Einführung in die Informatik 1

– Elementare Konzepte von Java –

Sven Kosub

AG Algorithmik/Theorie komplexer Systeme
Universität Konstanz

E 202 | Sven.Kosub@uni-konstanz.de | Sprechstunde: Freitag, 12:30-14:00 Uhr, o.n.V.

Wintersemester 2008/2009

Hello, world!

`Test.java`:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : "  "+ args[i]);
    System.out.println();
  }
}
```

Beachte:

- Inhalt von `Test.java` ist der Quelltext
- Quelltext mit beliebigem (ASCII-/Unicode-)Editor erstellbar
- Endung `.java` zwingend zur Kennzeichnung von Java-Quelltexten

Hello, world!

Erinnerung:

- Java-Programme werden sowohl übersetzt als auch interpretiert
- Java unterteilt Hochsprachenschicht in zwei Schichten
- Java-Programme werden in speziellen Byte-Code übersetzt
- Byte-Code wird von JVM (*Java virtual machine*) interpretiert
- JVM und Systembibliotheken pro Rechnertyp einmal entwickelt

Ausführung von `Test.java` in zwei Phasen:

- `javac Test.java`
- `java Test Hello, world!`

Hello, world!

Übersetzung:

- Aufruf des Java-Compilers `javac`
- Compiler produziert für jede Klassendeklaration eine Datei
- Datei hat den Namen der Klasse und die Endung `.class`
- Datei enthält den Byte-Code für die Klasse

`javac Test.java` produziert nur die Datei `Test.class`

Hello, world!

Interpretation:

- Aufruf der Java-Laufzeitumgebung `java`
- Klasse muss Methode `main` enthalten
- alles Weitere hängt vom Programminhalt ab

`java Test Hello, world!` ergibt die Ausgabe `Hello, world!`

Beachte:

- Compiler: `javac Test.java` mit Endung
- Laufzeitumgebung: `java Test` ohne Endung

Hello, world!

Elemente des Quelltextes:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : "  "+ args[i]);
    System.out.println();
  }
}
```

- Deklaration einer Objektklasse `Test` durch Schlüsselwort `class`
- Objektvariablen und -methoden innerhalb von { und }

Hello, world!

Elemente des Quelltextes:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : "  "+ args[i]);
    System.out.println();
  }
}
```

- Deklaration der Methode `main` (Struktur ist immer gleich!)
- `public` legt fest, dass `main` von allen Objekten aufrufbar ist
- `static` legt fest, dass `main` eine Klassenmethode ist
- `void` legt fest, dass `main` keinen Wert zurück gibt

Hello, world!

Elemente des Quelltextes:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : "  "+ args[i]);
    System.out.println();
  }
}
```

- obligatorischer Eingabeparameter für `main`
- `String[]` legt fest, dass `args` Array vom Typ `String` ist
- dient zur übermittlung von Eingaben beim Aufruf von `Test` von außen
- kann innerhalb von `main` ignoriert werden

Hello, world!

Elemente des Quelltextes:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : "  "+ args[i]);
    System.out.println();
  }
}
```

- `for`-Schleife mit festgelegten Grenzen
- `int i=0` legt fest, dass Schleifenvariable `i` vom Typ `int` ist und mit 0 initialisiert ist
- `i<args.length` legt fest, dass in allen Durchläufen `i` kleiner als Anzahl `args.length` der Elemente von `args` ist
- `i++` legt fest, dass `i` nach jedem Durchlauf inkrementiert wird
- Anweisung hinter `for (...)` wird in jedem Durchlauf ausgeführt

Hello, world!

Elemente des Quelltextes:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : "  "+ args[i]);
    System.out.println();
  }
}
```

- vordefinierte Druckmethoden `print` und `println`
- gehören zur vordefinierten Klassenvariable `out` der vordefinierten Klasse `System`
- `print` gibt Parameter als Zeichenkette in aktueller Konsolenzeile
- `println` gibt Parameter als Zeichenkette in aktueller Konsolenzeile mit Wagenrücklauf und Zeilenvorschub aus

Hello, world!

Elemente des Quelltextes:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
              System.out.print(i == 0 ? args[i] : " "+ args[i]);
    System.out.println();
  }
}
```

- bedingter Ausdruck zur Parameterbeschreibung
- Ist Bedingung `i==0` wahr, so ist Parameter `args[i]` sonst `" "+args[i]`
- `args[i]` ist *i*-tes Element im Array `args` (Zählung beginnt bei 0)
- `" "` ist Zeichenkette, die nur aus Leerzeichen besteht
- `+` ist Konkatenation von Zeichenketten

Hello, world!

Elemente des Quelltextes:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
              System.out.print(i == 0 ? args[i] : " "+ args[i]);
    System.out.println();
  }
}
```

Was passiert beim Aufruf `java Test Hello, world!` ?

