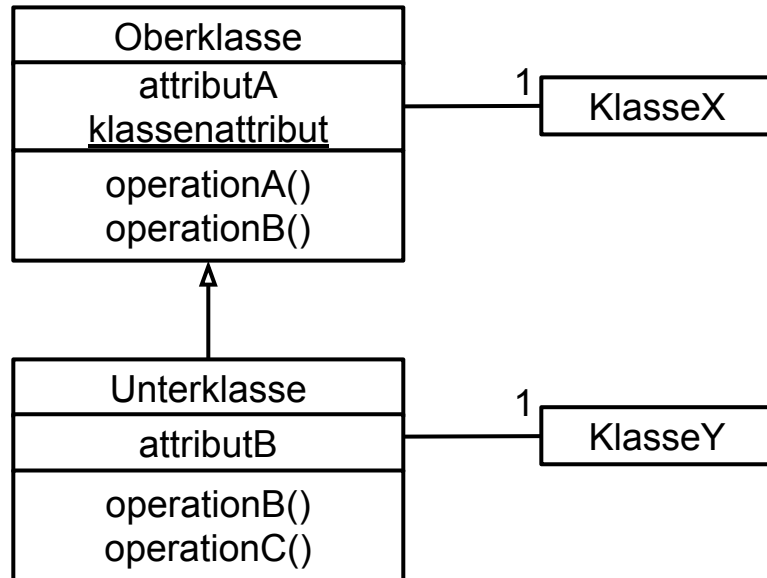
```
class Klasse1 {
    private int a1 = 45;
    private String a2 = "hi";
    public Klasse2 k2;
}

class Klasse2 {
    private double a1;
    public Klasse2(double a1) {
        this.a1 = a1;
    }
}

// ...
Klasse1 o1 = new Klasse1();
Klasse2 o2 = new Klasse2(3.1415);
o1.k2 = o2;
```

7. Vererbung und Polymorphie (1/5)

- in UML Klassendiagramm:

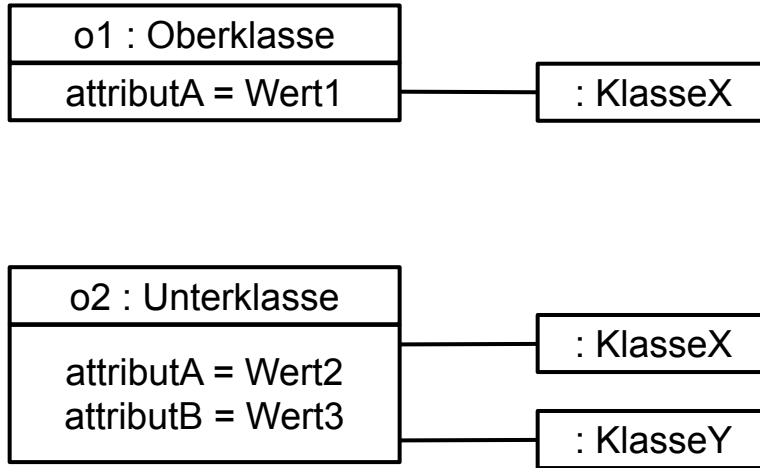


```
class Oberklasse {
    int attributA;
    static double klassenattribut;
    KlasseX kx;
    void operationA() { // ... }
    int operationB() { return 4; }
}

class Unterklasse extends Oberklasse {
    char attributB;
    KlasseY ky;
    int operationB() { return 5; }
    void operationC() { // ... }
}
```


