

# Syntax und Kontrollfluss

Prof. Dr. Christian Becker

Universität Stuttgart, Institut für Parallele und Verteilte Systeme

15. Mai 2025

Strings . . . . .	39
ASM Instructions vom Anfang nochmal aufgreifen . . . . .	46

# Unsere Herangehensweise

- ▶ Wir schauen uns **grundlegende Probleme der Informatik** und ihre Lösung mithilfe von **Algorithmen** an
- ▶ Dafür verwenden wir beispielhaft Java, um die Grundlagen des Programmierens zu verstehen und Algorithmen zu definieren
- ▶ Die grundlegende Syntax (zumindest auf der Ebene, auf der wir uns in dieser Vorlesung bewegen) der meisten höheren Programmiersprachen unterscheidet sich wenig



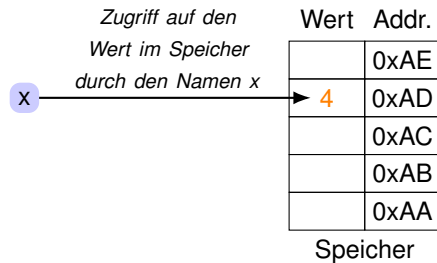
## Keywords / Schlüsselwörter

- ▶ Jede Programmiersprache hat eine Reihe an **Schlüsselwörtern** mit festgelegter Bedeutung
- ▶ Sind dem Compiler vorab bekannt
- ▶ Dürfen nicht zur Benennung von Variablen oder Funktionen verwendet werden



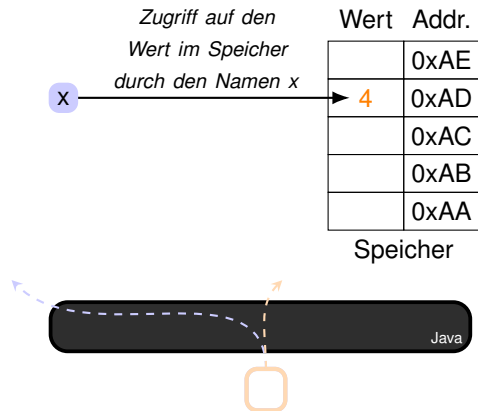
# Variablen

- ▶ Symbolischer Verweis auf eine Speicheradresse
- ▶ Verwendung eines Bezeichners anstelle der physischen Speicheradresse
- ▶ Ermöglicht Zugriff auf Speicheradressen durch Verwendung des Variablenbezeichners



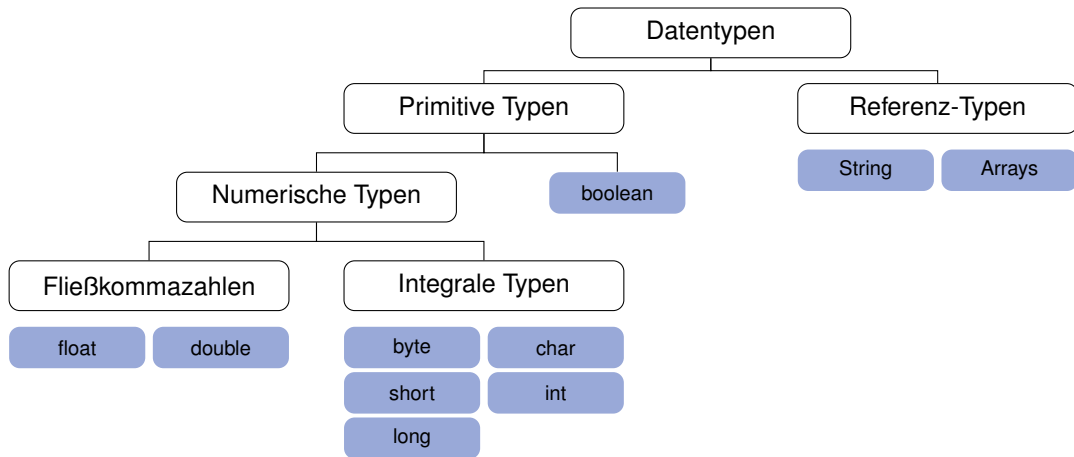
# Variablen

- ▶ Symbolischer Verweis auf eine Speicheradresse
- ▶ Verwendung eines Bezeichners anstelle der physischen Speicheradresse
- ▶ Ermöglicht Zugriff auf Speicheradressen durch Verwendung des Variablenbezeichners