- Parameter `Hello, world!` wird in durch Leerzeichen getrennte Einzelparameter zerlegt
- Parameterübergabe als `args[0]="Hello,"`, `args[1]="world!"` und `args.length=2`
- `main` wird mit `args` (wie beschrieben) gestartet
- ...

Hello, world!

```
...
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " "+ args[i]);
    System.out.println();
...
```

Was passiert beim Aufruf `java Test Hello, world!` ? (Fort.)

- `for`-Schleife wird mit Durchlauf für $i = 0$ gestartet:
  - es gilt $0 < 2$ (`i<args.length` ist wahr)
  - wegen $i = 0$ wird `System.out.print(args[0])` ausgeführt
  - `Hello,` wird auf aktueller Zeile ausgegeben
  - Ausgabe auf aktueller Zeile ist jetzt `Hello,`
  - i wird inkrementiert, d.h. i wird auf 1 gesetzt
- ...

Hello, world!

```
...
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " "+ args[i]);
    System.out.println();
...
```

Was passiert beim Aufruf `java Test Hello, world!` ? (Fort.)

- `for`-Schleife wird mit Durchlauf für $i = 1$ fortgesetzt:
  - es gilt $1 < 2$ (`i<args.length` ist wahr)
  - wegen $i = 1$ wird `System.out.print(" "+args[1])` ausgeführt
  - `world!` wird auf aktueller Zeile ausgegeben
  - Ausgabe auf aktueller Zeile ist jetzt `Hello, world!`
  - i wird inkrementiert, d.h. i wird auf 2 gesetzt
- ...

Hello, world!

```
...
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " "+ args[i]);
    System.out.println();
...
```

Was passiert beim Aufruf `java Test Hello, world!` ? (Fort.)

- `for`-Schleife wird mit Durchlauf für $i = 2$ fortgesetzt:
  - es gilt nicht $2 < 2$ (`i<args.length` ist falsch)
  - `for`-Schleife wird beendet ohne Ausführung von `System.out.print(...)`
- `System.out.println();` wird ausgeführt, d.h. leeres Wort wird auf aktueller Zeile ausgegeben und Ausgabe auf Zeile mit Wagenrücklauf und Zeilenvorschub beendet
- Ausgabe ist damit `Hello, world!`
- `main` und Laufzeitumgebung werden beendet

Compilerphasen

- **Lexikalische Analyse:**
  Scanning, Identifikation bedeutungstragender Elemente (Tokens) des Quelltextes, Erkennung von Terminalsymbolen
- **Syntaktische Analyse:**
  Parsing, Herleitung von Nichtterminalsymbolen, Überprüfung des Quellcodes auf strukturelle Korrektheit (als passend zur Grammatik) und Umwandlung des Quellcodes in Syntaxbaum
- **Semantische Analyse:**
  Überprüfung des statischen Semantik (Deklaration von Variablen, Verträglichkeit von Datentypen bei Zuweisungen), Bestimmung eines attributierten Syntaxbaums
- **Code-Erzeugung**

Kontextfreie Grammatiken

Programmiersprachen werden durch kontextfreie Grammatiken definiert

Beispiel für eine Anweisungsdefinition (in Java):

Statement :
`if` (*Expression*) *Statement* ;

Interpretation: `if`-Anweisungen dürfen geschachtelt werden

Kontextfreie Grammatiken

Programmiersprachen werden durch kontextfreie Grammatiken definiert

Elemente einer kontextfreien Grammatik (nur grob):

- Terminale: werden durch `True-type`-Schrift gekennzeichnet
- Nichtterminale: werden durch *Kursiv*-Schrift gekennzeichnet
- Produktionsregeln: geben an, wie Nichtterminale durch gemischte Folge von Terminalen und Nichtterminalen ersetzt werden kann
- Startsymbol: ein bestimmtes Nichtterminal
- ...

Kontextfreie Grammatiken

DecimalIntegerLiteral :	Startsymbol	Regel 1
0	Terminal	
NonZeroDigit	Nichtterminal	
NonZeroDigit Digits	Folge von Nichtterminalen	
Digits :	Nichtterminal	Regel 2
Digit	Nichtterminal	
Digits Digit	Folge von Nichtterminalen	
Digit :	Nichtterminal	Regel 3
0	Terminal	
NonZeroDigit	Nichtterminal	
NonZeroDigit : ein Terminal aus	Nichtterminal	Regel 4
1 2 3 4 5 6 7 8 9	Terminale	

Kontextfreie Grammatiken

Programmiersprachen werden durch kontextfreie Grammatiken definiert

Elemente einer kontextfreien Grammatik (nur grob):