7. Vererbung und Polymorphie (2/5)

- UML Objektdiagramm:



```
KlasseX x = new KlasseX();
KlasseY y = new KlasseY();

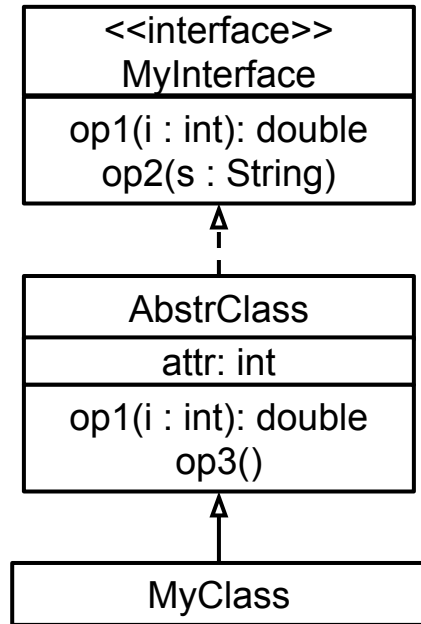
Oberklasse.klassenattribut = 1.234;

Oberklasse o1 = new Oberklasse();
o1.attributA = 1; o1.kx = x;
o1.operationA();
o1.operationB(); // returns 4

Unterklasse o2 = new Unterklasse();
o2.attributA = 2; o2.attributB = 'x';
o2.kx = x; o2.ky = y;
o2.operationA(); o2.operationC();
o2.operationB(); // returns 5
```

7. Vererbung und Polymorphie (3/5)

- Schnittstellen (Interfaces), abstrakte Klassen und Methoden:



```
interface MyInterface {
    public double op1(int i);
    public void op2(String s);
}

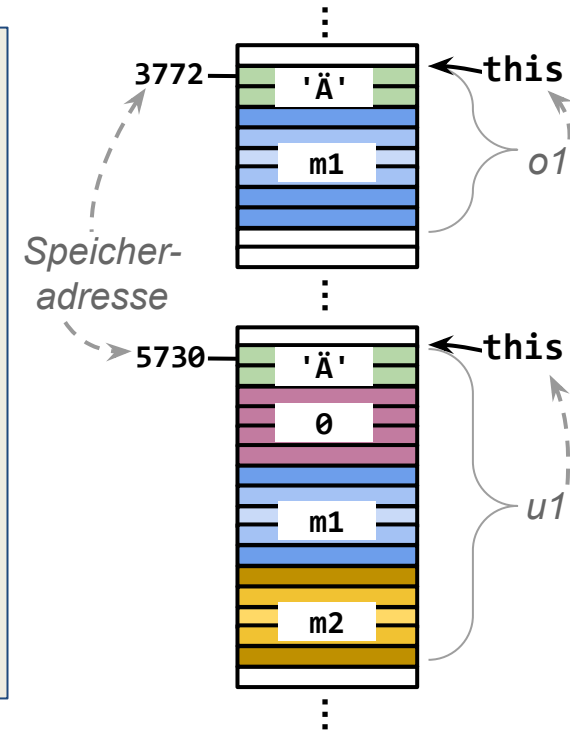
abstract class AbstrClass implements MyInterface {
    protected int attr;
    public double op1(int i) { /* ... */ }
    public void op3() { /* ... */ }
}

class MyClass extends AbstrClass {
    public void op2(String s) { /* ... */ }
}
```

7. Vererbung und Polymorphie (4/5)

- Konstruktor, **this** und **super**:

```
class Ober {  
    protected char a1 = 'Ä';  
    public boolean m1(char a1) { return this.a1 == a1; }  
    public Ober() { a1 = '?'; }  
}  
class Unter extends Ober {  
    private int a2 = 0;  
    public boolean m1(char a1) { return a2 == (int) a1; }  
    public void m2() { this.a2 = (int) super.a1; }  
    public Unter() { super(); a2 = (int) '?'; }  
}  
// ...  
Ober o1 = new Ober();  
Unter u1 = new Unter();  
u1.m1('A');
```



7. Vererbung und Polymorphie (5/5)

- Polymorphie, **Object**:

```
class Hund extends Tier {  
    // ...  
    public String toString() {  
        return "Ich bin " + name + ", der " + rasse + ".";  
    }  
}
```

```
Hund waldi = new Hund ("Waldi", "Dackel");  
System.out.println(waldi);
```

```
public void println(Object obj) {  
    String str = obj.toString(); // Polymorphie!  
    println(str);                // Aufruf der überladenen Methode  
}
```

8. Ausnahmebehandlung

- **throw**, Definition eigener Exceptions: `class MyExcept extends Exception { //...`

```
public static double invert(double x) {  
    if (x == 0.0) throw new ArithmeticException("Divide by 0.");  
    return 1.0 / x;  
}
```

