2. Einfache Programme

Grundsymbole

Namen

bezeichnen Variablen, Typen, ... in einem Programm

- bestehen aus Buchstaben, Ziffern und "_"
- beginnen mit Buchstaben
- beliebig lang
- Groß-/Kleinschreibung signifikant

```
x
x17
myVar
myvar
my_var
```

Schlüsselwörter

- heben Programmteile hervor
- dürfen nicht als Namen verwendet werden

```
if
while
```

Zahlen

- ganze Zahlen (dezimal oder hexadezimal)
- Gleitkommazahlen

```
376     dezimal
0x1A5   1*16^2+10*16^1+5*16^0
3.14    Gleitkommazahl
```

Zeichenketten (Strings)

- beliebige Zeichenfolgen zwischen Hochkommas
- dürfen nicht über Zeilengrenzen gehen
- " in der Zeichenkette wird als \" geschrieben

```
"a simple string"
"sie sagte \"Hallo\""
```

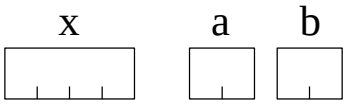
2. Einfache Programme

Variablendeklarationen

Jede Variable muss vor ihrer Verwendung deklariert werden

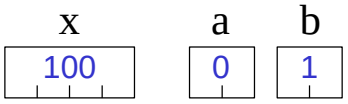
- macht den Namen und den Typ der Variablen bekannt
- Compiler reserviert Speicherplatz für die Variable

int x;	deklariert eine Variable *x* vom Typ *int* (integer)
short a, b;	deklariert 2 Variablen *a* und *b* vom Typ *short* (short integer)



Variablen können bei ihrer Deklaration initialisiert werden

int x = 100;	deklariert *int*-Variable *x*; weist ihr den Anfangswert 100 zu
short a = 0, b = 1;	deklariert 2 *short*-Variablen *a* und *b* mit Anfangswerten



Grammatik

```
VarDecl  = Type Var {"," Var} ";".
Var      = ident [ "=" ConstExpr ].
Type     = "byte" | "short" | "int" | "long" | ... .
```

Ganzzahlige Typen

byte	8-Bit-Zahl	-2^7 .. 2^7-1	(-128 .. 127)
short	16-Bit-Zahl	-2^{15} .. $2^{15}-1$	(-32768 .. 32767)
int	32-Bit-Zahl	-2^{31} .. $2^{31}-1$	(-2 147 483 648 .. 2 147 483 647)
long	64-Bit-Zahl	-2^{63} .. $2^{63}-1$	

Typhierachie

long Ê int Ê short Ê byte

der größere Typ schließt den kleineren ein

Minima und Maxima

Byte.MIN_VALUE	-2^7	(-128)	Byte.MAX_VALUE	2^7-1	(127)
Short.MIN_VALUE	-2^{15}	(-32768)	Short.MAX_VALUE	$2^{15}-1$	(32767)
Integer.MIN_VALUE	-2^{31}	(-2_147_483_648)	Integer.MAX_VALUE	$2^{31}-1$	(2_147_483_647)
Long.MIN_VALUE	-2^{63}		Long.MAX_VALUE	$2^{63}-1$	

Ganzzahlige Konstanten

Wert	*Typ*
10	int
10L	long

(passt auch in *byte* und *short*)

Beispiele

```
byte  a = 1;
short b = 100;
int   c = 100000;
long  d = 1L;
long  e = 1;
```

Fehler

```
byte  a = 200;
short b = 100000;
int   c = 3_000_000_000;
```

Gleitkommatypen

float	32-Bit-Zahl	~ ± 3.4 * 10^{38}	Float.MIN_VALUE .. Float.MAX_VALUE
double	64-Bit-Zahl	~ ± 1.8 * 10^{308}	Double.MIN_VALUE .. Double.MAX_VALUE

auch höhere Genauigkeit (mehr Nachkommastellen)