# Datentypen

- ▶ Variablen haben einen Datentyp
- ▶ Der Datentyp bestimmt unter anderem, wie viel Speicherplatz für die Variable reserviert wird
- ▶ Abhängig vom Datentyp sind unterschiedliche Operationen möglich



# Datentypen

Datentyp	Beispiel	Größe	Standardwert	Wertebereich
boolean	boolean female = true;	1 bit	false	true/false
byte	byte age = 28;	1 byte	0	[−128; 127]
char	char letter = 'H';	2 byte	'\u0000'	Unicode-Zeichen
short	short amount = 63427;	2 byte	0	[−21 768; 32 767]
int	int credit = 12455432;	4 byte	0	[−2 147 483 648; 2 147 483 647]
long	long route = 5689537943;	8 byte	0L	[−2 <sup>63</sup> ; 3 <sup>63</sup> − 1]
float	float grade = 1.456f;	4 byte	0.0f	[±1.4x10 <sup>−45</sup> ; ±3.4x10 <sup>38</sup> ] <b>Genauigkeit:</b> 7 signifikante Stellen
double	double pi = 3.141592653;	8 byte	0.0	[±4.9x10 <sup>−324</sup> ; ±1.7x10 <sup>308</sup> ] <b>Genauigkeit:</b> 15 signifikante Stellen




# Deklaration von Variablen

- ▶ Variablen müssen vor ihrer ersten Verwendung **deklariert** werden
- ▶ Erstellt einen Bezeichner für eine neue Variable
- ▶ Der Datentyp einer Variable kann (in Java) später nicht mehr geändert werden

## Syntax: Variablendeklaration

```
<Eigenschaften> <Datentyp> <Bezeichner>;
```

Java

 **Achtung:** Java unterscheidet zwischen Groß- und Kleinschreibung!

- ▶ `int`  $\neq$  `Int`  $\neq$  `INT`
- ▶ `sum`  $\neq$  `Sum`  $\neq$  `SUM`

# Zuweisung

- ▶ Variablen können Werte zugewiesen werden
- ▶ Durch Zuweisung (=) erhält die **Zielvariable** den Wert des Arguments (rechte Seite der Gleichung)
- ▶ Eine Variable kann immer nur **einen** Wert speichern
- ▶ Zuweisung überschreibt den alten Wert der Variable

Java

⚠ Achtung: Zuweisung (=) ist **kein Vergleich**

# Initialisierung

- ▶ Deklaration und erste Zuweisung (Initialisierung) möglich in einem Schritt (empfohlen!)
- ▶ Ansonsten implizite Initialisierung (möglicherweise gefährlich)
- ▶ Manche Programmiersprachen machen keine implizite Initialisierung (z.B. C, C++)

# Konstanten

- ▶ Wir können Variablen als **unveränderlich** markieren
- ▶ Zum Beispiel einheitliche Definition von Werten: Gefrierpunkt von Wasser,  $\pi$ , Schriftgröße in einem Template, ...
- ▶ Schlüsselwort `final` markiert eine Variable als unveränderlich (konstant)

Java

- ▶ Eine Konstante kann **nur** initialisiert werden
- ▶ Keine weitere Zuweisung möglich

Java



# Kommentare

- ▶ Kommentare werden vom Computer ignoriert und dienen nur der Programmierin als Zusatzinformation
- ▶ Einzeilige Kommentare werden durch `//` eingeleitet

Java

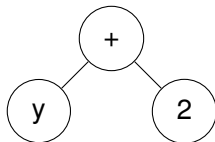
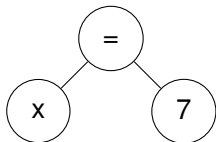
- ▶ Mehrzeilige Kommentare werden durch `/*` eingeleitet und mit `*/` beendet

Java

# Ausdrücke (Expressions)

- **Ausdrücke** bestehen aus Variablen oder Konstanten (**Operanden**) und **Operatoren**