- **try, catch, final**

```
PrintWriter pw = null;  
try {  
    Student s = new Student("Hugo", "Test", 12345678);  
    s.writeToStream(new PrintWriter(System.out));  
    pw = new PrintWriter( new FileWriter("out.txt") );  
}  
catch (IOException e) { /*...*/ }  
finally {  
    if (pw != null) pw.close();    // Keine IOException  
}  
BufferedReader reader = null;
```

```
import java.util.Scanner; // Importiere Scanner Klasse

class Klausur01 {
    public static void main(String[] str) {
        // lese Benutzereingaben von Konsole in max:
        Scanner input = new Scanner(System.in);
        System.out.println("Maximum?");
        int max = input.nextInt();
        // 1. Schreiben Sie Code, um (möglicherweise wiederholt) zu prüfen, ob der Wert von max zwischen 6
        // und 999 liegt (6 und 999 inklusive). Falls das nicht so ist, sollte den Benutzer erneut um einen
        // Wert gefragt werden:

        // 2. Schreiben Sie alle Zahlen von 5 bis zum Wert von max auf die Konsole, eine pro Zeile, und
        // ersetzen Sie die Zahl durch "hop", wenn sie ein Vielfaches von 7 ist, und "pla", wenn sie ein
        // Vielfaches von 9 ist:

    }
}
```

```
import java.util.Scanner; // Importiere Scanner Klasse

class Klausur01 {
    public static void main(String[] str) {
        // lese Benutzereingaben von Konsole in max:
        Scanner input = new Scanner(System.in);
        System.out.println("Maximum?");
        int max = input.nextInt();
        // 1. Schreiben Sie Code, um (möglicherweise wiederholt) zu prüfen, ob der Wert von max zwischen 6
        // und 999 liegt (6 und 999 inklusive). Falls das nicht so ist, sollte den Benutzer erneut um einen
        // Wert gefragt werden:
        while ( ! ( ( 6 <= max ) && ( max <= 999 ) ) ) {
            System.out.println("Maximum?");
            max = input.nextInt();
        }
        // 2. Schreiben Sie alle Zahlen von 5 bis zum Wert von max auf die Konsole, eine pro Zeile, und
        // ersetzen Sie die Zahl durch "hop", wenn sie ein Vielfaches von 7 ist, und "pla", wenn sie ein
        // Vielfaches von 9 ist:
        for (int num = 5; num <= max; num++) {
            if ( num%7==0 )      System.out.println("hop");
            else if ( num%9==0 ) System.out.println("pla");
            else                 System.out.println(num);
        }
    }
}
```

```

/** Implementiere die Methode dreieck(), die auf der Konsole ein Dreieck aus dem Zeichen X ausgibt,
    abhängig von der Variablen int size (size = 3, 4, ...):
size = 3:      size = 4:      size = 5:      etc.
  XXX          XXXX          XXXXX
   XX          XXX           XXXX
    X          XX            XXX
                  X          XX
                      X
*/
import java.util.Scanner; // Importiere Scanner Klasse

class Klausur02 {
    public static void dreieck(int size) {
        // 1. Überprüfe, ob size 3 oder größer ist und wenn nicht, verlasse die Methode:

        // 2. Gebe das Dreieck in Größe size aus:
    }
    public static void main(String[] str) {
        int size = 7;
        System.out.print("Dreieck von Größe ");
        System.out.println( size + ":" );
        // 3. Rufen Sie hier die Methode dreieck auf mit size als Parameter:
    }
}