Typhierachie

double Ê float Ê long Ê int Ê short Ê byte

Gleitkommakonstanten

```
3.14        // Typ double
3.14f       // Typ float
3.14E0      // 3.14 * 10^0
0.314E1     // 0.314 * 10^1
31.4E-1     // 31.4 * 10^-1
.23         // 0.23
1.E2        // 1 * 10^2 = 100
```

Beispiele

```
float   a = 3.14F;
double  b = 3.14;
double  c = 3;
```

Grammatik

```
FloatConstant  = [Digits] "." [Digits] [Exponent] [FloatSuffix].
Digits         = digit {digit}.
Exponent       = ("e" | "E") ["+" | "-"] Digits.
FloatSuffix    = "f" | "F" | "d" | "D".
```

Konstantendeklarationen

Initialisierte "Variablen", deren Wert man nicht mehr ändern kann

```
static final int MAX = 100;
```

Zweck: Lesbarere Namen für häufig verwendete Konstanten

```
int[] a = new int[100];
...
int sum;
for (int i = 0; i < 100; i++) {
   sum = sum + a[i];
}
double mean = sum / (double)100;
```

- Was bedeutet 100?
- Fehlergefahr bei Änderungen

```
static final int MAX = 100;
...
int[] a = new int[MAX];
...
int sum;
for (int i = 0; i < MAX; i++) {
   sum = sum + a[i];
}
double mean = sum / (double)MAX;
```

- Konstantenname MAX is lesbarer als 100
- bei Änderungen muss nur eine Stelle geändert werden

Konstantendeklaration muss auf Klassenebene stehen (siehe später)

2. Einfache Programme

Kommentare

Geben Erläuterungen zum Programm

Zeilenendekommentare

- beginnen mit //
- gehen bis zum Zeilenende

```
int sum;  // total sales
int totalSales;
```

Klammerkommentare

- durch /* ... */ begrenzt
- können über mehrere Zeilen gehen
- dürfen nicht geschachtelt werden
- oft zum "Auskommentieren" von Programmteilen

```
/* Das ist ein längerer
   Kommentar, der über
   mehrere Zeilen geht */
```

Sinnvoll kommentieren!

- alles kommentieren, was Erklärung bedarf
- statt unklares Programm mit Kommentar, besser klares Programm ohne Kommentar
- nicht kommentieren, was ohnehin schon im Programm steht;
  folgendes ist z.B. unsinnig

```
int sum; // Summe
```

Sprache in Kommentaren und Namen

Deutsch

- \+ einfacher

Englisch

- \+ meist kürzer
- \+ passt besser zu den englischen Schlüsselwörtern (if, while, ...)
- \+ Programm kann international verteilt werden (z.B. über das Web)

Jedenfalls: Deutsch und Englisch nicht mischen!!

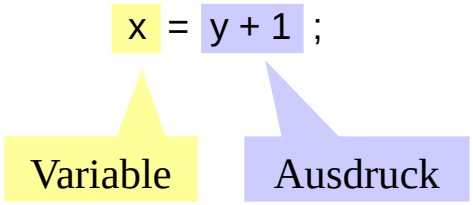
Namenswahl für Variablen und Konstanten

Einige Tipps

- aussagekräftige aber nicht zu lange Namen
  z.B. *sum*, *width* ~~*sumOfAllEntriesInInput*~~
- Hilfsvariablen, die man nur über kurze Strecken braucht, eher kurz:
  z.B. *i*, *j*, *x*
- Variablen, die man im ganzen Programm braucht, eher länger:
  z.B. *inputText*
- Variablennamen beginnen mit Kleinbuchstaben
  Worttrennung durch Großbuchstaben oder "_"
  z.B. *inputText*, *input_text*
- Konstantennamen ganz in Großbuchstaben
  z.B. *MAX_VALUE*
- Englisch oder Deutsch?

