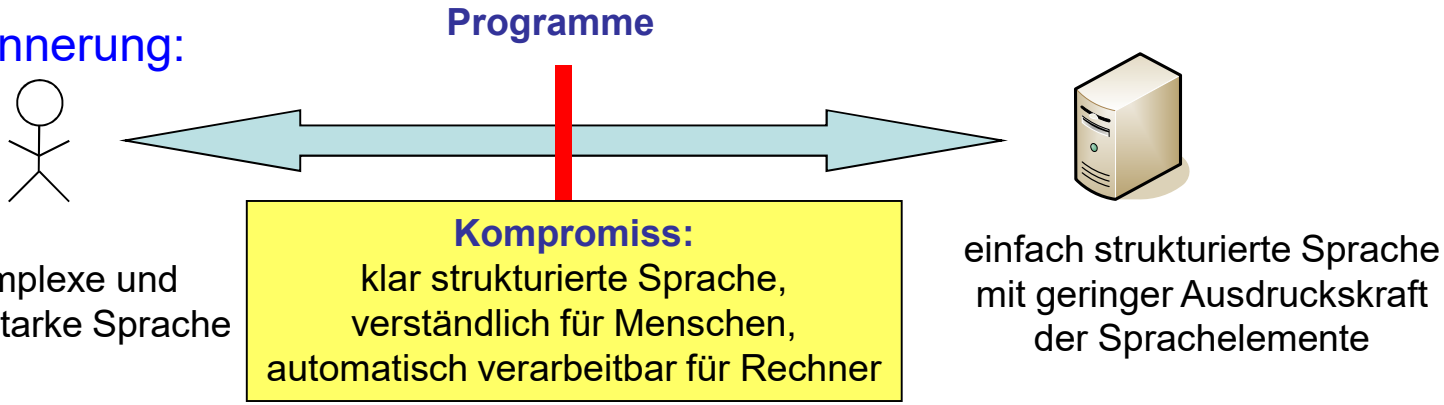
Ausführen von Programmen

- Zur Erinnerung:



- Frage: Wie kann ein Rechner nun Programme ausführen?
  - Antwort 1: Interpretierer
  - Antwort 2: Übersetzer / Compiler
  - Antwort 3: Mischung aus beiden

Beispielgrammatik

- Regeln einer Beispielgrammatik (kein Java):

```
<Programm>          ::= { <Anweisung> }
<Anweisung>         ::= <Leseanweisung> | <Schreibanweisung>
                        | <Zuweisung> | <Schleife>
<Leseanweisung>     ::= read <Bezeichner> ;
<Schreibanweisung>  ::= write <Bezeichner> ;
<Zuweisung>         ::= <Bezeichner> = <Ausdruck> ;
<Schleife>          ::= while <relAusdruck> do { <Anweisung> } endwhile
<Ausdruck>          ::= <Zahl> | <Bezeichner>
                        | <Ausdruck> + <Ausdruck> | <Ausdruck> * <Ausdruck>
<relAusdruck>       ::= <Ausdruck> < <Ausdruck>
<Bezeichner>        ::= …
<Zahl>              ::= …
```

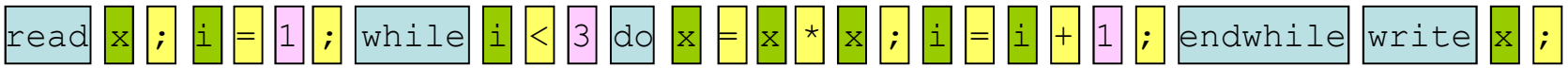
- Beispielprogramm:

```
read x;
i = 1;
while i < 3 do
    x = x * x;
    i = i + 1;
endwhile
write x;
```

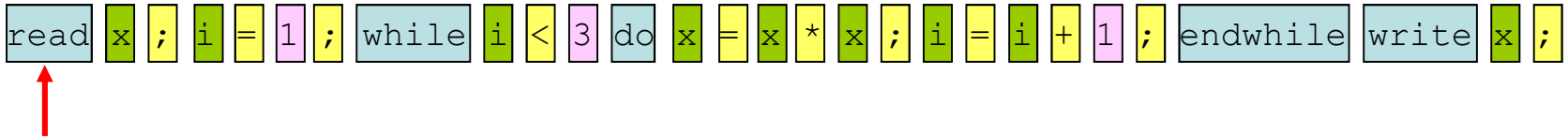
Interpretierer

- Aufgaben eines Interpretierers
  - Zerlegung des Programms in Grundsymbole (Token)
  - Analyse des Programms (Erkennen der Struktur) anhand von Grammatikregeln
  - Ausführung von Programmcode entsprechend erkannter Strukturelemente

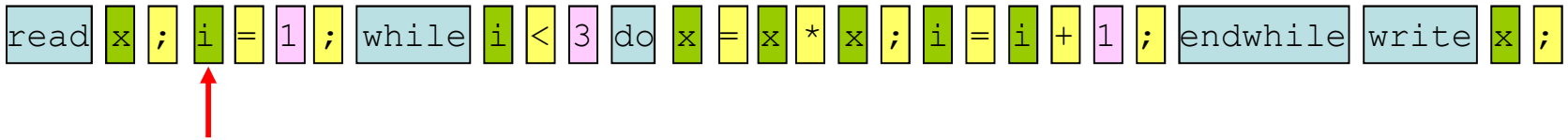
- Token sind relativ einfach aufgebaut (in der Praxis üblicherweise Typ-3-Grammatik) und können effizient erkannt werden
- Token werden dabei unterschieden in verschiedene Token-Klassen:
  - Konstanten / Literale (Beispiel: `1`, `31`)
  - Bezeichner (Beispiel: `x`, `i`, `x25`)
  - Schlüsselwörter (Beispiel: `read`, `while`)
  - Sonderzeichen (Beispiel: `=`, `<`, `*`)
- Nach Zerlegung:



