

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

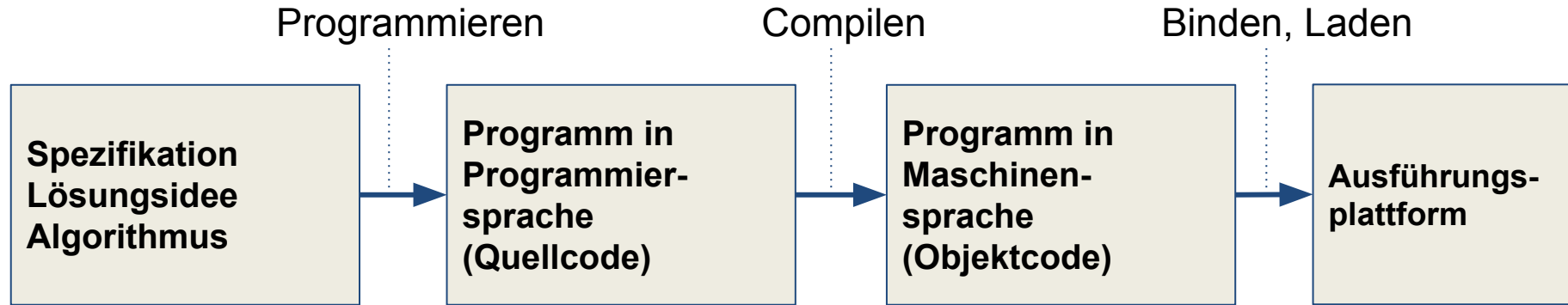
- 8. Ausnahmebehandlung (Exceptions)

Woche 6

- 9. Dateien, Ströme und Serialisierung
- 10. Threads und Sockets
- Zusammenfassung

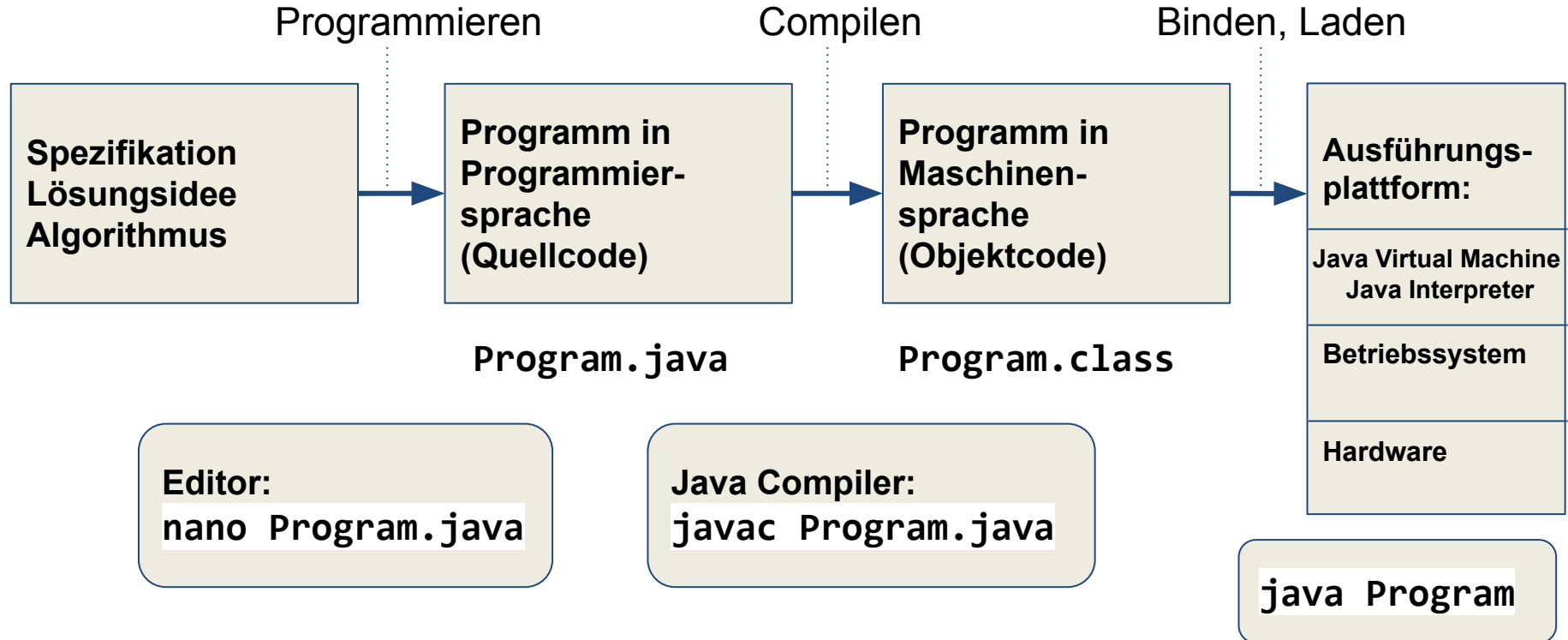
- <https://docs.oracle.com/javase/tutorial/>
- <https://beginnersbook.com/java-tutorial-for-beginners-with-examples/>
- <https://codingbat.com/java>
- <https://github.com/kristofvl/OFP> (neu!)