2. Einfache Programme

Zuweisungen



1. berechne den Ausdruck
2. speichere seinen Wert in der Variablen

Bedingung: linke und rechte Seite müssen zuweisungskompatibel sein

- müssen dieselben Typen haben, oder
- Typ $_{links}$ Ê Typ $_{rechts}$

Typhierarchie

double Ê float Ê long Ê int Ê short Ê byte

Statische Typenprüfung: Compiler prüft:

- dass Variablen nur erlaubte Werte enthalten
- dass auf Werte nur erlaubte Operationen ausgeführt werden

Beispiele für Zuweisungen

double Ê float Ê long Ê int Ê short Ê byte

Welche der folgenden Zuweisungen sind korrekt?

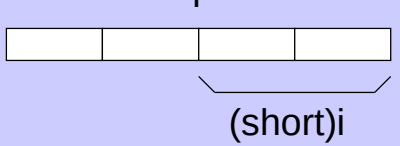
i = j;	✓	ok: derselbe Typ (*int*)
i = s;	✓	ok: *short* ist in *int* enthalten
s = i;	✗	falsch: *int* ist nicht in *short* enthalten
s = (short) i;	✓	ok: *int* wird in *short* umgewandelt
n = s;	✓	ok: *short* ist in *long* enthalten
d = i;	✓	ok: *int* ist in *double* enthalten
i = d;	✗	falsch: *double* ist nicht in *int* enthalten
i = (int) d;	✓	ok: Kommastellen werden abgeschnitten
i = 300;	✓	ok: Zahlkonstanten sind vom Typ *int*
b = 300;	✗	falsch: 300 passt nicht in *byte*
b = 1;	✓	ok: 1 (obwohl *int*) passt in *byte*
f = 2.5;	✗	falsch: 2.5 ist vom Typ *double*
f = 2.5f;	✓	ok: 2.5f ist vom Typ *float*
f = 2;	✓	ok: *int* ist in *float* enthalten

```
byte   b;
short  s;
int    i, j;
long   n;
float  f;
double d;
```

Typumwandlung (type cast)

(*type*) *expression*

- wandelt Typ von *expression* in *type* um
- dabei kann etwas abgeschnitten werden



2. Einfache Programme

Arithmetische Ausdrücke

Vereinfachte Grammatik

```
Expr       = Operand {BinaryOperator Operand}.
Operand = [UnaryOperator] ( ident | number | "(" Expr ")" ).
```

z.B.: - x + 3 * (y + 1)

Binäre Operatoren

+	Addition				
-	Subtraktion				
*	Multiplikation				
/	Division, Ergebnis ganzzahlig	5 / 3 = 1	(-5) / 3 = -1	5 / (-3) = -1	(-5) / (-3) = 1
%	Modulo (Divisionsrest)	5 % 3 = 2	(-5) % 3 = -2	5 % (-3) = 2	(-5) % (-3) = -2

Unäre Operatoren

+	Identität (+x = x)
-	Vorzeichenumkehr

Vorrangregeln

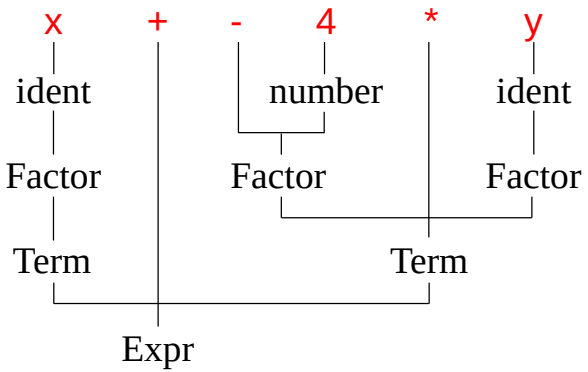
Wie in der Mathematik üblich

- Punktrechnung (*, /, %) vor Strichrechnung (+, -), z.B. 2 + 3 * 4 ergibt 14
- Operatoren auf gleicher Stufe werden von links nach rechts ausgewertet (a + b + c)
- Unäre Operatoren binden stärker als binäre, z.B.: 2 + -4 * 3 ergibt -10