Ablauf der Interpretierung des Beispiels

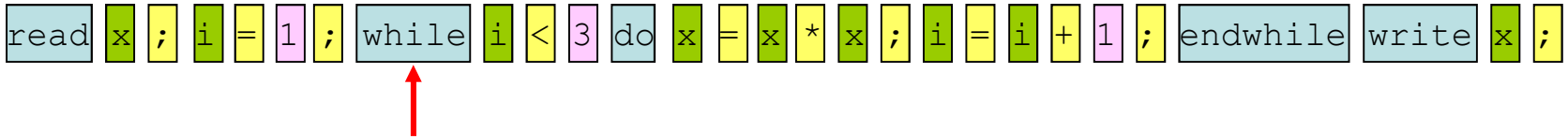


1. Abarbeitung der Token-Sequenz an der aktuellen Arbeitsposition
   - nach `read` muss ein Bezeichner kommen gefolgt von einem Semikolon
   - Bezeichner muss evtl. in einer Name-Wert-Tabelle neu angelegt werden
   - Einlesen eines Wertes von der Tastatur und Speichern dieses Wertes unter dem Namen in der Tabelle



2. Aufgrund des fehlenden Schlüsselworts muss eine Zuweisung vorliegen
   - Erkennen des Bezeichners (evtl. neu anlegen in Tabelle) und =
   - Abarbeitung des nachfolgenden Ausdrucks bis zum schließenden Semikolon
   - Berechnung des Ausdruckswertes und Speichern in Name-Wert-Tabelle

Ablauf der Interpretierung des Beispiels



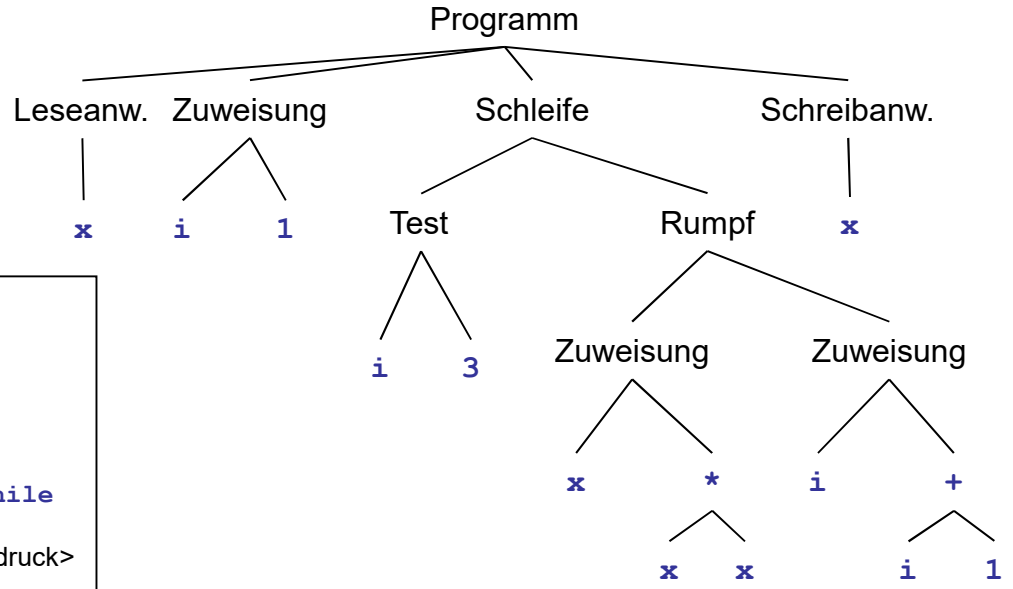
3. Schleife bearbeiten
   - Token-Position merken
   - Nachfolgenden Vergleichsausdruck (`i < 3`) bis zum `do` erkennen und Wert berechnen
   - Falls der Ergebniswert `false` ist, `endwhile` suchen und dahinter weitermachen
   - Falls der Ergebniswert `true` ist, den Schleifenrumpf abarbeiten bis zum `endwhile`. Anschließend an gespeicherter alter Token-Position wieder aufsetzen
   - (Frage: was passiert, wenn ein Schleifenrumpf eine weitere Schleife enthält)

- ...