- ...
- Anwendung der Produktionsregeln produziert aus Folge von Terminalen und Nichtterminalen eine neue Folge von Terminalen und Nichtterminalen (Ableitung)
- auf Folge, die nur aus Terminalen besteht, kann keine Regel angewendet werden
- von Grammatik erzeugte Sprache: alle Terminalfolgen, die durch Anwendung endlich vieler Regeln aus Startsymbol entstehen (abgeleitet werden können)
- kontextfreie Sprache: von kontextfreier Grammatik erzeugt

Kontextfreie Grammatiken

DecimalIntegerLiteral :
- 0
- *NonZeroDigit* $Digits_{opt}$

Digits :
- *Digit*
- *Digits Digit*

Digit :
- 0
- *NonZeroDigit*

NonZeroDigit : ein Terminal aus

1 2 3 4 5 6 7 8 9

DecimalIntegerLiteral

$\Longrightarrow$ *NonZeroDigit Digits* (Regel 1)

$\Longrightarrow$ 2 *Digits* (Regel 4)

$\Longrightarrow$ 2 *Digits Digit* (Regel 2)

$\Longrightarrow$ 2 *Digits NonZeroDigit* (Regel 3)

$\Longrightarrow$ 2 *Digits* 9 (Regel 4)

$\Longrightarrow$ 2 *Digits Digit* 9 (Regel 2)

$\Longrightarrow$ 2 *Digits NonZeroDigit* 9 (Regel 3)

$\Longrightarrow$ 2 *Digits* 1 9 (Regel 3)

$\Longrightarrow$ 2 *Digit* 1 9 (Regel 2)

$\Longrightarrow$ 2 0 1 9 (Regel 3)

Grammatik erzeugt alle Dezimalzahlen ohne führende 0 (außer 0 selbst)

Lexikalische Struktur von Java

(vereinfachter) Symbolvorrat von Java:

UnicodeInputCharacter :
EscapeSequence
RawInputCharacter

RawInputCharacter :
irgendein über Tastatur erzeugbares Unicode-Zeichen

EscapeSequence :

`\ b`	Rückschritt (*backspace*)
`\ t`	(horizontaler) Tabulator (*tabulator*)
`\ n`	Zeilenvorschub (*line feed, new line*)
`\ f`	Seitenvorschub (*form feed*)
`\ r`	Wagenrücklauf (*carriage return*)
`\ "`	doppeltes Anführungszeichen
`\ '`	einfaches Anführungszeichen
`\ \`	umgekehrter Schrägstrich (*backslash*)

Lexikalische Struktur von Java

Zeilenumbrüche:

LineTerminator :
ASCII-Zeichen für Zeilenvorschub (*new line*, *line feed*, LF, 10, `\n`, `\u000a`)
ASCII-Zeichen für Wagenrücklauf (*carriage return*, CR, 13, `\r`, `\u000d`)
ASCII-Zeichen für Wagenrücklauf gefolgt von Zeilenvorschub

InputCharacter :
UnicodeInputCharacter außer LF, CR

Vorbereitungsphasen der lexikalischen Analyse:

- erste Phase der lexikalischen Analyse überprüft, ob Text dem Symbolvorrat entspricht
- zweite Phase der lexikalischen Analyse bestimmt Zeilenumbrüche

Lexikalische Struktur

Java-Programm:

Input :
InputElements$_{opt}$ *Eof*$_{opt}$

InputElements :
InputElement
InputElements InputElement

Eof :
Unicode-Zeichen für Dateiende (*end of file*, EOF, „control-Z“, 26, `\u001a`)

Bedeutung:

- Programm ist Folge von Eingabeelementen beliebiger Länge (inkl. leeres Programm)
- dritte Phase der lexikalischen Analyse unterscheidet Eingabeelemente

Lexikalische Struktur

InputElement :

WhiteSpace	Zwischenräume
Comment	Kommentare
Token	Tokens

Token :

Identifier	Bezeichner
Keyword	Schlüssewörter
Literal	Literale
Separator	Separatoren
Operator	Operatoren

Bedeutung:

- Tokens sind Terminale für die Syntax-Grammatik von Java
- Zwischenräume und Kommentare trennen Tokens

- Symbolfolge -= beschreibt einen Operatortoken
- Symbolfolge - = beschreibt zwei Operatortokens (- und =)

Lexikalische Struktur

Zwischenräume:

WhiteSpace :
- Unicode-Zeichen für Leerraum (*space*, SP, 32, 20_{16},)
- Unicode-Zeichen für Tabulator (*tabulator*, HT, 9, `\t`)
- Unicode-Zeichen für Seitenvorschub (*form feed*, FF, 12, $0c_{16}$, `\f`)
- *LineTerminator*

Kommentare:

Comment :
- / / $CharactersInLine_{opt}$

CharactersInLine :
- *InputCharacter*
- *CharactersInLine InputCharacter*

`//// steht für Kommentar` steht für Kommentar: „//steht für Kommentar“

Lexikalische Struktur

Bezeichner:

Identifier :
IdentifierChars außer *Keyword*, *BooleanLiteral* oder *NullLiteral*

IdentifierChars :
JavaLetter
IdentifierChars JavaLetterOrDigit

JavaLetter :
irgendein Unicode-Zeichen, das als Buchstabe erkannt wird

JavaLetterOrDigit :
irgendein Unicode-Zeichen, das als Buchstabe oder Ziffer erkannt wird

Bedeutung von Bezeichnern:

- stehen für Selbstdefiniertes (Variablen, Klassen, Methoden)
- Folge von Buchstaben und Ziffern beginnend mit Buchstaben

```
String   i3   MAX_VALUE   isLetterOrDigit
```

Lexikalische Struktur

Namenskonventionen (nicht Bestandteil der lexikalischen Grammatik):

- Namen für Klassen beginnen mit Großbuchstaben
- Name für Variablen und Methoden beginnen mit kleinem Buchstaben
- bei Zusammensetzungen beginnen neue Wörter wieder mit Großbuchstaben
- Konstanten werden komplett groß geschrieben

- `String` ist Klassenname
- `i3` ist Variablenname
- `MAX_VALUE` ist Konstantenname
- `isLetterOrDigit` ist Methodenname

Lexikalische Struktur

Schlüsselwörter:

Keyword : ein Terminal aus

`abstract`	`double`	`interface`	`switch`
`assert`	`else`	`long`	`synchronized`
`boolean`	`extends`	`native`	`this`
`break`	`final`	`new`	`throw`
`byte`	`finally`	`package`	`throws`
`case`	`float`	`private`	`transient`
`catch`	`for`	`protected`	`try`
`char`	`goto`	`public`	`void`
`class`	`if`	`return`	`volatile`
`const`	`implements`	`short`	`while`
`continue`	`import`	`static`	
`default`	`instanceof`	`strictfp`	
`do`	`int`	`super`	

Lexikalische Struktur

Literale:

Literal :

IntegerLiteral	ganze Zahlen
FloatingPointLiteral	Gleitkommazahlen
BooleanLiteral	Wahrheitswerte
CharacterLiteral	Buchstaben, Zeichen
StringLiteral	Zeichenketten, Strings, Wörter
NullLiteral	Nullreferenz

Bedeutung von Literalen:

- dienen der Beschreibung von Konstanten und konkreten Werten
- Verwendung bei elementaren Datentypen, Zeichenketten und Nullreferenz

Lexikalische Struktur

ganze Zahlen:

IntegerLiteral :
- *DecimalIntegerLiteral*
- *HexIntegerLiteral*

DecimalIntegerLiteral :
- `0`
- *NonZeroDigit* $Digits_{opt}$

Digits :
- *Digit*
- *Digits Digit*

Digit :
- `0`
- *NonZeroDigit*

NonZeroDigit : ein Terminal aus

`1 2 3 4 5 6 7 8 9`

HexIntegerLiteral :
- `0 x` *HexDigits*
- `0 X` *HexDigits*

HexDigits :
- *HexDigit*
- *HexDigit HexDigits*

HexDigit : ein Terminal aus

`0 1 2 3 4 5 6 7 8 9 a`
`b c d e f A B C D E F`

Lexikalische Struktur

`0  2  0xDadaCafe  2009  0x00FF00FF` sind `int`-Literale

Beachtung:

- 2147483648 ($= 2^{31}$) ist größtes `int`-Literal (als Dezimalzahl)
- 2147483648 darf nur mit - verwendet werden wegen Zweierkomplement (sonst Fehlermeldung)
- `0x7fffffff` größtes positives `int`-Literal (als Hexadezimalzahl)
- `0x80000000` größtes negatives `int`-Literal (als Hexadezimalzahl)
- `0xffffffff` repräsentiert Dezimalzahl -1

Lexikalische Struktur

Gleitkommazahlen:

FloatingPointLiteral :
Digits . $Digits_{opt}$ $ExponentPart_{opt}$
. $Digits_{opt}$ $ExponentPart_{opt}$
Digits ExponentPart

ExponentPart :
ExponentIndicator SignedInteger

ExponentIndicator : ein Terminal aus
`e` `E`

SignedInteger :
$Sign_{opt}$ *Digits*

Sign : ein Terminal aus
`+` `-`

`1e1` `2.` `.3` `0` `3.14` `6.022137e+23` sind `float`-Literale

Lexikalische Struktur

Wahrheitswerte:

BooleanLiteral : ein Terminal aus

`true` `false`

Buchstaben:

CharacterLiteral :

' *SingleCharacter* '

' *EscapeSequence* '

SingleCharacter :