Tatsächliche Grammatik (legt Vorrangregeln fest)

```
Expr    = Term {("+" | "-") Term}.
Term    = Factor {("*" | "/" | "%") Factor}.
Factor  = ["+" | "-" | "(" ident ")"] ( ident | number | "(" Expr ")" ).
```

Reihenfolge der Auswertung eines Ausdrucks



- zuerst - 4 => *t1*
- dann *t1* * y => *t2*
- dann x + *t2*

Typregeln in arithm. Ausdrücken

Operandentypen double Ê float Ê long Ê int Ê short Ê byte

Ergebnistyp Kleinster Typ, der alle Operandentypen enthält, aber zumindest *int*

Beispiele

```
s + s   // short + short   Þ int
s + 1   // short + int     Þ int
i + s   // int + short     Þ int
n + s   // long + short    Þ long
i + f   // int + float     Þ float
d + f   // double + float  Þ double
```

```
short  s;
int    i;
long   n;
float  f;
double d;
```

Welche der folgenden Zuweisungen sind korrekt?

`i = 2 * i + 1;`	✓ ok: Zuweisung von *int* an *int*
`s = s + 1;`	✗ falsch: Zuweisung von *int* an *short* nicht erlaubt
`s = (short)(s + 1);`	✓ ok: Zuweisung von *short* an *short*
`f = 2.5 * f + 1;`	✗ falsch: Zuweisung von *double* an *float* nicht erlaubt
`d = 2.5 * f + 1;`	✓ ok: Zuweisung von *double* an *double*

Increment und Decrement

Variablenzugriff kombiniert mit Erhöhung/Erniedrigung der Variablen

x++	nimmt den Wert von *x* und erhöht *x* anschließend um 1
++x	erhöht *x* um 1 und nimmt anschließend den erhöhten Wert
x--	nimmt den Wert von *x* und erniedrigt *x* anschließend um 1
--x	erniedrigt *x* um 1 und nimmt anschließend den erniedrigten Wert

Beispiele

```
x = 1;  y = x++ * 3;      // x = 2, y = 3
```
entspricht: `y = x * 3; x = x + 1;` *Postincrement*

```
x = 1;  y = ++x * 3;      // x = 2, y = 6
```
entspricht: `x = x + 1; y = x * 3;` *Preincrement*

Ist nicht schneller (!), sondern nur kürzer in der Schreibweise => möglichst vermeiden

Kann auch als eigenständige Anweisung verwendet werden

```
x = 1; x++;               // x = 2
```
entspricht: `x = x + 1;`

Darf nur auf *Variablen* angewendet werden (nicht auf *Ausdrücke*)

```
y = (x + 1)++;            // falsch!
```

Shift-Operationen

Binärdarstellung von Zahlen

x

0	0	0	0	0	0	1	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$2^0 + 2^1 = 1 + 2 = \mathbf{3}$

Negative Zahlen im *Zweierkomplement*

0	0	0	0	0	0	1	1	3
1	1	1	1	1	1	0	0	Einerkomplement
							+ 1	
1	1	1	1	1	1	0	1	Zweierkomplement

Shift-Operationen

Shift Left

x << 1

0	0	0	0	0	1	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$2^1 + 2^2 = 2 + 4 = \mathbf{6}$

x * 2

Shift Right

x >> 1

0	0	0	0	0	0	0	1
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1

$2^0 = \mathbf{1}$

x / 2

Multiplikation/Division mit Zweierpotenzen

Mit Shift-Operationen effizient implementierbar

Multiplikation	
x * 2	x << 1
x * 4	x << 2
x * 8	x << 3
x * 16	x << 4
...	...

Division	
x / 2	x >> 1
x / 4	x >> 2
x / 8	x >> 3
x / 16	x >> 4
...	...

Division nur bei *positiven* Zahlen durch Shift ersetzbar.
Bei *negativen* Zahlen wird auf die nächstkleinere ganze Zahl gerundet.