Compiler

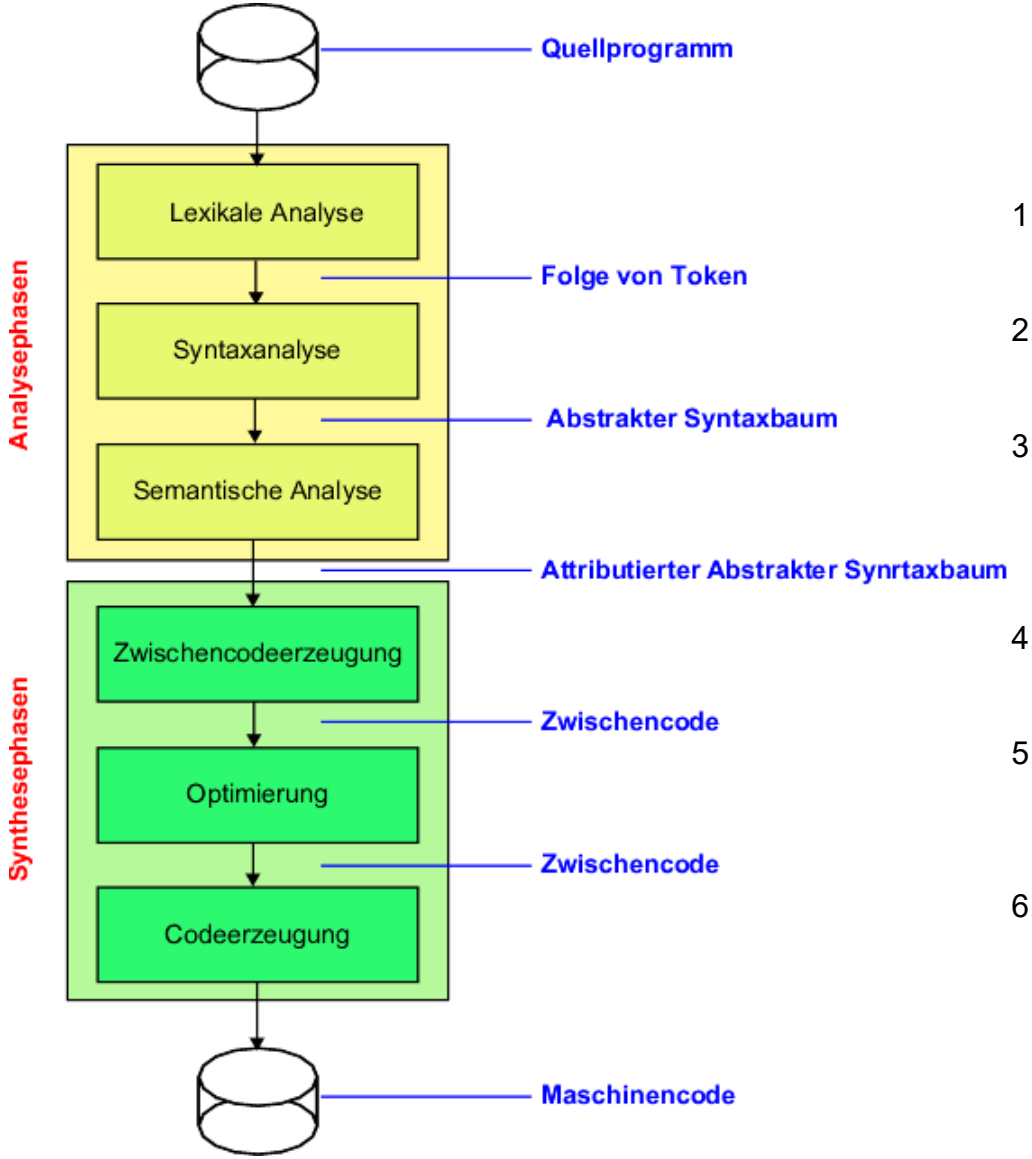
- Ein Compiler analysiert (genau einmal) das Quellprogramm (Source) und generiert daraus ein Objektprogramm für den Zielrechner
- Analysephase: Finden einer Ableitung vom Startsymbol der Grammatik zum vorliegenden Programm (mehrere Strategien dazu möglich; sehr komplex)
- Kennt man aufgrund der Analyse die Struktur des Programms, so kann man daraus entsprechend den Strukturelementen Code generieren (Synthese)
- Beispiel zur Programmstruktur:

```
read x;
i = 1;
while i < 3 do
    x = x * x;
    i = i + 1;
endwhile
write x;
```




```
<Programm>          ::= { <Anweisung> }
<Anweisung>         ::= <Leseanweisung> | <Schreibanweisung>
                        | <Zuweisung> | <Schleife>
<Leseanweisung>     ::= read <Bezeichner> ;
<Schreibanweisung>::= write <Bezeichner> ;
<Zuweisung>         ::= <Bezeichner> = <Ausdruck> ;
<Schleife>          ::= while <relAusdruck> do { <Anweisung> } endwhile
<Ausdruck>          ::= <Zahl> | <Bezeichner>
                        | <Ausdruck> + <Ausdruck> | <Ausdruck> * <Ausdruck>
<relAusdruck>       ::= <Ausdruck> < <Ausdruck>
...
```

Phasen eines Compilers



1. Zerlegung der Eingabedaten in Folge von Token (Grundsymbole der Sprache wie Bezeichner, Konstanten, Sonderzeichen, Trennzeichen
2. Erkennung der Programmstruktur durch Ableitung des Startsymbols der Grammatik zur Tokenfolge
3. Semantische Überprüfung, beispielsweise, dass eine Variable vor ihrer Nutzung deklariert sein muss
4. Aus der Struktur des Programms wird ein Code erzeugt, der ähnlich einer Maschinensprache ist
5. Optimierung des Zwischencodes hinsichtlich Zeit und/oder Speicherbedarf
6. Generierung des Codes für einen Zielrechner

Gegenüberstellung Interpretierer / Compiler

- Interpreter
  - analysieren Schritt für Schritt nur den Programmteil, der als nächstes zur Abarbeitung ansteht
  - Analyse ist schneller als in einem Compiler, wiederholt sich aber mit jeder Interpretierung dieses Programmstücks
  - gut geeignet für Programme, die selten / einmalig laufen
  - schlecht geeignet für Programme, die wiederholt / effizient ablaufen sollen
  - Einsatzgebiete: Kontrollsprachen (Shell), interaktive Umgebungen

- Compiler
  - analysieren und übersetzen das komplette (Teil-) Programm
  - umfassende Fehleranalyse und Programmoptimierung möglich
  - Übersetzungsvorgang ist nur ein mal nötig für beliebig viele Programmausführungen
  - primär geeignet für oft laufende Programme oder wo es auf Effizienz ankommt
  - Einsatzgebiete: größere Programme, die oft ausführt werden (das sind die meisten Programme)

