

# ESOP - Einfache Programme

Assoc. Prof. Dr. Mathias Lux  
ITEC / AAU

# Agenda



- Grundsymbole
- Variablen, Konstanten
- Zuweisungen
- Operatoren

# Grundsymbole: Namen



Bezeichnen Variablen, Typen, ... in einem Programm

- bestehen aus Buchstaben, Ziffern und „\_“
- beginnen mit Buchstaben
- beliebig lang
- Groß-/Kleinschreibung signifikant
- Beispiele
  - x, x17, my\_Var, myVar

# Grundsymbole: Schlüsselwörter



- Heben Programmteile hervor
- Dürfen nicht als Namen verwendet werden
- Beispiele:
  - if, while, for, enum, class, static, ...

# Grundsymbole: Zahlen



- Ganze Zahlen
  - (dezimal oder hexadezimal)
- Gleitkommazahlen
- Beispiele
  - 376 ... dezimal
  - 0x1A5 ... hexadezimal
  - 3.14 ... Gleitkommazahl

# Grundsymbole: Zeichenketten



- Beliebige Zeichenfolgen zwischen Hochkommas
- Dürfen nicht über Zeilengrenzen gehen
- " in der Zeichenkette wird als \" geschrieben
- Beispiele
  - "a simple string"
  - "sie sagte \"Hallo\""

# Grundsymbole: Zeichenketten



- String ... Zeichenkette
  - Eigentlich kein Basisdatentyp, sondern ein Objekt!
- char ... ein einzelnes Unicode Zeichen
  - 2 Bytes
  - unter einfachem Hochkomma, z.B. 'L', ')', ...

# Variablendeklaration



- Jede Variable muss vor ihrer Verwendung deklariert werden
  - macht den Namen und den Typ der Variablen bekannt
  - Compiler reserviert Speicherplatz für die Variable
- Beispiele:
  - `int x;` ... deklariert eine Variable x vom Typ int (integer)
  - `short a, b;` ... deklariert 2 Variablen a und b vom Typ short (short integer)



# Ganzzahlige Typen



<b>byte</b>	8 bit	$-2^7 \dots 2^7-1$	(-128 .. 127)
<b>short</b>	16 bit	$-2^{15} \dots 2^{15}-1$	(-32.768 .. 32.767)
<b>int</b>	32 bit	$-2^{31} \dots 2^{31}-1$	(-2.147.483.648 .. )
<b>long</b>	64 bit	...	...

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

# Konstantendeklaration



- Keine Konstanten in Java.
- Initialisierte "Variablen", deren Wert man nicht mehr ändern kann
  - `final int max = 100;`
- Zweck
  - bessere Lesbarkeit
    - max ist lesbarer als 100
  - bessere Wartbarkeit
    - wenn die „Konstante“ mehrmals vorkommt und geändert werden muss, dann muss das nur an 1 Stelle erfolgen

# Kommentare



- Zeilenendekommentare
  - Beginnen mit `//` .. reichen bis zum Zeilenende (EOL)
- Klammerkommentare
  - durch `/* ... */` begrenzt, können über mehrere Zeilen gehen
- Kommentare & Lesbarkeit
  - alles kommentieren, was Erklärung bedarf
  - nicht kommentieren, was schon da ist;

```
// Hier ist ein Zeilenkommentar

int x = 15; // Initialisierung an dieser Stelle erforderlich!
short y = -12;

/* *****
   Dieses Programm wurde von Mathias Lux geschrieben
   ***** */
```

# Sprache für Kommentare & Namen







- Am “potentiellen Team” orientieren
  - Englisch besser als Deutsch
- Auf keinen Fall mischen
- Achtung bei
  - Schimpfworten, Emailadressen, Namen, Lizenzen!

Search

shit

Search

 Repositories	203
 Code	26,340
 Issues	2,815
 Users	31

## Languages

C	224,353
HTML	34,242
JavaScript	x
GAS	25,739
Less	23,338
C++	18,093
PHP	15,306
XML	9,977
Ruby	9,436
Markdown	8,795

[Advanced search](#) [Cheat sheet](#)

We've found 26,340 code results

Sort: Best match ▾

 [romankalb/PMapp – main.js](#) JavaScript  
Last indexed on 3 Aug

```
1 debug("Shit");
```

 [matthewwcv/nodestuff – mmcolors.js](#) JavaScript  
Last indexed on 31 Jul

```
1 alert('shit');
```

 [lwl8851206/HelloWorld – test.js](#) JavaScript  
Last indexed on 29 Jul

```
1 function shit(){};
```

 [ACSysFMI/theDoctors – shit.js](#) JavaScript  
Last indexed on 25 Jul

```
1 alert('shit');
```