Java



Operator

Operanden

# Operatoren

## Unvollständige Übersicht

Operator	Beispiel	Ergebnis
<b>Inkrement ++ / Dekrement --</b>	<code>int x = 3;</code>	
<code>x++ / x--</code>	<code>int y = ++x;</code>	<code>y = 4</code>
<code>++x / --x</code>	<code>int z = y++;</code>	<code>z = 4, y = 5</code>
<b>Arithmetische Operatoren:</b>	<code>int x = 2;</code>	
Addition +	<code>int y = 5;</code>	
Subtraktion -		
Multiplikation *	<code>int a = x + y;</code>	<code>a = 7</code>
Division /	<code>int b = x + y * a;</code>	<code>b = 37</code>
Modulo %	<code>double c = b / x;</code>	<code>c = 18.0</code>
	<code>int c = b / x;</code>	<code>c = 18</code>
	<code>double c = b / 2.0;</code>	<code>c = 18.5</code>
Abkürzung: <code>x = x + 1 → x += 1</code> (analog für alle arithmetischen Operatoren)		

# Operatoren

## Unvollständige Übersicht

Operator	Beispiel	Ergebnis
<b>Vergleichsoperatoren:</b> gleich == ungleich != kleiner < größer > kleiner/gleich <= größer/gleich >=	<pre>int j = 5; int j = 7;  i &lt; j; i &gt;= j;</pre>	   wahr falsch
<b>Boolesche (logische) Operatoren:</b> Konjunktion: AND (&&) Disjunktion: OR (  ) Negation: NOT (!) Exklusives OR (^)	<pre>boolean b1 = true; boolean b2 = false;  b1 &amp;&amp; b2; b1    !b2;</pre>	  falsch wahr
<b>Wertzuweisung</b>	<pre>int a, c;  a = 5; c = a;</pre>	  a = 5 c = 5



# Präzedenz von Operatoren (Precedence)

- ▶ Präzedenz ist eine Rangfolge
- ▶ Operatoren mit höherer Präzedenz werden zu erst ausgewertet
- ▶ Analog zu „Punkt von Strich“
- ▶ Präzedenz kann durch Verwendung von Klammern manipuliert werden

Unäre Operatoren		Arithmetisch		Vergleich		Bitweise	Logisch	Zuweisung
a++	++a	*	+	<	==	&	&&	=
a--	--a	/	-	>	!=			+ =
	!	%		<=		^		- =
	~			>=				* =
								/ =
								% =

höchste Präzedenz

niedrigste Präzedenz

# Arithmetische Operatoren

Addition, Subtraktion (+, -)

Java

⚠ Achtung: Gleitkommazahlen können zu unerwarteten Ergebnissen führen:

Java

**Output:** 0.19999999999999998

# Arithmetische Operatoren

Multiplikation, Division (\*, /)



Java

# Arithmetische Operatoren

## Modulo %

Modulo berechnet den **Rest** einer ganzzahligen Division

Berechnung ohne Modulo:

Java

Berechnung mit Modulo:

Java

# Widening Primitive Conversion

- ▶ Bei Operationen mit unterschiedlichen Operandentypen werden die Operanden in den Datentyp mit größerem Wertebereich konvertiert
- ▶ Diesen Vorgang bezeichnen wir als **widening conversion**
- ▶ Die Konvertierung passiert implizit

Rangordnung von primitiven Datentypen nach Größe Ihres Wertebereichs:

byte → short → int → long → float → double

## Vorsicht bei widening conversions!

⚠ Achtung: Größerer Wertebereich heißt nicht, dass die Konvertierung frei von Rundungsfehlern ist!

Java

**Output:** -47

# Casts

- ▶ Typ-Konvertierungen zwischen (kompatiblen) Typen sind auch explizit möglich
- ▶ Man nennt diese Konvertierungen **type casts**
- ⚠ Achtung: Möglicherweise gehen dabei Informationen verloren bzw. es wird gerundet

## Type casts

```
(<Datentyp>) <Bezeichner>
```

Konvertierung von Kommazahlen zu Ganzzahlen: Nachkommastellen werden abgeschnitten