Virtuelle Machinen

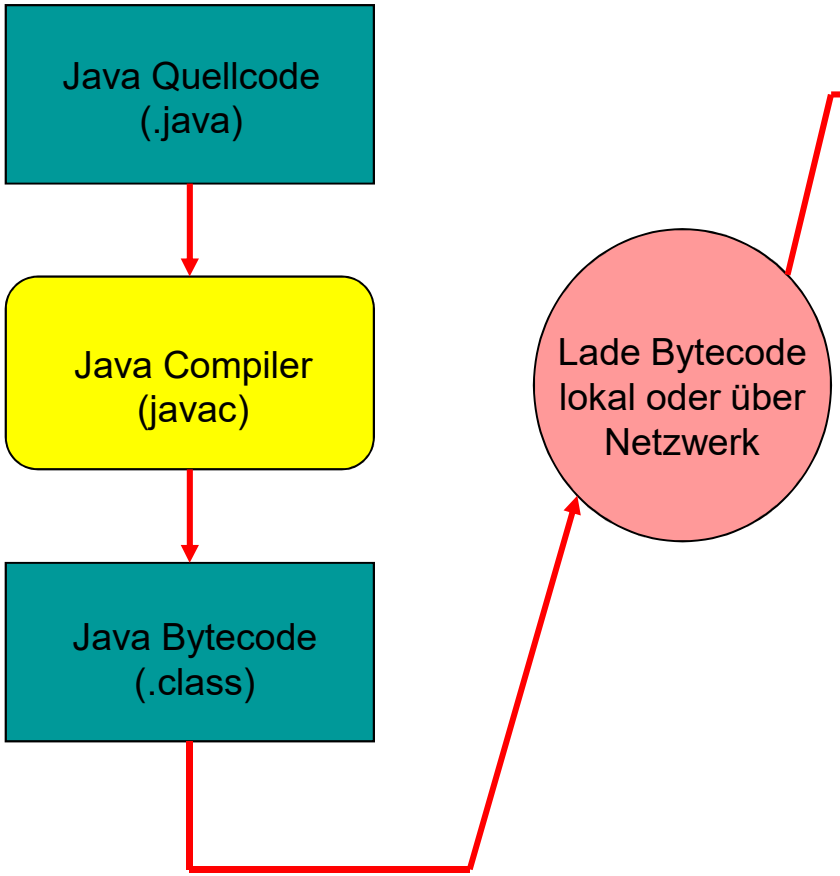
- Generierter Code eines Compilers ist maschinenspezifisch
- Problem: ich möchte ein Programm im Netz zum Download anbieten, das auf allen Rechnern der Welt laufen soll ($\rightarrow$ geht nicht)

- Lösung: Übersetzung für einen hypothetischen Prozessor
- Beim Start des Programms wird der Code für diesen hypothetischen Prozessor dann auf dem Zielrechner
  - interpretiert (wenig Aufwand, langsame Ausführung)
  - übersetzt (größerer Aufwand, aber schnelle Ausführung)
- Dazu benötigt man allerdings Laufzeitunterstützung auf jedem Rechner, auf dem solch ein Programm gestartet werden soll

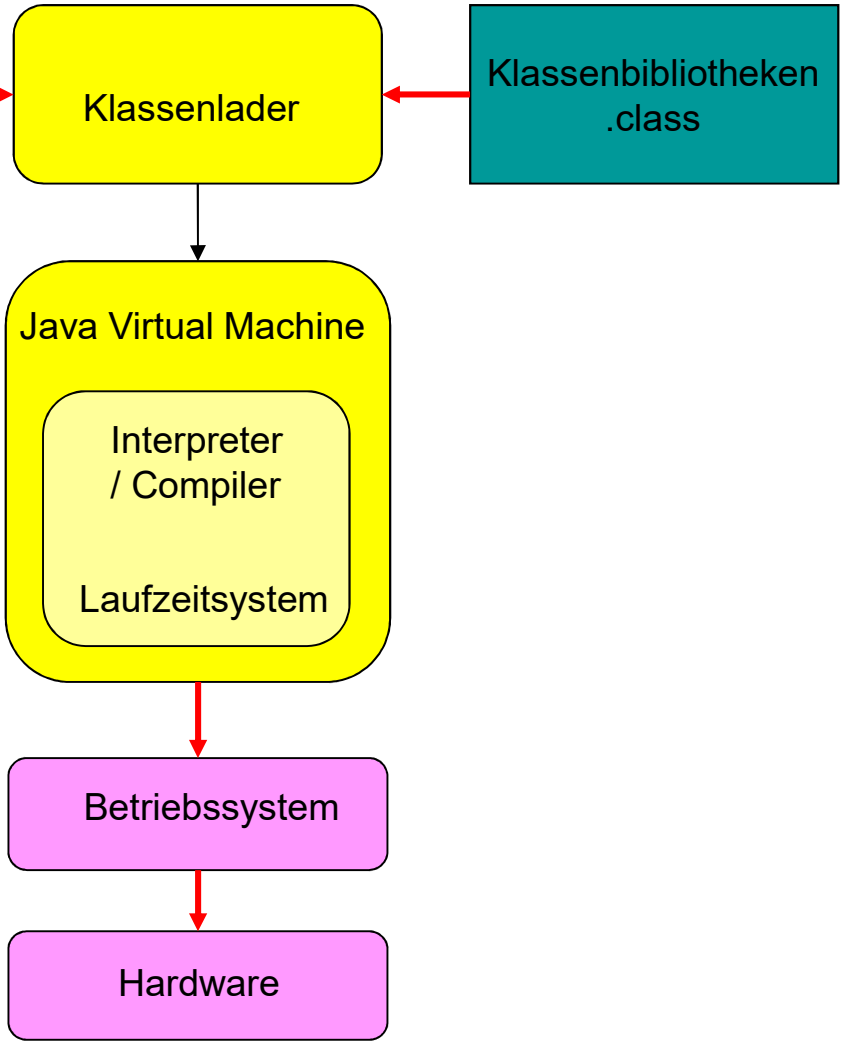
- Java hat dazu die JVM (Java Virtual Machine) als Teil der JRE (Java Runtime Environment)
- Microsoft hat analog dazu die .NET Umgebung
  - Common Intermediate Language (CIL): Prozessorsprache
  - Common Language Runtime (CLR): Laufzeitunterstützung