InputCharacter außer ' oder \

`'a'` `'%'` `'\t'` `'\\'` `'\''` sind Buchstabenliterale

Lexikalische Struktur

Wörter:

StringLiteral :
" *StringCharacters*$_{opt}$ "

StringCharacters :
StringCharacter
StringCharacters StringCharacter

StringCharacter :
InputCharacter außer " oder \
EscapeSequence

- `"Die ist eine Zeichenkette"` ist ein Zeichenkettenliteral
- `""` ist Literal für die leere Zeichenkette
- `"\""` ist Literal für die Zeichenkette, die lediglich aus " besteht

Lexikalische Struktur

Nullreferenz:

NullLiteral :

`null`

Separatoren:

Separator : ein Terminal aus

`(` `)` `{` `}` `[` `]` `;` `,` `.` `?` `:`

Operatoren:

Operator : ein Terminal aus

`=`	`>`	`<`	`!`	`~`						
`==`	`<=`	`>=`	`!=`	`&&`	`||`	`++`	`--`			
`+`	`-`	`*`	`/`	`&`	`|`	`^`	`%`	`<<`	`>>`	`>>>`
`+=`	`-=`	`*=`	`/=`	`&=`	`|=`	`^=`	`%=`	`<<=`	`>>=`	`>>>=`

Lexikalische Struktur

Tokenstruktur von `Test.java`:

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
    System.out.println();
  }
}
```

Lexikalische Struktur

Tokenstruktur von `Test.java`: Bezeichner

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
    System.out.println();
  }
}
```

Lexikalische Struktur

Tokenstruktur von `Test.java`: Schlüsselwörter

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
    System.out.println();
  }
}
```

Lexikalische Struktur

Tokenstruktur von `Test.java`: Literale

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
    System.out.println();
  }
}
```

Lexikalische Struktur

Tokenstruktur von `Test.java`: Separatoren

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
    System.out.println();
  }
}
```

Lexikalische Struktur

Tokenstruktur von `Test.java`: Operatoren

```
class Test {
  public static void main(String[] args) {
    for (int i=0; i<args.length; i++)
            System.out.print(i == 0 ? args[i] : " " + args[i]);
    System.out.println();
  }
}
```

Elementare Datentypen

elementare Datentypen:

PrimitiveType :
    NumericType
    `boolean`

NumericType :
    IntegralType
    FloatingPointType

IntegralType : ein Terminal aus
    `byte` `short` `int` `long` `char`

FloatingPointType : ein Terminal aus
    `float` `double`

Beachte: Elementare Datentypen sind lexikalisch Schlüsselwörter

Elementare Datentypen

`boolean`	Wahrheitswerte `true` oder `false`
`byte`	8-Bit-Zahl in Zweierkomplementdarstellung
`short`	16-Bit-Zahl in Zweierkomplementdarstellung
`int`	32-Bit-Zahl in Zweierkomplementdarstellung
`long`	64-Bit-Zahl in Zweierkomplementdarstellung
`char`	16-Bit-Unicode-Zeichen (untere 8 Bit als ASCII)
`float`	32-Bit-Gleitkommazahl
`double`	64-Bit-Gleitkommazahl

Elementare Datentypen

Subtyp:

- numerischer Datentyp S ist Subtyp von T, falls Wertebereich von S in dem von T enthalten ist
- numerischer Datentyp T kann überall dort verwendet werden, wo Subtyp S verlangt ist

automatische Typkonversionen:

- `byte` $\subseteq$ `short` $\subseteq$ `int` $\subseteq$ `long`
- `float` $\subseteq$ `double`
- `long` $\subseteq$ `float`
- `char` $\subseteq$ `int`
- `int` $\subseteq$ `char`

- `float x=0xffffffff; System.out.println(x);` ergibt -1.0
- `int x='a'; System.out.println(x);` ergibt 97

Elementare Datentypen

Hüllenklassen:

- zu jedem elementaren Datentyp eine Hüllenklasse
- Bündelung von Konstanten und Methoden für elementare Datentypen
- zur Verwendung von Daten elementarer Datentypen als Datenobjekte

elementarer Datentyp	Hüllenklasse
`boolean`	`Boolean`
`byte`	`Byte`
`short`	`Short`
`int`	`Integer`
`long`	`Long`
`char`	`Character`
`float`	`Float`
`double`	`Double`

Beachte: Hüllenklassen sind lexikalisch Bezeichner

Variablen

Variablen dienen Speicherung von Werten

- Name bezeichnet die Variable im Programm
- Wert ist ein Element eines Datentyps
- Typ legt erlaubte Werte und Codierungen fest
- Definition:

Variable :
Type Identifier

- `int x=17;` — `x` ist Variable vom Typ `int` mit Wert 17
- `String s;` — `s` ist Variable vom Typ `String` (ohne Initialisierung)
- `String[] s;` — `s` ist Array-Variable vom Typ `String` (ohne Initialisierung)
- `Object obj;` — `obj` ist Variable vom Typ `Object` (ohne Initialisierung)

Variablen

Variablen (intern)

- Variablen abstrahieren Konzept der Speicherstelle (Adresse, Inhalt)
- Compiler ordnet Variablen Speicherstelle zu (Referenz, Zeiger):

$$\begin{aligned} \text{Name} &\mapsto \text{Adresse} \\ \text{Wert (Literal)} &\mapsto \text{Inhalt} \end{aligned}$$

Betrachten Deklaration: `int x=17;`

- Wert der Variable mit Namen `x` ist 17
- Referenz ist die Adresse 203, an der der Wert 17 gespeichert wird
- tatsächlich wird 10001_2 gespeichert (ohne führende Nullen)

Variablen

Referenzvariablen

- Variable, deren Wert eine Referenz (ein Zeiger) ist
- leere Referenz (Nullreferenz) wird mit Literal `null` beschrieben
- Variablen mit nicht-elementarem Datentyp sind Referenzvariablen

Reihungsvariablen (Array-Variablen)

- Reihungen (Arrays) Spezialfälle von Klassen
- Definition:

ArrayVariable :
Type `[ ]` *Identifier*

Variablen

Reihungsvariablen (intern)

- Erzeugung einer Reihung:

  Type`[] a = new` *Type*`[n]`

- Reihungsvariable *Type*`[] a;` ist Referenzvariable, die auf konkretes Reihungsobjekt verweist
- `new` *Type*`[n]` **erzeugt** (anonymes) Reihungsobjekt der Länge n
- Nach Zuweisung = ist (anonymes) Reihungobjekt der Wert von `a`
- Compiler stellt passende Anzahl von Variablennamen `a[0], a[1], ..., a[n-1]` zur Verfügung

Speicherbereiche nach Anweisung `int[] a=new int[3];`

Arithmetik

Zahlbereiche der Ganzzahltypen:

Typ	Bits	Minimalwert	Maximalwert
`byte`	8	-128	127
`short`	16	-32.768	32.767
`int`	32	-2.147.483.648	2.147.483.647
`long`	64	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
`char`	16	0 (`'\u0000'`)	65.535 (`'\uffff'`)

Arithmetik

Konversionsmethoden der Hüllenklassen:

Typ	Konversion aus `String`
`byte`	`Byte.parseByte(String a);`
`short`	`Short.parseShort(String a);`
`int`	`Integer.parseInt(String a);`
`long`	`Long.parseLong(String a);`
`float`	`Float.parseFloat(String a);`
`double`	`Double.parseDouble(String a);`

Arithmetik