Beispiele

x = 5;
0000 0101

x = x << 2;
0001 0100
20
(16 + 4)
x * 4

x = - 5;
1111 1011
- (0000 0100 + 1) = -5

x = x << 2;
1110 1100
- 20
- (0001 0011 + 1)
x * 4

x = 5;
0000 0101

x = x >> 2;
0000 0001
1
(1)
x / 4

x = - 5;
1111 1011
- (0000 0100 + 1) = -5

x = x >> 2;
1111 1110
- 2
- (0000 0001 + 1)
ëx / 4û

ë-1.25û

Arithmetisches und Logisches Shift Right

Arithmetisches Shift Right

x >> 1

x = - 3;

1111 1101

x = x >> 1;

1111 1110

Vorzeichenbit wird reingeschoben

Logisches Shift Right

x >>> 1

x = - 3;

1111 1101

x = x >>> 1;

0111 1110

0 wird reingeschoben

Macht nur bei negativen Zahlen einen Unterschied
Wird nur in der Systemprogrammierung verwendet

Modulo-Operationen mit Zweierpotenzen

	Und-Verknüpfung mit
x % 2	0000 0001
x % 4	0000 0011
x % 8	0000 0111
x % 16	0000 1111
...	...

Beispiele

x = 21;

0001 0101

16 + 4 + 1

x % 2

```
  0001 0101
& 0000 0001
-----------
  0000 0001     = 1  (21 % 2)
```

x % 4

```
  0001 0101
& 0000 0011
-----------
  0000 0001     = 1  (21 % 4)
```

x % 8

```
  0001 0101
& 0000 0111
-----------
  0000 0101     = 5  (21 % 8)
```

Wird vom Compiler effizient durch *Maskierung* implementiert

Zuweisungsoperatoren

Arithmetischen Operationen lassen sich mit Zuweisung kombinieren

	Kurzform	*Langform*
+=	x += y;	x = x + y;
-=	x -= y;	x = x - y;
*=	x *= y;	x = x * y;
/=	x /= y;	x = x / y;
%=	x %= y;	x = x % y;