Java

Konvertierung in kleinere Datentypen: Bits werden abgeschnitten

Java

# Boolesche Operatoren

- ▶ George Boole (1815-1864) war ein englischer Mathematiker (Autodidakt), Logiker und Philosoph
- ▶ Seine Hauptschrift „The Mathematical Analysis of Logic“ (1847) begründete die moderne mathematische Logik, die sich von der bis dahin üblichen philosophischen Logik durch eine konsequente Formalisierung abhebt
- ▶ **Boolesche Logik** hat als Ergebnis immer einen der Wahrheitswerte „**wahr**“ (true) oder „**falsch**“ (false)
- ▶ Zugrundeliegender Datentyp: `boolean`
  - ▶ Negation: NOT (!)
  - ▶ Konjunktion: AND (&&)
  - ▶ Disjunktion: OR (||)
  - ▶ Exklusives OR: XOR (^)



# Boolesche Ausdrücke

## Negation: NOT (!)

- ▶ Ergebnis einer Negation ist `true`, wenn der Operand `false` ist
- ▶ Ist der Operand `true`, so ist das Ergebnis `false`

<b>a</b>	<b>!a</b>
false	true
true	false

Java

# Boolesche Ausdrücke

## Konjunktion: AND (&&)

- ▶ Ergebnis einer Konjunktion ist `true`, wenn **beide** Operanden `true` sind
- ▶ Andernfalls ist das Ergebnis `false`

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

Java

# Boolesche Ausdrücke

## Disjunktion: OR (||)

- ▶ Ergebnis einer Konjunktion ist `true`, wenn **mindestens einer** der Operanden `true` ist
- ▶ Andernfalls ist das Ergebnis `false`

a	b	a    b
false	false	false
false	true	true
true	false	true
true	true	true

Java

# Boolesche Ausdrücke

## Verkürzte Auswertung (Short-Circuit-Evaluation)

- ▶ Bei den booleschen Operatoren OR (||) und AND (&&) wird der zweite Operand nur dann ausgewertet, wenn der erste Operand nicht ausreicht, um den Ausdruck eindeutig auszuwerten
- ▶ AND (&&) wertet den zweiten Operanden nur dann aus, wenn der erste Operand true ist
- ▶ OR (||) wertet den zweiten Operanden nur dann aus, wenn der erste Operand false ist

a	b	a && b
false	–	false
true	false	false
true	true	true

a	b	a    b
false	false	false
false	true	true
true	–	true

# Boolesche Ausdrücke

## Exklusives OR: XOR (^)

- ▶ Ergebnis eines exklusiven OR ist `true`, wenn **genau einer** der Operanden `true` ist
- ▶ Andernfalls ist das Ergebnis `false`

a	b	a ^ b
false	false	false
false	true	true
true	false	true
true	true	false

Java

# Vergleichsoperatoren

- ▶ Vergleichsoperatoren vergleichen Variablen
- ▶ Ergebnis eines Vergleichs ist ein boolescher Wert
- ▶ **Beispiele:** Sind zwei `ints` gleich? Ist ein `float` größer als ein anderer `float`?
- ▶ Alle primitiven Datentypen (`boolean`, `char`, `byte`, `short`, `int`, `float`, `double`) können mit diesen Operatoren verglichen werden

# Vergleichsoperatoren

## Gleichheitsvergleiche

- ▶ Vergleich auf Gleichheit (==) oder Ungleichheit (!=) von zwei Variablen

Java

# Vergleichsoperatoren

## Ordnungsvergleiche

- ▶ Relativer Vergleich von zwei Variablen
- ▶ Größer als (>), Größer oder gleich (>=)
- ▶ Kleiner als (<), Kleiner oder gleich (<=)



# Boolesche Ausdrücke

## Präzedenz (Wiederholung)

- ▶ Erinnern Sie sich an die Präzedenz von Operatoren?
- ▶ Wie wird `A || B && C` ausgewertet? Von links nach rechts? Punkt vor Strich??

# Boolesche Ausdrücke

## Präzedenz (Wiederholung)

- ▶ Erinnern Sie sich an die Präzedenz von Operatoren?
- ▶ Wie wird  $A \parallel B \&\& C$  ausgewertet? Von links nach rechts? Punkt vor Strich??