Entstehungsschritte eines Programms:



1. Erstellen von Programmen

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
{
    ...
}
```

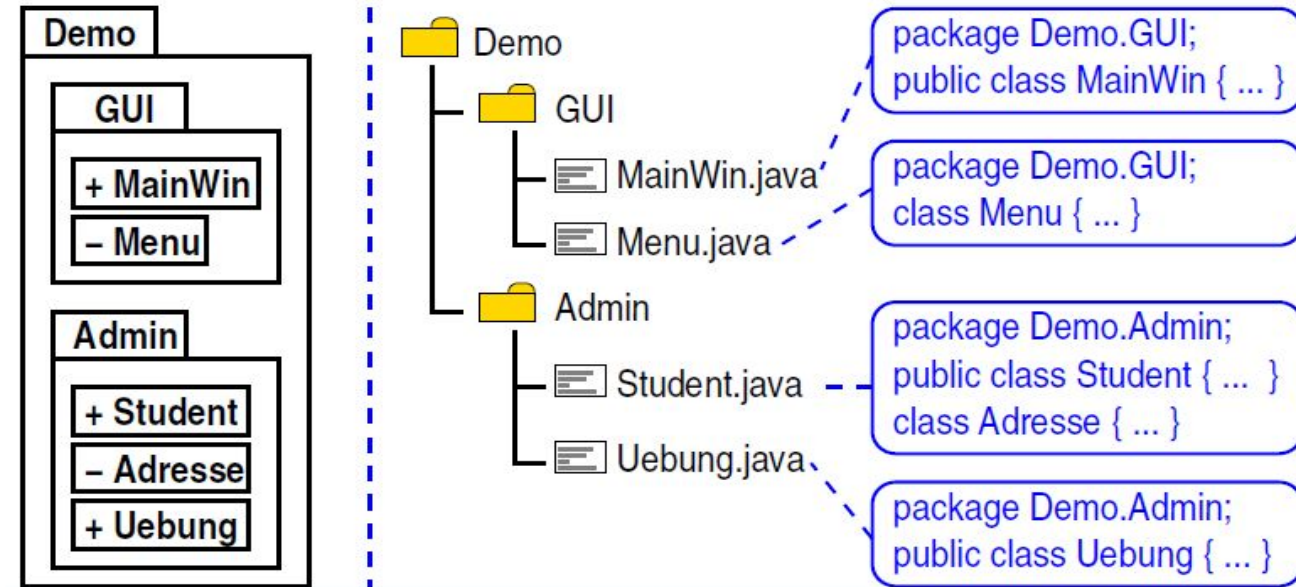
1. Erstellen von Programmen



```
/** 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

```
/**
 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;
        System.out.print( "Binärzahl zu dezimal: " );
        System.out.println( y ); // Methode zum Drucken auf die Konsole
    }
}
```

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 (🌶️)

```
/**
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 (🌶️🌶️🌶️🌶️)

```
/**
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

```
{  
    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'
```

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 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

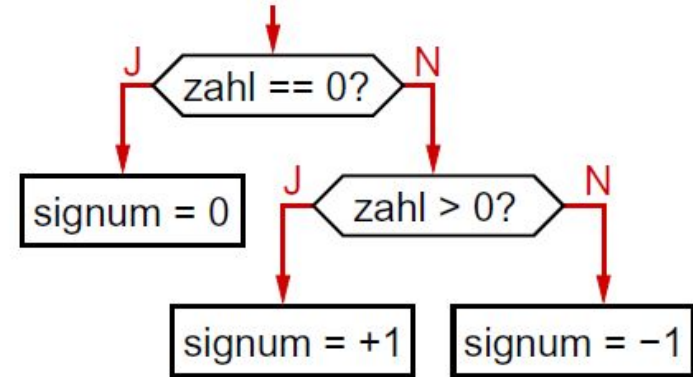
Beispiel: Berechne das Signum einer Zahl

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):