Spart Schreibarbeit, ist aber nicht schneller als die Langform

Mathematische Operationen

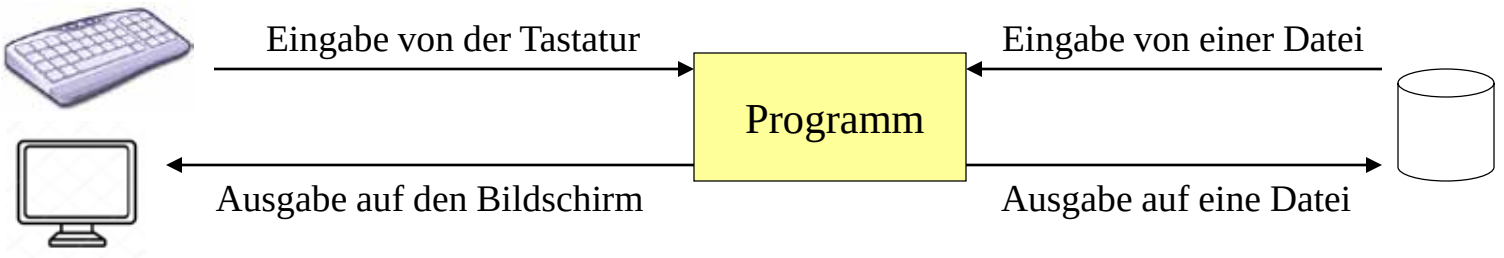
Klasse *Math*

`import java.lang.Math;`	Macht math. Operationen im Programm verfügbar
`d1 = Math.abs(d2);`	Absolutwert (funktioniert auch für *int* und *long*)
`d1 = Math.ceil(d2);`	nächstgrößere ganze Zahl
`d1 = Math.floor(d2);`	nächstkleinere ganze Zahl
`n = Math.round(d);`	nächstkleinere ganze Zahl
`n1 = Math.min(n2, n3);`	Minimun (für *int*, *long*, *float*, *double*)
`n1 = Math.max(n2, n3);`	Maximum (für *int*, *long*, *float*, *double*)
`d1 = Math.sqrt(d2);`	Maximum (für *int*, *long*, *float*, *double*)
`d1 = Math.exp(d2);`	e^{d2}
`d1 = Math.pow(d1, d2);`	$d1^{d2}$
`d1 = Math.log(d2);`	ln d2
`d1 = Math.log10(d2);`	$\log_{10}$ d2
`d1 = Math.sin(d2);`	Sinus (in Radianten: Winkel / 180 * Math.PI), dasselbe für *cos* und *tan*
`d1 = Math.asin(d2);`	Arcus Sinus (d2 in Radianten, Winkel = d1 * 180 / Math.PI), dasselbe für *acos* und *atan*

d ... *double*-Wert
n ... *long*-Wert

2. Einfache Programme

Eingabe und Ausgabe von Werten



Eingabe

```
int x = In.readInt();          // liest eine Zahl vom Eingabestrom
if (In.done()) ...             // liefert true oder false, je nachdem, ob Lesen erfolgreich war
In.open("MyFile.txt");         // öffnet angegebene Datei als neuen Eingabestrom
In.close();                    // schließt Datei und kehrt zum alten Eingabestrom zurück
```

Ausgabe

```
Out.print(x);                  // gibt x auf dem Ausgabestrom aus (x kann von bel. Typ sein)
Out.println(x);                // gibt x aus und beginnt eine neue Zeile
Out.open("MyFile.txt");        // öffnet angegebene Datei als neuen Ausgabestrom
Out.close();                   // schließt Datei und kehrt zum alten Ausgabestrom zurück
```

http://ssw.jku.at/JavaBuch

Beispiel

Mittelwertberechnung dreier Zahlen a, b, c

Eingabe von Tastatur
Ausgabe auf Bildschirm

```
int a = In.readInt();
int b = In.readInt();
int c = In.readInt();
double mean = (a + b + c) / 3.0;
Out.println("mean = " + mean);
```

String-Verkettung

Um double-Division zu erzwingen

Ein/Ausgabe auf Datei

```
In.open("input.txt");
Out.open("output.txt");
int a = In.readInt();
int b = In.readInt();
int c = In.readInt();
double mean = (a + b + c) / 3.0;
Out.println("mean = " + mean);
In.close();
Out.close();
```

Eingabe: **`3 25 15`**

Ausgabe: **`mean = 14.333333`**

Besonderheiten zur Eingabe

Eingabe von Tastatur

Eintippen von: 12 100 ↵ Return-Taste

füllt Lesepuffer

Programm:

```
int a = In.readInt();     // liest 12
int b = In.readInt();     // liest 100
int c = In.readInt();     // blockiert, bis Lesepuffer wieder gefüllt ist
```

Ende der Eingabe: Eingabe von *Strg-Z* in leerer Zeile

Eingabe von Datei

kein Lesepuffer, *In.readInt()* liest direkt von der Datei

Ende der Eingabe wird automatisch erkannt (kein *Strg-Z* nötig)

Eingabeoperationen

Klasse *In*

`int i = In.readInt();`	Liest eine ganze Zahl, z.B. -123
`long n = In.readLong();`	Liest eine ganze Zahl, z.B. 3000000000
`float f = In.readFloat();`	Liest eine Gleitkommazahl, z.B. 3.14 oder 0.314E1
`double d = In.readDouble();`	Liest eine Gleitkommazahl, z.B. 0.15E42
`String s = In.readString();`	Liest einen String unter Hochkommas, z.B. "Hello"
`char ch = In.read();`	Liest ein Zeichen, z.B. x
`String w = In.readWord();`	Liest ein Wort (alle Zeichen bis zum nächsten Leerzeichen)
`String ln = In.readLine();`	Liest eine Zeile (alle Zeichen bis zum nächsten Zeilenende)
`String fi = In.readFile();`	Liest die ganze Datei (bis zum Dateiende)
`In.open("myfile.txt");`	Öffnet Datei myfile.txt; weitere Eingabe von dieser Datei
`In.close();`	Schließt offene Datei; weitere Eingabe von Tastatur