```
public class CalcInt {
  public static void main(String[] args){
    char c=args[0].charAt(0);
    int x=Integer.parseInt(args[1]);
    int y=Integer.parseInt(args[2]);
    switch(c) {
      case '+': System.out.println(x+y); break;              // Addition
      case '-': System.out.println(x-y); break;           // Subtraktion
      case '*': System.out.println(x*y); break;        // Multiplikation
      case '/': System.out.println(x/y); break;
                                          // (ganzzahlige) Division
      case '%': System.out.println(x%y); break;
                                     // Divisionsrest: (x/y)*y+x%y == x
      default: System.out.println("Keine zulässige Operation");
    }
  }
}
```

Beachte: Java rundet immer zur Null (d.h. 3/2==1, -3/2==-1, 3%2==1 und -3%2==1)

Beachte: Programm nur korrekt auf Eingaben mit drei Eingabeparametern

Arithmetik

```
public class CalcFloat {
  public static void main(String[] args){
    char c=args[0].charAt(0);
    float x=Float.parseFloat(args[1]);
    float y=Float.parseFloat(args[2]);
    switch(c) {
      case '+': System.out.println(x+y); break;              // Addition
      case '-': System.out.println(x-y); break;           // Subtraktion
      case '*': System.out.println(x*y); break;        // Multiplikation
      case '/': System.out.println(x/y); break;              // Division
      case '%': System.out.println(x%y); break;
                                          // Divisionsrest: x − [x/y] · y
      default: System.out.println("Keine zulässige Operation");
    }
  }
}
```

Beachte: Programm nur korrekt auf Eingaben mit drei Eingabeparametern

Sonderfälle der Gleitkommadivision

`x`	`y`	`x/y`	`x%y`
endlicher Wert	± 0.0	$\pm\infty$	keine Zahl (`NaN`)
endlicher Wert	$\pm\infty$	± 0.0	`x`
± 0.0	± 0.0	keine Zahl (`NaN`)	keine Zahl (`NaN`)
$\pm\infty$	endlicher Wert	$\pm\infty$	keine Zahl (`NaN`)
$\pm\infty$	$\pm\infty$	keine Zahl (`NaN`)	keine Zahl (`NaN`)

Operatoren

Zuweisungsoperator (=):

- Zuweisungen haben immer einen Wert (in Java)
- Wert einer Zuweisung ist rechte Seite der Zuweisung
- Zuweisungen können als Teilausdrücke verwendet werden; eigentliche Zuweisung erfolgt als Nebeneffekt

Betrachten Zuweisung `x=y=1`:

- Zuweisung ist äquivalent zu `x=(y=1)`
- `x` wird Wert von `y=1` zugewiesen
- Wert von `y=1` ist `1`
- als Nebeneffekt wird `y` der Wert `1` zugewiesen

Operatoren

Zuweisungsoperator (=):

- Zuweisungen haben immer einen Wert (in Java)
- Wert einer Zuweisung ist rechte Seite der Zuweisung
- Zuweisungen können als Teilausdrücke verwendet werden; eigentliche Zuweisung erfolgt als Nebeneffekt

Betrachte Zuweisung `x=(y=1)+1`:

- `x` wird Wert von `(y=1)+1` zugewiesen
- Wert von `y=1` ist `1`
- Wert von `(y=1)+1` ist damit 2
- als Nebeneffekt wird `y` der Wert `1` zugewiesen

Operatoren

rekursive Zuweisungen:

Variable = *Variable* *Operator* *Expression*

Variable *Operator* = *Expression*

- kombinierter Zuweisungsoperator für jeden binären Operator möglich (+=, -=, *=, /=, %=)
- Wert beider Zuweisungen (zunächst) gleich

- `i=i+1;` ist äquivalent zu `i+=1;`
- `i=i*2;` ist äquivalent zu `i*=2;`
- `i=i+i;` ist äquivalent zu `i+=i;`

Operatoren

rekursive Zuweisungen:

Variable = *Variable* *Operator* *Expression*

Variable *Operator* = *Expression*

interne Aspekte:

- bei erstem Zuweisungstyp wird *Variable* zweimal ausgewertet
- bei zweitem Zuweisungstyp wird *Variable* nur einmal ausgewertet
- wird *Variable* durch Auswertung verändert, dann Unterschied möglich