**Negation (!)**

**Vergleich (==)**

**Konjunktion AND  
(&&)**

**Disjunktion OR (||)**

höchste Präzedenz



niedrigste Präzedenz

### Beispiele:

- ▶  $!A \&\& B \hat{=} (!A) \&\& B$
- ▶  $A \parallel B \&\& C \hat{=} A \parallel (B \&\& C)$
- ▶  $A \&\& B \parallel !C \hat{=} (A \&\& B) \parallel (!C)$

**Bitte benutzen Sie Klammern!!**

# Funktionen

Eine Funktion besteht aus vier Teilen

1. Der Datentyp des Ergebnisses (**return type**)
2. Der Name der Funktion
3. Die Parameter oder Argumente der Funktion
4. Der Definitionsblock der Funktion (body)

## Funktionen

```
<Datentyp> <Bezeichner>(<Argumentliste>) {  
    <Anweisungen>  
}
```

return typ Funktionsname

Java

body parameter

# Funktionen

- ▶ Die **Signatur** einer Funktion ist der Returntyp, der Name und die Liste der Argumente:

```
// ...
```

Java

- ▶ Um einen Wert aus einer Funktion zurück zu geben, benutzen wir das Keyword `return`
- ▶ Eine Funktion, die keinen Wert zurück gibt, hat `void` („Leere“) als return type

# Scopes, Sichtbarkeit

Java

- ▶ Beispiel: Die Bezeichner `a, b, temporary` tauchen mehrfach auf
- ▶ Ist es immer die gleiche Variable? Nein!
- ▶ Variablen haben eine bestimmte Sichtbarkeit (visibility)
- ▶ Blöcke in Java sind curly braces `{ }` und die Anweisungen dazwischen
- ▶ Sichtbarkeit von Variablen ist oft beschränkt durch den aktuellen Block
- ▶ Man sagt auch, der aktuelle **Scope**

# Scopes

Java

► `a` ist sichtbar im inneren Scope

! Fehler: `someVariable` ist nicht sichtbar außerhalb des Scopes



- Variablen aller umgebenden Scopes sind sichtbar
- Eine Variable ist nicht außerhalb des aktuellen Scopes sichtbar!

# Strings

Strings



# String-Konkatenation

⚠ Achtung: Bei **Strings** wird der + Operator zur **Konkatenation** verwendet

```
String s1 = "Hallo";
```

Java

```
String s2 = "Welt";
```

Java

- ▶ Strings können auch mit anderen Datentypen konkateniert werden
- ▶ Dabei wird die andere Variable automatisch in einen String konvertiert

```
String s3 = "Hallo" + 123456789;
```

Java



# Arrays

- ▶ Arrays sind eine Menge von Elementen mit den gleichen Datentypen
- ▶ Arrays haben eine feste Länge
- ▶ Jedes Element hat eine Nummer (index)
- ▶ Wir können über den Index auf die Elemente zugreifen
- ▶ `T[]` ist der Typ eines Arrays in dem jedes Element vom Typ `T` ist

## Deklaration von Arrays

```
<Datentyp>[] <Bezeichner> = new <Datentyp>[<Expression>];
```

# Zugriff auf Elemente eines Arrays

- ▶ Jedes Element hat eine Nummer (index)
- ▶ Wir können über den Index auf die Elemente zugreifen



5	7			
0	1	2	3	4

- ▶ Die Array-Variable speichert die Länge des Arrays: `array.length`
- ▶ Der höchste Index ist immer `array.length - 1`

# Direct initialization von Arrays

- ▶ Ähnlich wie Variablen können wir Arrays auch Deklarieren und Initialisieren in einem Schritt

## Direct initialization

```
<Datentyp>[] <Bezeichner> = {<Value list>;
```

Java

# Was ist Kontrollfluss?

Grob gesagt, die Ausführungsreihenfolge der Anweisungen unseres Programms

# Kontrollfluss

## Anfangspunkt eines Programms

- ▶ Die Main-Methode ist der Anfangspunkt eines Programms in Java
- ▶ Die Java Virtual Machine (JVM) führt diese Methode beim Start des Programms aus
- ▶ Danach übernimmt die Programmiererin den Kontrollfluss
- ▶ Das Programm endet, wenn das Ende von `main` erreicht ist

# Kontrollfluss

## Ausführungsreihenfolge

- ▶ Anweisungen werden der Reihe nach ausgeführt.