Java Umgebung

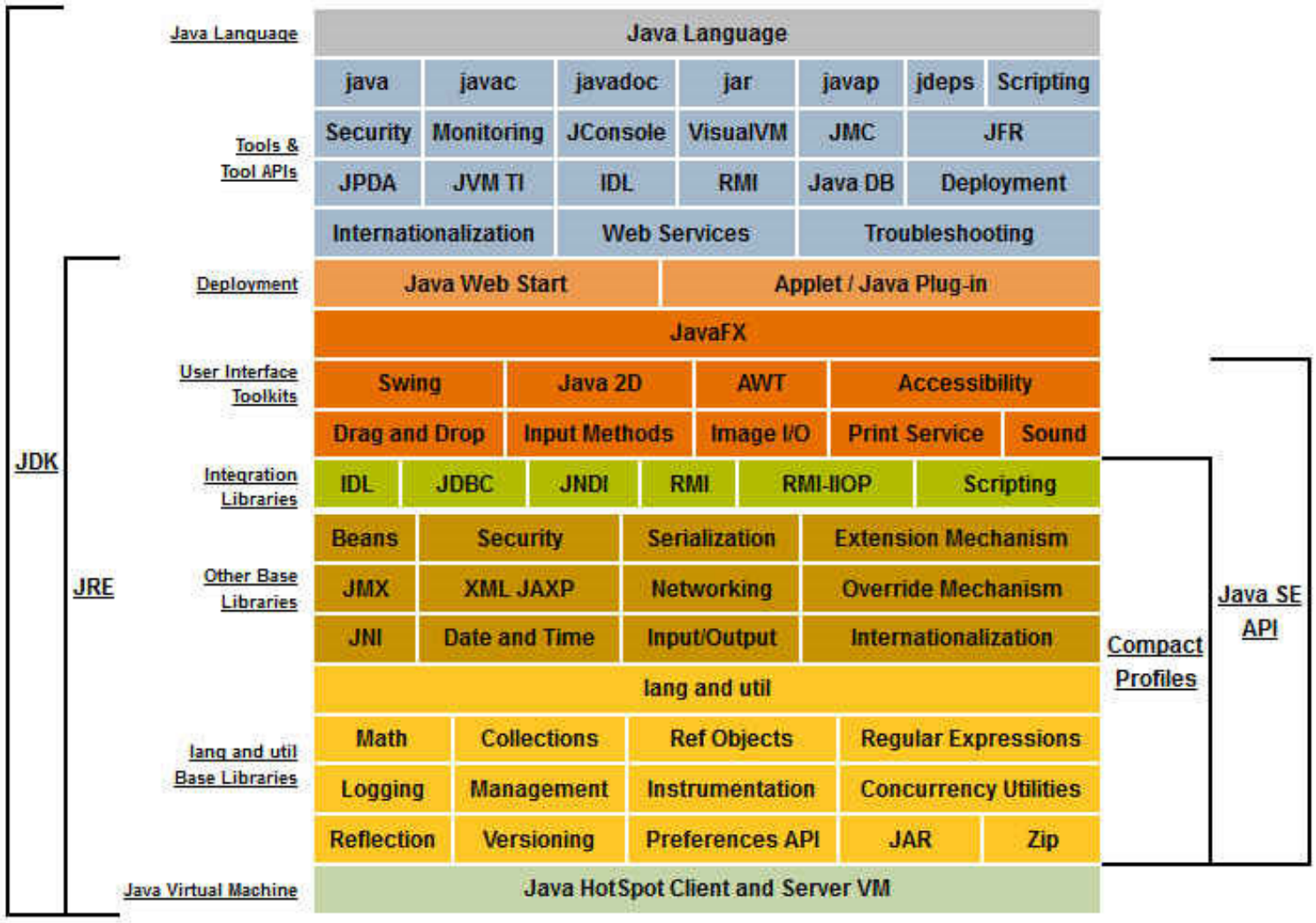
Entwicklungsumgebung



Laufzeitumgebung (JRE)



Java Software Development Kit (SDK 8)



Quelle: `http://docs.oracle.com/javase/8/docs/`

Zwischenstand

- Interpreter gehen schrittweise durch einen Programm durch und schauen jeweils, was an der aktuellen Betrachtungsstelle zu tun ist.
- Compiler übersetzen ein Programm in den Binärcode des Zielrechners, wo dieser beliebig oft ohne weiteren Aufwand ausgeführt werden kann.
- Compilern haben mit einer virtuellen Maschine nur eine Zielarchitektur. Auf einem realem Rechner muss dann aber der Code dieses virtuellen Rechners auf den vorliegenden realen Rechner abgebildet werden .
- Für die Sprache Java ist die Java Virtual Machine definiert.

Reflektion

- Erläutern Sie ihrem Nachbarn die Vor- und Nachteile einer virtuellen Maschine.
- Könnte man eine virtuelle Maschine wie die JVM auch als realen Rechner bauen?
- Könnte man einen realen Rechner auch als virtuelle Maschine angeben?