- Zuweisung `a[i++]=a[i++]+(a[i++]=1)` ergibt:

i	0	1	2
a[i]	1	0	1

- Zuweisung `a[i++]+=(a[i++]=1)` ergibt:

i	0	1	2
a[i]	1	1	0

schlechter Programmierstil!

Operatoren

Boolesche Operatoren:

Operator	Interpretation	
`&`	logisches UND ($\wedge$)	AND-Funktion
`|`	logisches ODER ($\vee$)	OR-Funktion
`^`	logisches EXKLUSIVES ODER ($\oplus$)	XOR-Funktion
`!`	logische Negation ($\neg$)	NOT-Funktion
`&&`	sequenzielles logisches UND ($\wedge$)	AND-Funktion
`||`	sequenzielles logisches ODER ($\vee$)	OR-Funktion

- bei `(left=false) & (right=true)` werden beide Zuweisungen ausgeführt
- bei `(left=false) && (right=true)` wird nur erste Zuweisung ausgeführt (da Gesamtergebnis schon gegeben)

$\longrightarrow$ bequeme Auswertung (*lazy evaluation*)

Operatoren

Vergleichsoperatoren (Prädikate):

Operator	Interpretation
>	größer
>=	größer oder gleich
<	kleiner
<=	kleiner oder gleich
==	(genau) gleich
!=	ungleich

- `x%2 == 0` ergibt `true` $\Longleftrightarrow$ Wert der `int`-Variablen x ist gerade Zahl
- `((a>=b) || (a<=b)) && (a!=a)` ergibt `false`

Ausdrücke

atomare Ausdrücke (Induktionsanfang):

- Literal vom Typ T ist Ausdruck vom Typ T

- `17.0` ist atomarer Ausdruck vom Typ `float`

- Konstante vom Typ T ist Ausdruck vom Typ T

- `Double.NaN` (Konstante) ist atomarer Ausdruck vom Typ `double`

- Variable vom Typ T ist Ausdruck vom Typ T

- mit `Object obj` (Variable) ist `obj` atomarer Ausdruck vom Typ `Object`

Ausdrücke

zusammengesetzte Ausdrücke (Induktionsschritt):

- Für Funktion (Methode) $F : T_1 \times \cdots \times T_n \rightarrow T$, Ausdrücke $E_1, \ldots, E_n$ mit Typen $T_1, \ldots, T_n$ ist $F(E_1, \ldots, E_n)$ Ausdruck vom Typ T

- für `s.charAt` : `int` $\rightarrow$ `char` ist `s.charAt(0)` Ausdruck vom Typ `char`

- Für binären Operator $Op : T_1 \times T_2 \rightarrow T$, Ausdrücke E_1, E_2 mit Typen T_1, T_2 ist $(E_1 Op E_2)$ Ausdruck vom Typ T

- `(5+1.0)` ist Ausdruck vom Typ `float`

- Für unären Operator $Op : T \rightarrow T'$, Ausdruck E mit Typ T ist $Op(E)$ Ausdruck vom Typ T'

- `!(a>b)` ist Ausdruck vom Typ `boolean`

- Für Ausdruck E mit Typ T ist (E) Ausdruck vom Typ T

Ausdrücke

Präzedenzen (Bindungskraft)

Operatortyp	Beispiele
Postfix-Operatoren	`[ ]` `.` (*Params*) *Expr*`++` *Expr*`--`
Unäre Operatoren	`++`*Expr* `--`*Expr* `+`*Expr* `-`*Expr* `!`
Erzeugung und Anpassung	`new` (*Type*) *Expr*
multipikative Operatoren	`*` `/` `%`
additive Operatoren	`+` `-`
Ordnungsoperatoren	`<` `>` `<=` `>=`
Gleichheitsoperatoren	`==` `!=`
logisches UND	`&`
logisches ENTWEDER ODER	`^`
logisches ODER	`|`
sequenzielles logisches UND	`&&`
sequenzielles logisches ODER	`^^`
Bedingung	`? :`
Zuweisungsoperator	`=` `+=` `-=` `*=` `/=` `%=` `&=` `|=` `^=`

Ausdruck `a&&b==b|a` steht für Ausdruck `(a&&((b==b)|a))`

Anweisungen

Anweisungen:

Statements :
Statement$_{opt}$;
Statements Statement ;

Statement :
ExpressionStatement
DeclarationStatement
ControlFlowStatement

- *ExpressionStatement* für Berechnungen oder Auswertungen
- *DeclarationStatement* für Einrichtung und Initialisierung von Variablen
- *ControlFlowStatement* für Verzweigung und Iterationen im Programmfluss
- Beachte: leere Anweisung zugelassen (;)

Anweisungen

Kontrollflussanweisungen:

ControlFlowStatement :

- *BlockStatement*
- *IfThenStatement*
- *IfThenElseStatement*
- *SwitchStatement*
- *WhileStatement*
- *ForStatement*

Anweisungen

Blockanweisungen:

BlockStatement :
{ *Statements* }

- Blöcke können geschachtelt werden
- Variablendeklaration bleiben immer nur innerhalb eines Blockes gültig