Wenn Eingabe nicht geklappt hat (z.B. keine passendes Zahl oder Eingabe zu Ende), liefert In.done() anschließend *false* (Benutzung siehe später)

Ausgabeoperationen

Klasse *Out*

`Out.print(x);`	Gibt den Wert von x aus x kann vom Typ *byte*, *short*, *int*, *long*, *float*, *double*, *char*, *String* sein
`Out.println(x);`	Gibt den Wert von x aus und beginnt dann eine neue Zeile x kann vom Typ *byte*, *short*, *int*, *long*, *float*, *double*, *char*, *String* sein
`Out.open("myfile.txt");`	Öffnet Datei myfile.txt; weitere Ausgabe geht auf diese Datei
`Out.close();`	Schließt offene Datei; weitere Ausgabe geht auf den Bildschirm

Wenn Datei nicht geöffnet werden konnte,
liefert Out.done() anschließend *false* (Benutzung siehe später)

2. Einfache Programme

Grundstruktur von Java-Programmen

```
class ProgramName {

    public static void main (String[] arg) {
        ... // Deklarationen
        ... // Anweisungen
    }

}
```

Text muss in einer Datei namens *ProgramName*.java stehen

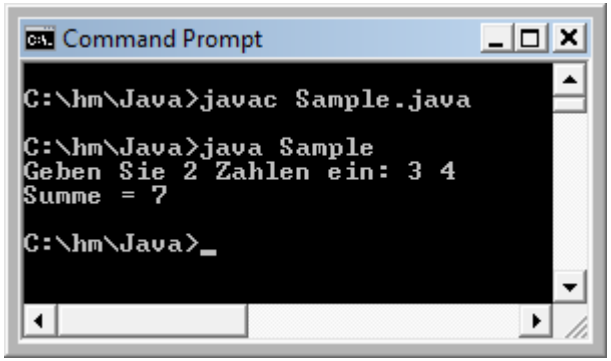
Beispiel

```
class Sample {

    public static void main (String[] arg) {
        Out.print("Geben Sie 2 Zahlen ein: ");
        int a = In.readInt();
        int b = In.readInt();
        Out.print("Summe = ");
        Out.println(a + b);
    }

}
```

Text steht in Datei *Sample.java*



Übersetzen und Ausführen mit JDK

Übersetzen

C:\> *cd MySamples*	wechselt ins Verzeichnis mit der Quelldatei
C:\MySamples> *javac Sample.java*	erzeugt Datei *Sample.class*

Ausführen

C:\MySamples> *java Sample*	ruft *main*-Methode der Klasse *Sample* auf
Geben Sie 2 Zahlen ein: *3 4*	Eingabe mit Return-Taste abschließen
Summe = 7	

Beispiel: Quadratische Gleichung

geg.: Koeffizienten *a*, *b*, *c* der quadratischen Gleichung $ax^2 + bx + c = 0$

ges.: Lösungen nach der Formel $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Datei *QuadraticEquation.java*

```
import java.lang.Math;

class QuadraticEquation {

   public static void main(String[] arg) {
      Out.print("Enter a, b, c: ");
      int a = In.readInt();
      int b = In.readInt();
      int c = In.readInt();
      double t1 = Math.sqrt(b * b - 4 * a * c);
      double t2 = 2.0 * a;
      Out.println("x1 = " + (-b + t1) / t2);
      Out.println("x2 = " + (-b - t1) / t2);
   }

}
```

Übersetzen und ausführen

```
javac QuadraticEquation.java
java QuadraticEquation
Enter a, b, c: 2 4 -16
x1 = 2.0
x2 = -4.0
```