Java

ASM Instructions vom Anfang nochmal aufgreifen

# Funktionsaufrufe

- ▶ Bei einem Funktionsaufruf springt der Kontrollfluss in die Funktion
- ▶ Wenn der Kontrollfluss `return` erreicht, springen wir zurück zum Funktionsaufruf
- ▶ Wenn die Funktion einen return type hat, der nicht `void` ist, können wir das Ergebnis der Funktion verwenden

# Kontrollfluss beeinflussen

Wir können den Kontrollfluss beeinflussen und ...

- ▶ manche Anweisungen nur unter bestimmten Umständen ausführen (if, else)
- ▶ manche Anweisungen wiederholt ausführen (for, while)



# Verzweigung

## If-Statement

- ▶ Das If-Statement (If-Anweisung) wertet einen booleschen Ausdruck aus
- ▶ Wenn das Ergebnis `true` ist, werden Anweisungen ausgeführt

### If-Statement

```
if (<Bedingung>) {  
    <Anweisungen>  
}
```

Java

# Verzweigung

## If-Else

- ▶ If kann um einen Else-Block erweitert werden
- ▶ Wenn das Ergebnis `false` ist, werden `else`-Anweisungen ausgeführt

### If-Statement

```
if (<Bedingung>) {  
    <Anweisungen>  
} else {  
    <Anweisungen>  
}
```

# Verzweigung

## If-Else

- ▶ Wichtig: Es wird immer genau eine der Alternativen ausgeführt!
- ▶ Niemals werden der If-Block UND der Else-Block ausgeführt!



# Verzweigung

## If-Else, Else-If

- ▶ Else und If können verschachtelt werden um mehrere Möglichkeiten abzufragen

Java

# Verzweigung

## Switch

- ▶ Else-If-Konstrukte können schnell unübersichtlich werden
- ▶ Switch-Case Statements sind eine Alternative für viele repetitive Vergleiche



## Vorsicht mit fall-through

- ▶ `break` am Ende von einem `case` verhindert *fall-through*
- ▶ Bei *fall-through* werden nachfolgende `case` auch ausgeführt anstatt ans Ende von `switch` zu springen!



Output:

5

6

default

## Vorsicht mit fall-through

- ▶ `break` am Ende von einem `case` verhindert *fall-through*
- ▶ Bei *fall-through* werden nachfolgende `case` auch ausgeführt anstatt ans Ende von `switch` zu springen!



Output:

5

6

# Wiederholungen (Looping statements)

## While-Loop

- ▶ Neben Verzweigungen kann man Wiederholungen in den Kontrollfluss einbauen
- ▶ Eine While-Schleife wiederholt Anweisungen so lange die Bedingung `true` ist
- ▶ Man nennt die einzelnen Wiederholungen auch *Iterationen*

### While-Loop

```
while (<Bedingung>) {  
    <Anweisungen>  
}
```

### Endlosschleife



# Wiederholungen (Looping statements)

## While-Loop

- ▶ Man kann (und sollte) die zu prüfende Bedingung in der While-Schleife beeinflussen

Java

# Wiederholungen (Looping statements)

## For-Loop

- ▶ Eine For-Schleife ist zum Beispiel für eine bekannte Anzahl Iterationen gut geeignet
- ▶ Die Schleife wird so lange wiederholt wie die Bedingung `true` ist
- ▶ Die Inkrement expression wird nach jeder Schleifeniteration ausgeführt

### For-Loop

```
for(<Initialisierung>; <Bedingung>; <Inkrement>) {  
    <Anweisungen>  
}
```

Java

# Wiederholungen (Looping statements)

## Range-based for

- ▶ Über bestimmte Listen, Mengen, etc kann man noch einfacher iterieren
- ▶ Dazu später mehr

### For-Loop

```
for(<Element> : <Array>) {  
    <Anweisungen>  
}
```

Java

Vergleichbar mit der mathematischen Schreibweise „für alle  $e \in A$ “.

# Wiederholungen abbrechen

break the cycle

- ▶ Mit `break` kann man eine For- oder While-Schleife jederzeit verlassen

Java

# Wiederholungen abkürzen

`continue`

- ▶ Mit `continue` springt der Kontrollfluss zurück zur Prüfung der Bedingung

Java