```
{
   int k=1;
   {
      int i=2;
      k +=i;              // Zugriff auf k im inneren Block korrekt
   }
   k += i;          // Zugriff auf i im äußeren Block nicht erlaubt
}
```

Anweisungen

Verzweigung:

IfThenStatement :
`if (` *Expression* `)` *Statements*

IfThenElseStatement :
`if (` *Expression* `)` *Statements* `else` *Statements*

SwitchStatement :
`switch (` *Variable* `) {` *CaseStatements* $DefaultStatement_{opt}$ `}`

CaseStatements :
CaseStatement
CaseStatements *CaseStatement*

CaseStatement :
`case` *Constant* `: {` *Statements* $\texttt{break;}_{opt}$ `}`

DefaultStatement :
`default :` *BlockStatement*

Anweisungen

```
public class CalcInt {
  public static void main(String[] args){
    char c=args[0].charAt(0);
    int x=Integer.parseInt(args[1]);
    int y=Integer.parseInt(args[2]);
    switch(c) {
      case '+': { System.out.println(x+y); break; }
      case '-': { System.out.println(x-y); break; }
      case '*': { System.out.println(x*y); break; }
      case '/': { System.out.println(x/y); break; }
      case '%': { System.out.println(x%y); break; }
      default: { System.out.println("Keine zulässige Operation"); }
    }
  }
}
```

Anweisungen

Iterationen:

WhileStatement :
`while (` *Expression* `)` *Statements*

DoStatement :
`do` *Statements* `while (` *Expression* `)`

- *Expression* muss vom Typ `boolean` sein
- *WhileStatement* beschreibt anfangsprüfende Schleife
- *DoStatement* beschreibt endprüfende Schleife

Anweisungen

Iterationen:

ForStatement :

`for (` *DeclarationStatement*$_{opt}$ `;` *Expression*$_{opt}$ `;` *ExpressionStatement*$_{opt}$ `)`

Statements

Iteration mit *ForStatement*

```
for(int i=0;i<10;i++) { r=r+i; }
```

ist äquivalent zur Iteration mit `WhileStatement`

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

`for(;;);` beschreibt eine Endlosschleife