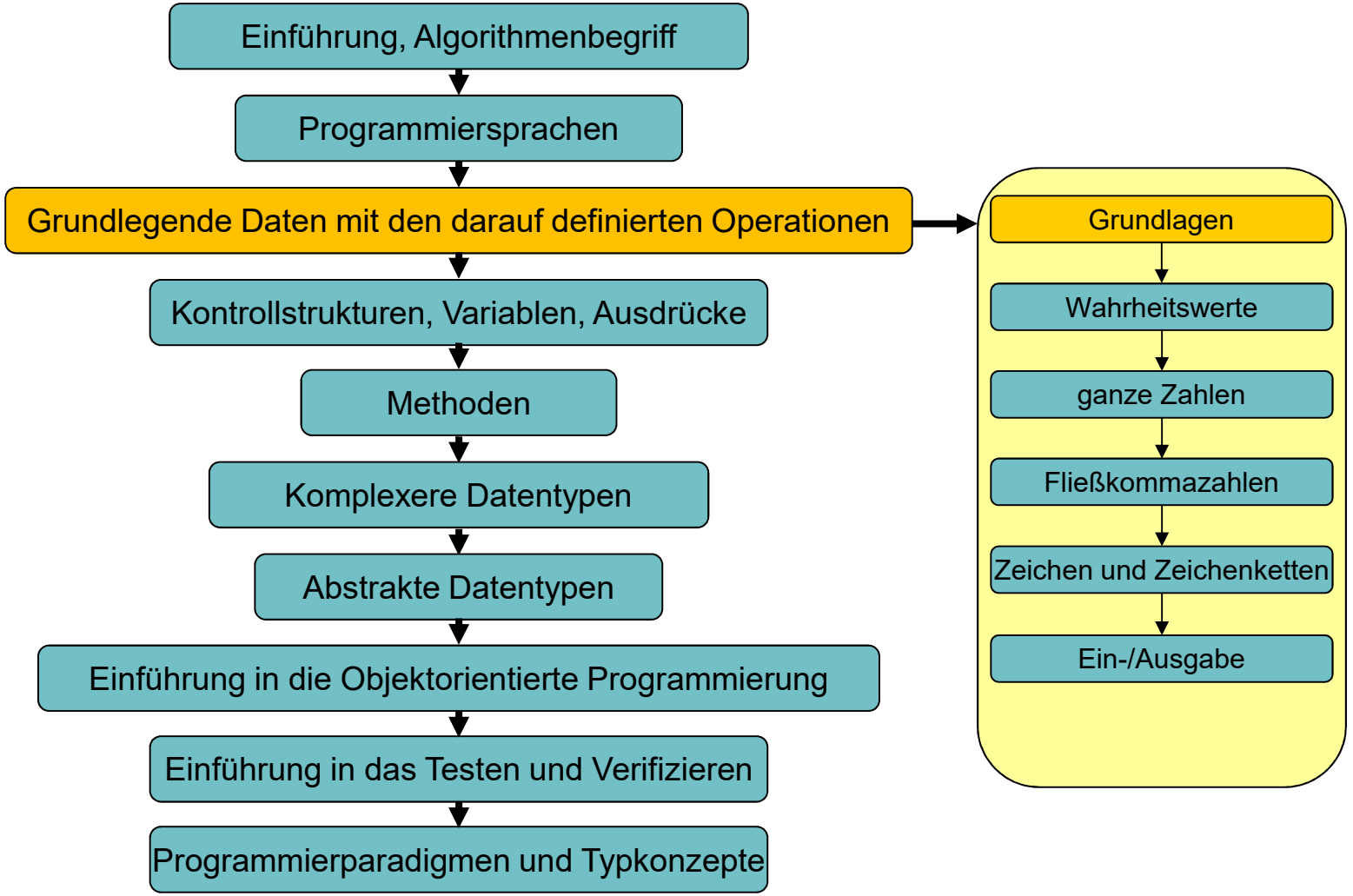
Zusammenfassung

- Programmiersprachen als Kompromiss zwischen Bedürfnissen von Menschen und Möglichkeiten von Prozessoren
- Verschiedene Klassen von Programmiersprachen

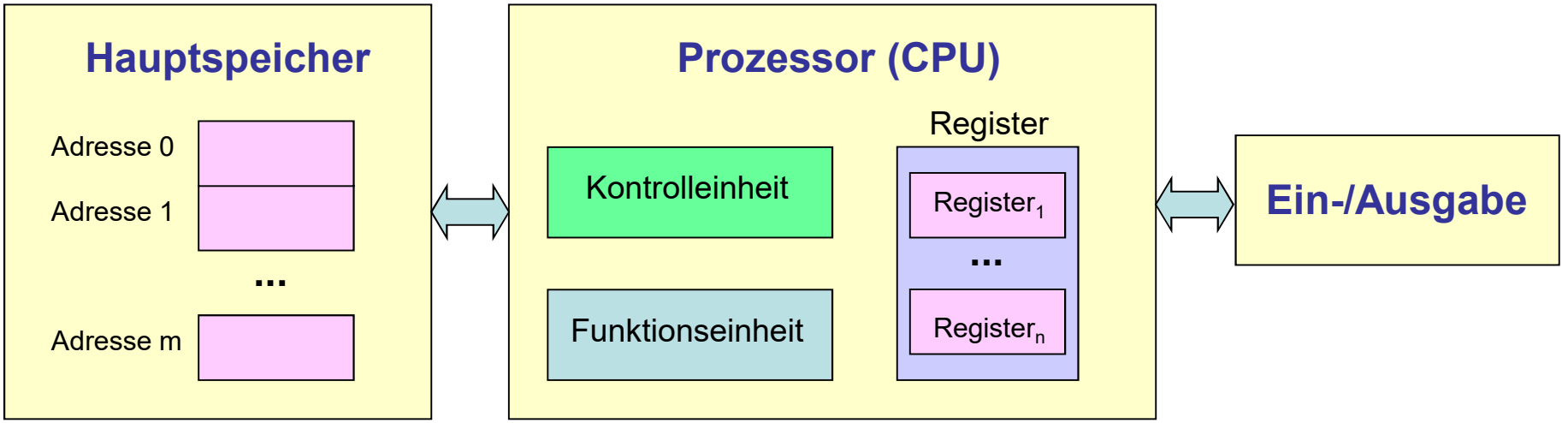
- Programmiersprachen werden definiert durch Grammatiken. Die Produktionen der Grammatik definieren die Struktur der Sprache.
- Angabe von Grammatikregeln durch direkte Grammatikregeln (Theoretiker), BNF / EBNF (maschinenverarbeitbar) und Syntaxdiagramme (Menschen)

- Compiler zur Übersetzung eines Programms für einen Rechner nötig. Der erzeugte Maschinencode ist nur auf dieser Prozessorfamilie / diesem Betriebssystem lauffähig
- Statt Code für einen speziellen Prozessor zu erzeugen, kann Code für eine virtuelle Maschine erzeugt werden und dann auf einem beliebigem Rechner bei Start des Programms interpretiert / übersetzt werden. Dazu ist eine Laufzeitunterstützung nötig (Beispiele: JVM/JRE, .NET)

Inhalt dieser Veranstaltung



Architektur von Rechnern



- Kontrolleinheit: Ablaufsteuerung des Programms (welcher Befehl wird als nächstes ausgeführt)
- Funktionseinheit: einfache Berechnungen ausführen auf Registerinhalten (addieren,...)
- Register: ein Register kann ein einzelnes Datum aufnehmen (1-8 Bytes), begrenzte Anzahl
- Hauptspeicher: jedes Kästchen hat eine eindeutige Adresse und kann genau ein Byte aufnehmen

Bits

- Rechner kennen als atomare Einheit Bits (Strom an/aus, magnetisiert / nicht magnetisiert, Licht an/aus,...)
- Mit einem Bit lassen sich zwei verschiedene Zustände darstellen.
- Welche Werte diese Zustände bedeuten sollen, hängt von der Interpretation der beiden Bitzustände ab!