```



```
/** Finden Sie die 3 Compiler-Fehler:
 */

class Klausur03 {
    public static void main(String[] str) {
        boolean wot = ! false;
        if (wot) {
            char symbol1 = '@', symbol2 = '!';
            String returnStr;
            if (symbol1 > 0) {
                String str = " " + symbol1 + symbol2;
                str += symbol1;
            }
            returnStr = str;
            short returnShort = (short)symbol2;
            wot = (boolean) symbol2;
        }
    }
}
```

```
/** Finden Sie die 3 Compiler-Fehler:
 */
```

```
class Klausur03 {
    public static void main(String[] str) {
        boolean wot = ! false;
        if (wot) {
            char symbol1 = '@', symbol2 = '!';
            String returnStr;
            if (symbol1 > 0) {
                String str = " " + symbol1 + symbol2; → Variable str ist bereits Parameter der Methode main
                str += symbol1;
            }
            returnStr = str; → Variable str war nur im oberen Block gültig
            short returnShort = (short)symbol2;
            wot = (boolean) symbol2; → inkompatible Typen in Umwandlung von/zu boolean
        }
    }
}
```

```
/** 1. In welchen 3 Zeilen befinden sich Compiler-Fehler?  
    2. Was gibt dieses Programm aus, wenn die Zeilen gelöscht werden?  
*/
```

```
class Klausur04 {  
    public static void main(String[] s) {  
        double a, b = 1.2;  
        char c = 'a';  
        System.out.println("abracadabr"+c);  
        int d = 3, e = 8;  
        final boolean f = false;  
        a = (d > 1) ? 4.5 : 6.7;  
        b = c + a * d;  
        d = d + a;  
        f = (e <= 8);  
        d = (int)a + (int)c;  
        e = (int)(a == b);  
        System.out.println(d);  
    }  
}
```

```
/** 1. In welchen Zeilen befinden Sie die 3 Compiler-Fehler?  
    2. Was gibt dieses Programm aus, wenn die Zeilen gelöscht werden?  
*/
```

```
class Klausur04 {  
    public static void main(String[] s) {  
        double a, b = 1.2;  
        char c = 'a';  
        System.out.println("abracadabr"+c);  
        int d = 3, e = 8;  
        final boolean f = false;  
        a = (d > 1) ? 4.5 : 6.7;  
        b = c + a * d;  
        d = d + a;      → inkompatible Typen in Umwandlung von double (d + a) zu int (d)  
        f = (e <= 8);   → Variable f ist final  
        d = (int)a + (int)c;  
        e = (int)(a == b); → inkompatible Typen in Umwandlung von/zu boolean (a==b)  
        System.out.println(d);  
    }  
}
```

```
/** Was ist die Ausgabe dieses Programms?
 */

class Klausur05 {

    public static String prnt(int size, boolean b, MyInt of) {
        if (!b) return "1 " + (--size) + " " + of.add3();
        else    return "2 " + (++size) + " " + of.add3();
    }

    public static void main(String[] s) {
        MyInt i = new MyInt();
        int j = 5;
        do {
            System.out.println( Klausur05.prnt(j, i.me < 9, i) );
        } while (i.me < 12);
    }

}

class MyInt {
    public int me = 7;
    public int add3() { me = me + 3; return me;}
}
```

```
/** Was ist die Ausgabe dieses Programms?
 */

class Klausur06 {

    public static double calc(int[] array, int number) {
        for (int z = 2; z > 0; z-=2)
            array[z]--;
        return ( (number % 2) == 0 )? 2.7 + array[0] : 2.1 - array[2];
    }

    public static void main(String[] s) {
        int[] myArray = { 1, 33, 7};
        myArray[0] = myArray.length + 1;
        while (myArray[2] > 5)
            System.out.println( calc(myArray, myArray[2]) );
    }
}
```