 [nitirajrathore/testrepo – tits.js](#) JavaScript  
Last indexed on 2 Aug

```
1 alert("fucking dick shit");
```

 [gpestana/legacy\\_slick.js – core\\_tests.js](#) JavaScript  
Last indexed on 1 Aug

```
1 /*Tests and shit.. */
```

 [bmelon11/myrepo – boo.js](#) JavaScript  
Last indexed on 28 Jul

```
1 console.log("eat shit");
```

 [apiengine/apiengine-client – page.js](#) JavaScript  
Last indexed on 23 Jul

```
1 some profile shit goes here
```

 [AchintyaAshok/NYT---Intern-Project-Front-End – storyView.js](#) JavaScript  
Last indexed on 28 Jul

```
1 console.log('django is shit');
```

 [JamieAppleseed/jamieappleseed.com – application.js](#) JavaScript  
Last indexed on 8 Aug

```
1 (function(win){
2   // do shit
3 })(this);
```



# Namenswahl für Variablen & Konstanten



- Coding Conventions existieren für
  - Lesbarkeit über Teams hinweg
  - Wartbarkeit & Preservation
- Naming Conventions siehe:  
<http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html#367>
- Tipps:
  - Bedeutungsvolle Namen (vgl. Kommentare)
  - Eher kurz als lang, aber IDE unterstützt.

# Schlechte Beispiele ...



[aeonsff/footlocker – dpc-hashcrack.py](#)

Last indexed on 31 Jul

Python

```
51         return licker
52
53     def toptobottom(crack):
54         i = 0
55         while i < (len(asshole)/2):
56             if len(crack) == 32:
57                 if crack == md5(asshole[i]):
58
59
60                 if crack == md5(asshole[i]):
61                     print "\n\t[p1] 3===D passwd is = %s\n"%asshole[i]
62                     break
63                 elif len(crack) == 40:
```



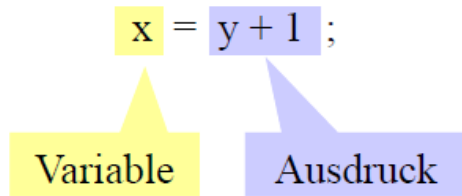
[hallfox/teampython – ex10b.py](#)

Last indexed on 3 Aug

Python

```
5     escape4 = "%s is a total asshole."
6
7     asshole = "Tyler \t\nFUCK\n Sontag \\"
8
9     singlequoteformatting = '''
10    This looks a lot cleaner and minimalistic.
11
12    For now on, let's use the single quotes instead.
13    '''
14
15    print escape1
16    print escape2
17    print escape3
18    print escape4 % asshole
19    print singlequoteformatting
```

# 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  $\supseteq$  Typ rechts
- Hierarchie der ganzzahligen Typen
  - long  $\supseteq$  int  $\supseteq$  short  $\supseteq$  byte



# Zuweisungen



- Beispiele

```
int i, j; short s; byte b;
```

```
i = j;           // ok: derselbe Typ
```

```
i = 300;         // ok (Zahlkonstanten sind int)
```

```
b = 300;         // nicht ok: 300 passt nicht in byte
```

```
i = s;          // ok
```

```
s = i;          // nicht ok
```

# Statische Typenprüfung



- Compiler prüft:
  - dass Variablen nur erlaubte Werte enthalten
  - dass auf Werte nur erlaubte Operationen ausgeführt werden

# Arithmetische Ausdrücke



- Vereinfachte Grammatik

`Expr = Operand {BinaryOperator Operand}.`

`Operand = [UnaryOperator] ( identifier | number | "(" Expr ")" ).`

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

# Arithmetische Ausdrücke



## 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

# Typregeln in arithmetischen Ausdrücken



- Vorrangregeln
  - Punktrechnung ( $*$ ,  $/$ ,  $\%$ ) vor Strichrechnung ( $+$ ,  $-$ )
    - z.B.  $2 + 3 * 4 = 14$
  - Linksassoziativität
    - z.B.  $7 - 3 - 2 = 2$
  - Unäre Operatoren binden stärker als binäre
    - z.B.:  $-2 * 4 + 3$  ergibt  $-5$
- Typregeln
  - Operandentypen - byte, short, int, long
  - Ergebnistyp - wenn mindestens 1 Operand long  $\rightarrow$  long, sonst  $\rightarrow$  int

# Beispiele



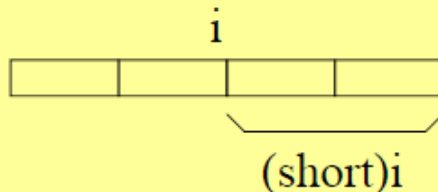
```
short s; int i; long x;
```

```
x = x + i;           // long
i = s + 1;           // int (1 ist vom Typ int)
s = s + 1;           // nicht ok!
s = (short)(s + 1);  // Typumwandlung nötig
```