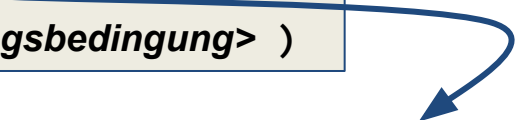
continue	verlasse den Block und beginne mit dem nächsten Durchlauf
break	verlasse die Schleife

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 (🌶️)

```
/**  
    Implementiere ein Programm, dass von 1 bis 26 zählt, aber statt der Zahl  
    "hoppla" ausgibt, wenn diese Zahl ein Vielfaches von 7 ist.  
*/  
  
class Schleife01 {  
    public static void main( String[] str ) {  
  
    }  
}
```

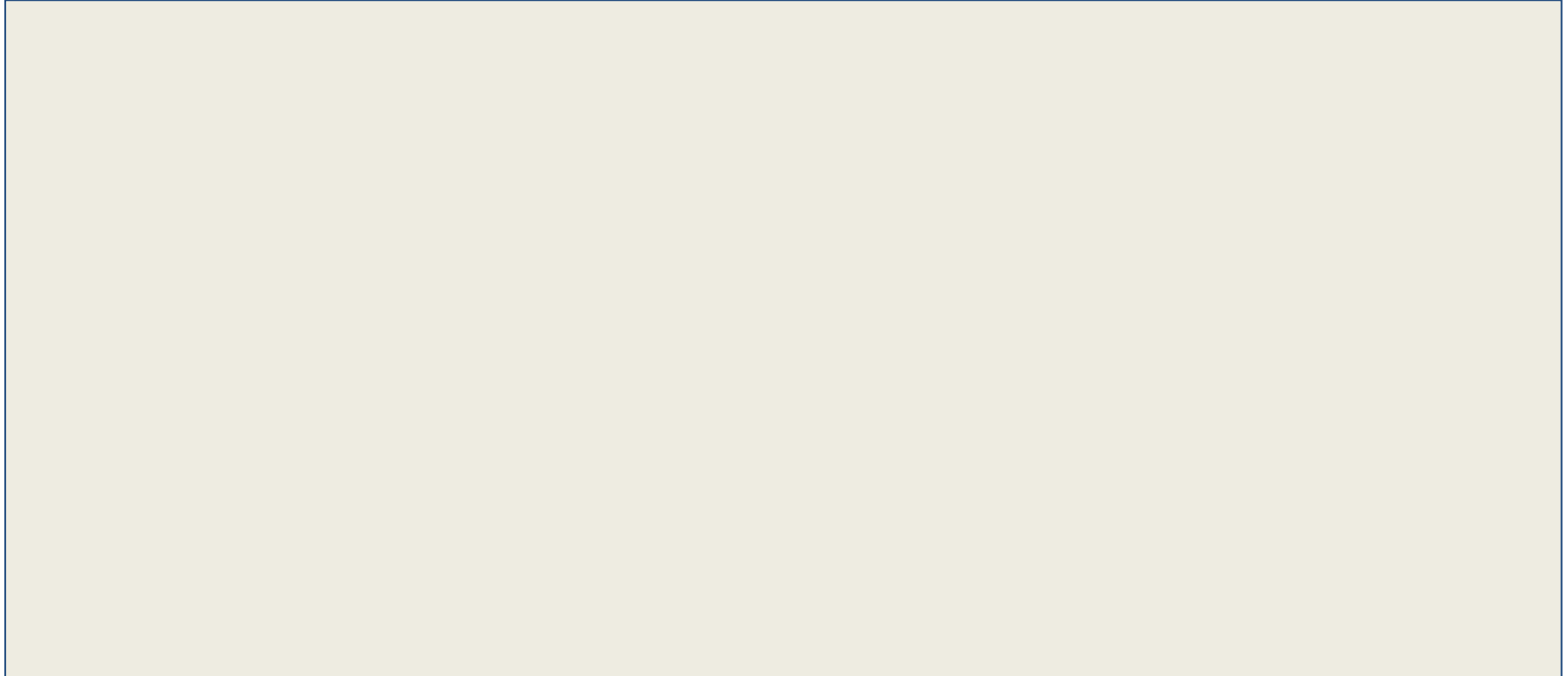
Beispiel (🌶️)

```
/**  
  Implementiere ein Programm, dass von 1 bis 26 zählt, aber statt der Zahl  
  "hoppla" ausgibt, wenn diese Zahl ein Vielfaches von 7 ist.  
*/
```

Beispiel (🌶️🌶️🌶️)

```
/**  
    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       X   X  
  
*/  
  
class Schleife02 {  
    public static void main( String[] str ) {  
  
    }  
}
```

Beispiel (🌶️🌶️🌶️)



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;

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
        }
    }
    if ( prim )
        System.out.println(zahl);
}
```