- Beispiele:

	Bit=0	Bit=1
Zahlendarstellung 0/1	0	1
Zahlendarstellung 500/510	500	510
wahr / falsch	falsch	wahr
Frank zuhause oder nicht	nicht zuhause	zuhause
Frank zuhause oder nicht	zuhause	nicht zuhause

Bytes

- Mit n Bits lassen sich 2^n verschiedene Zustände / Werte darstellen
- Beispiel: n=3

Bit 2	Bit 1	Bit 0	Zustandsnummer	später nutzen wir
0	0	0	0	$0 = 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
0	0	1	1	$1 = 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
0	1	0	2	$2 = 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
0	1	1	3	$3 = 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$
1	0	0	4	$4 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
1	0	1	5	$5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
1	1	0	6	$6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
1	1	1	7	$7 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

- Eine Folge von 8 Bits nennt man ein Byte
- Mit einem Byte lassen sich also 2^8=256 verschiedene Werte darstellen
- Bytes sind in Rechnern die kleinste Einheit, die gespeichert / verwaltet wird
- Jedes Byte im Hauptspeicher eines Rechners hat eine eindeutige Adresse (eine Nummer)

Wörter

- Die Datenmenge, die ein Register in einem Prozessor aufnehmen kann, nennt man ein (Daten-)Wort
- Die Register in einem Prozessor können heute je nach Prozessortyp 8-64 Bits / 1-8 Byte aufnehmen (Waschmaschinenprozessoren weniger, "normale" PC-Prozessoren 64 Bits; Spezialfälle hier nicht betrachtet)

- Operationen in Prozessoren finden immer auf Registerinhalten statt
- Deshalb müssen alle Operationen auf Daten (auch MP3, Video,...) letztlich auf Operationen auf höchstens die Wortlänge herunter gebrochen werden
- Diese Aufgabe erledigt der Compiler

- Aber: die atomaren Daten, mit denen ein Programmierer in einer Programmiersprache arbeiten kann (wir sehen gleich: primitive Datentypen), sind von diesen Restriktionen entscheidend beeinflusst

Darstellung von Informationen

- Wenn man Informationen durch ein oder mehrere Bits darstellen möchte, muss eine Kodierung von Informationswerten durch Bits definiert werden
- Bekannte Kodierungen (unabhängig von Bits): Dezimalzahlen, Morse-Code, Flaggencode, Braille-Schrift

- In Programmen können sehr unterschiedliche Informationen anfallen (Matrikelnummer, Temperaturwerte, MP3-Musik, H264-Video, ...)

- Fragen:
  1) Wie kann ich solche Informationen in einem Rechner darstellen?
  2) Wie kann ich solche Informationen in einem Programm angeben / manipulieren?

- Antworten:
  1) Darstellung in einem Rechner letztendlich nur mit Bits/Bytes
  2) Für einfache Wertemengen (Zahlen, Buchstaben) gibt es vordefinierte Datentypen in Programmiersprachen. Komplexe Werte müssen aufbauend auf einfachen Werten zusammengesetzt werden

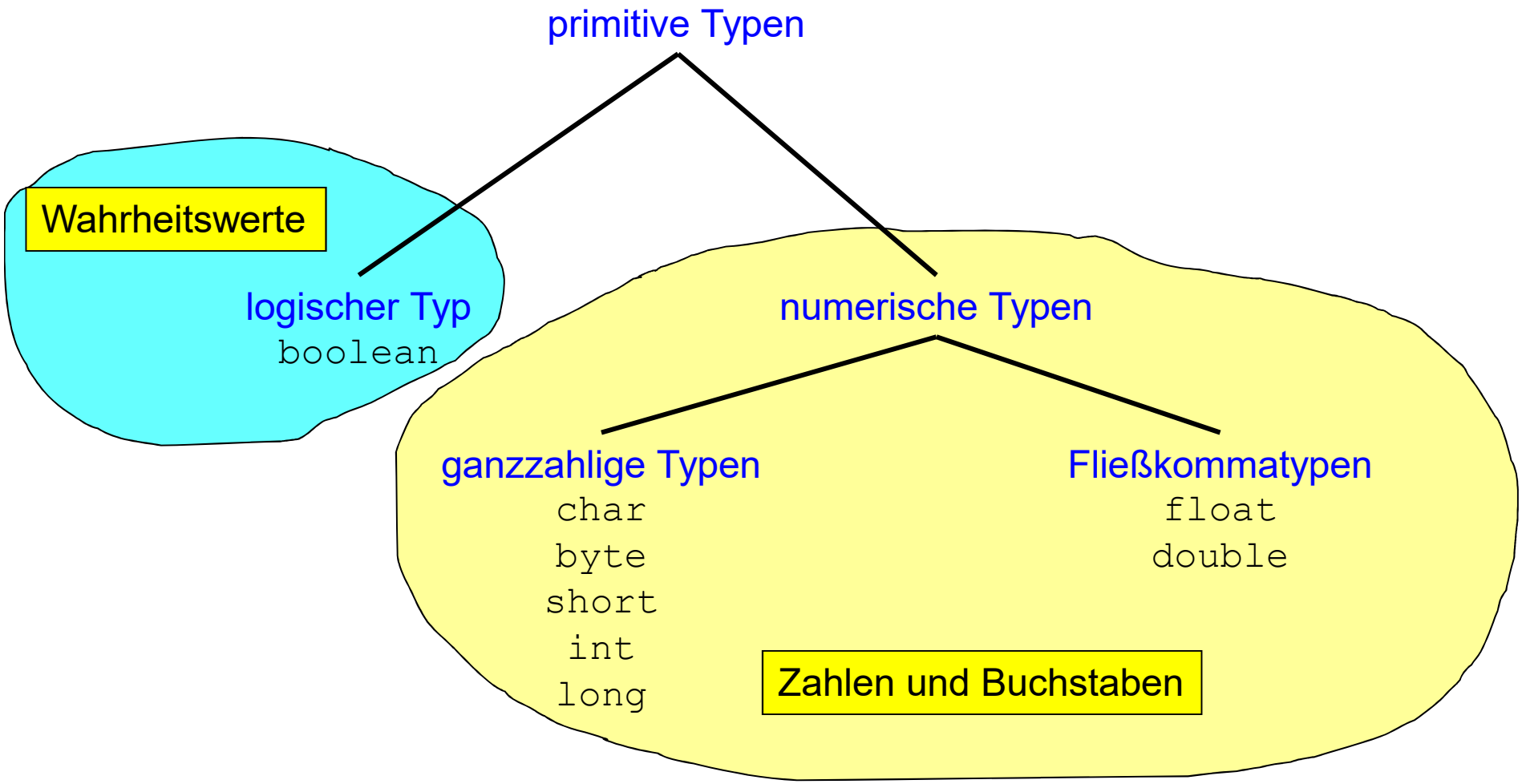
Datentyp

- (Konkreter) Datentyp: eine Wertemenge mit darauf definierten Operationen und einer Kodierung der möglichen Werte