## Typumwandlung (type cast)

*(type)expression*

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



# Increment / Decrement



- Variablenzugriff kombiniert mit Addition/Subtraktion
  - `x++` ... nimmt den Wert von `x` und erhöht `x` anschließend um 1
  - `++x` ... erhöht `x` um 1 und nimmt anschließend den erhöhten Wert
  - `x--`, `--x` ... entsprechend
- Kann auch als eigenständige Anweisung verwendet werden
  - `x = 1; x++;` // `x = 2` entspricht: `x = x + 1;`
- Beispiele
  - `x = 1; y = x++ * 3;` // `x = 2, y = 3` entspricht: `y = x * 3; x = x + 1;`
  - `x = 1; y = ++x * 3;` // `x = 2, y = 6` entspricht: `x = x + 1; y = x * 3;`
- Darf nur auf Variablen angewendet werden (nicht auf Ausdrücke)
  - `y = (x + 1)++;` // Fehler!

# Multiplikation/Division mit Zweierpotenzen



Mit Shift-Operationen effizient implementierbar

## *Multiplikation*

$x * 2$	$x \ll 1$
$x * 4$	$x \ll 2$
$x * 8$	$x \ll 3$
$x * 16$	$x \ll 4$
...	...

## *Division*

$x / 2$	$x \gg 1$
$x / 4$	$x \gg 2$
$x / 8$	$x \gg 3$
$x / 16$	$x \gg 4$
...	...

Division nur bei  
positiven Zahlen  
durch Shift ersetzbar



# Multiplikation/Division mit Zweierpotenzen



## Beispiele

`x = 3;`

0000 0011

`x = x << 2; // 12`

0000 1100

`x = -3;`

1111 1101

`x = x << 1; // -6`

1111 1010

`x = 15;`

0000 1111

`x = x >> 2; // 3`

0000 0011

*Java verwendet die Zweierkomplementdarstellung!*

# Zuweisungsoperatoren



- Arithmetischen Operationen lassen sich mit Zuweisung kombinieren

	<i>Kurzform</i>	<i>Langform</i>
<code>+=</code>	<code>x += y;</code>	<code>x = x + y;</code>
<code>-=</code>	<code>x -= y;</code>	<code>x = x - y;</code>
<code>*=</code>	<code>x *= y;</code>	<code>x = x * y;</code>
<code>/=</code>	<code>x /= y;</code>	<code>x = x / y;</code>
<code>%=</code>	<code>x %= y;</code>	<code>x = x % y;</code>

# String-Operatoren



- Strings können mit ‘+’ verknüpft werden
  - “Mathias” + “ “ + “Lux”
- Andere Operatoren gelten nicht für Strings
  - Vor allem nicht Prüfung auf Gleichheit
  - “Mathias” != “Lux” ... vergleicht Adressen!

# Bit-Operatoren



- Die Bits der Operanden werden miteinander verknüpft.
  - Beispiel (Java nutzt Zweierkomplement)
    - `byte a = 17; // 00010001`
    - `byte b = 7; // 00000111`
- Eine Eins steht im Ergebnis genau dort, wo...
  - Disjunktion: ... einer der Operanden eine Eins aufweist
    - `byte or = a | b; // 23`
  - Konjunktion: ... beide Operanden eine Eins aufweisen
    - `byte and = a & b; // 1`
  - Antivalenz: ... die Operanden unterschiedlich sind
    - `byte xor = a ^ b; // 22`
  - Komplement: ... der Operand eine Null aufweist
    - `byte notB = ~b; // -8`

# Grundstruktur von Java-Programmen



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

// Beispiel:

```
class Sample {  
    public static void main (String[] arg) {  
        int a = 23;  
        int b = 100;  
        System.out.print("Summe = ");  
        System.out.println(a + b);  
    }  
}
```

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

```
C:\Windows\system32\cmd.exe  
E:\Temp>javac Sample.java  
E:\Temp>java Sample  
Summe = 123  
E:\Temp>
```

# Übersetzen und Ausführen mit JDK



- Übersetzen

- C:\> cd MySamples  
wechselt zu Quelldatei
- C:\MySamples> javac Sample.java  
erzeugt Datei Sample.class

- Ausführen

- C:\MySamples> java Sample  
ruft main-Methode der Klasse Sample auf
- Summe = 123

# Beispiel: IDEA IDE



- Beispiele für Kommentare
  - Rechtschreibprüfung
- Live Templates
  - psvm + <tab>
- Automatische Benennung von Variablen
  - <Strg>-<Leertaste>