// Was ist die Ausgabe dieses Programms?

```
class Klausur07 {
    private final static short MAX = 4;
    private static int anzahl = 0;

    protected double note;

    public Klausur07() {
        this.note = (anzahl++) / 2;
    }

    public double getNote() {
        return note;
    }

    public static void main(String[] s) {
        Klausur07[] klausur = new Klausur07[MAX];
        for (int k = 0; k < MAX; k++) {
            klausur[k] = new Klausur07();
            System.out.println( klausur[k].getNote() );
        }
    }
}
```

```
class Klausur08 {  
    public static void main(String[] s) {  
        Zahl z1 = new Zahl(1.23), z2 = new Zahl(4.56);  
        // Was muss hier geschrieben werden, um die Summe von z1 und z2 in der Konsole zu erhalten?  
        // Benutzen Sie nur die summe() Methode. Achtung: println() druckt nicht automatisch bei einem  
        // Objekt von Klasse Zahl den wert der Variable wert aus:  
  
    }  
}  
  
class Zahl {  
    private double wert = 0.0;  
    // Initialisieren Sie die Attribute der Klasse hier:  
    public Zahl(double wert) {  
  
    }  
    // Diese Methode gibt neue Zahl (Summe aus dieser Zahl und im Parameter angegebenen Zahl) zurück:  
    public Zahl summe(Zahl neueZahl) {  
  
    }  
    // Vervollständigen Sie diese get-Methode:  
    public double getWert() {  
  
    }  
}
```



```
class Klausur08 {  
    public static void main(String[] s) {  
        Zahl z1 = new Zahl(1.23), z2 = new Zahl(4.56);  
        // Was muss hier geschrieben werden, um die Summe von z1 und z2 in der Konsole zu erhalten?  
        // Benutzen Sie nur die summe() Methode. Achtung: println() druckt nicht automatisch bei einem  
        // Objekt von Klasse Zahl den wert der Variable wert aus:  
        System.out.println( z1.summe( z2 ).getWert() );  
    }  
}  
  
class Zahl {  
    private double wert = 0.0;  
    // Initialisieren Sie die Attribute der Klasse hier:  
    public Zahl(double wert) {  
        this.wert = wert;  
    }  
    // Diese Methode gibt neue Zahl (Summe aus dieser Zahl und im Parameter angegebenen Zahl) zurück:  
    public Zahl summe(Zahl neueZahl) {  
        return new Zahl( wert + neueZahl.getWert() );  
    }  
    // Vervollständigen Sie diese get-Methode:  
    public double getWert() {  
        return wert;  
    }  
}
```

354

```
class Klausur09 implements CharMethoden {

    // Definieren und implementieren Sie hier die fehlende Methode:
    public String entferneLeerzeichen(char[] chars) {
        String ret = "";
        for (int i=0; i < chars.length; i++) {
            if (! (chars[i] == ' ')) {
                ret = ret + chars[i];
            }
        }
        return ret;
    }

    public static void main(String[] s) {
        Klausur09 kl = new Klausur09();
        char[] myString = {'h','a',' ','h','i',' ','h','o'};
        // Verwenden Sie die Methode dieser Klasse auf das char-Array,
        // um die Ausgabe "hahiho" in der Konsole zu erhalten:
        System.out.println( kl.entferneLeerzeichen( myString ) );
    }
}

interface CharMethoden {
    // Erzeuge String, die alle Zeichen ohne Leerzeichen aus dem angegebenen Array enthält:
    public String entferneLeerzeichen(char[] chars);
}
```

```
class Klausur10 {
    public static void main(String[] s) {
        // Schreiben Sie unten Code, damit die Konsolenausgabe dieses
        // Programms wie folgt aussieht: "Cheerio,Hi,Gutentag,".

        for (int i = gruesse.length-1; i>=0; i--) {
            System.out.print( gruesse[i].gruss()+"," );
        }
        System.out.println();
    }
}

class Gruss {
    public String gruss() { return new String("Hi"); }
}

class EnglischGruss extends Gruss {
    public String gruss() { return new String("Cheerio"); }
}

class DeutschGruss extends Gruss {
    public String gruss() { return new String("Gutentag"); }
}
```

```
class Klausur10 {  
    public static void main(String[] s) {  
        // Schreiben Sie unten Code, damit die Konsolenausgabe dieses  
        // Programms wie folgt aussieht: "Cheerio,Hi,Gutentag,".  
        Gruss[] gruesse = new Gruss[3];  
        gruesse[0] = new DeutschGruss();  
        gruesse[1] = new Gruss();  
        gruesse[2] = new EnglischGruss();  
        for (int i = gruesse.length-1; i>=0; i--) {  
            System.out.print( gruesse[i].gruss()+"," );  
        }  
        System.out.println();  
    }  
}  
  
class Gruss {  
    public String gruss() { return new String("Hi"); }  
}  
  
class EnglischGruss extends Gruss {  
    public String gruss() { return new String("Cheerio"); }  
}  
  
class DeutschGruss extends Gruss {  
    public String gruss() { return new String("Gutentag"); }  
}
```