- Bekannte Beispiele (Achtung: in dieses Beispiele kommen unendliche Mengen vor!):
  - natürliche Zahlen mit den Operationen +,-,·,/ und Dezimaldarstellung
  - natürliche Zahlen mit den Operationen +,-,·,/ und der Bierdeckeldarstellung (Zahl = Anzahl Striche)
  - Brüche x/y mit +,-, ·,/ und Darstellung durch zwei ganze Zahlen (Zähler, Nenner) in Dezimaldarstellung

- Weiteres Beispiel (etwas exotischer):
  - Wahrheitswerte {wahr, falsch} mit den Operationen Und, Oder und der Kodierung `ä` für wahr und `#` für falsch

Übersicht über die primitiven Datentypen in Java



Zwischenstand

- Fast alle Rechner haben einen prinzipiell sehr ähnlichen Aufbau.
- Innerhalb eines Rechners wird ausschließlich mit Bits, Bytes und Vielfachen von Bytes gearbeitet. Alle Daten müssen also rechnerintern damit dargestellt werden.
- Jedes Byte im Hauptspeicher hat eine eindeutige Adresse (eine natürliche Zahl).
- Mit n Bits lassen sich 2^n verschiedene Werte darstellen.
- Die Abbildung von Daten auf Bits oder Bit-Folgen stellt eine Codierung dar.

Reflektion

- Wenn man statt Bits mit 2 Zuständen „Trips“ mit 3 möglichen Zuständen hätte: Wieviele verschiedene Werte lassen sich mit n Trips darstellen?
- Überlegen Sie sich aus ihrem privaten Umfeld, welche Informationen sich nicht direkt / als Ganzes mit den Kategorien der primitiven Typen in Java darstellen lassen.

Speicherung von Informationen

- Variablen in Programmiersprachen dienen dazu, Werte während der Laufzeit eines Programms zu speichern, um später diese Werte im Programm wieder zu benutzen

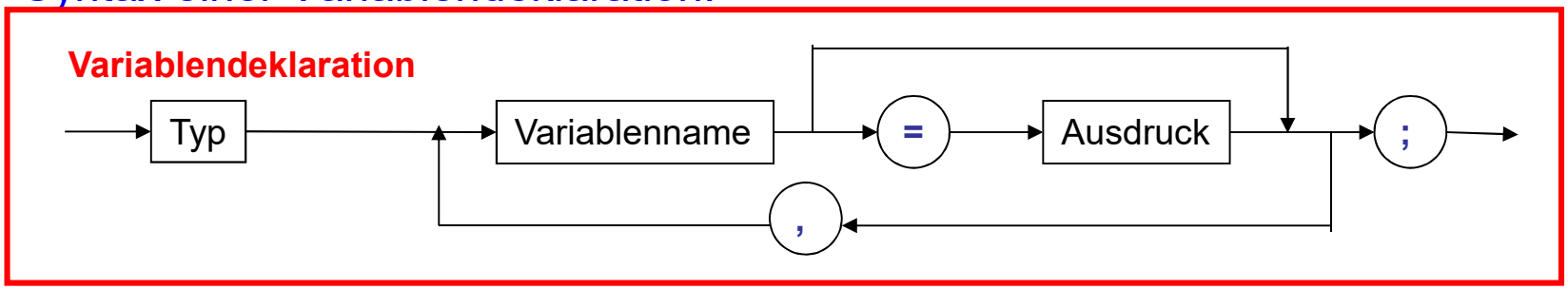
- Variablen haben einen Namen, über den man die Variable anspricht
- Variablennamen sind frei wählbar (Buchstabe gefolgt von Buchstaben/Ziffern; Klein-/Großschreibung relevant)

- Variablen sind immer von einem bestimmten Datentyp und können nur Werte dieses Typs aufnehmen

- Im Gegensatz zum Variablenbegriff der Mathematik kann man in Programmiersprachen den Wert einer Variablen ändern (gleich mehr)

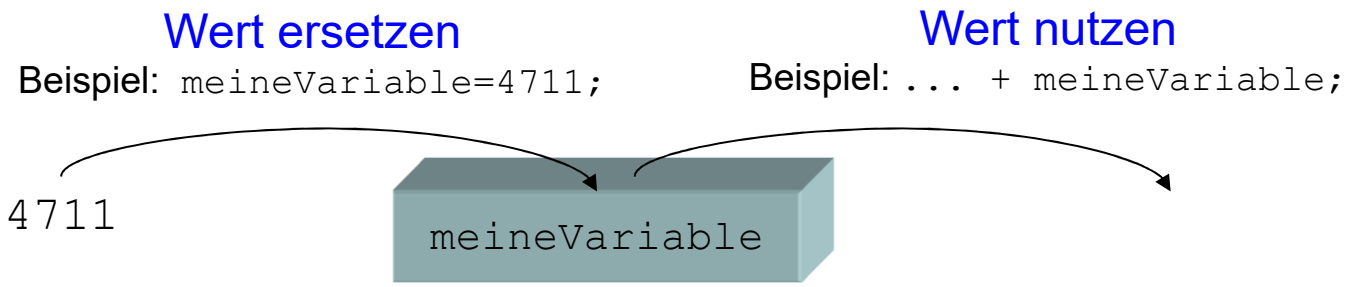
Deklaration von Variablen

- Jede Variable muss in Java deklariert werden. Bei einer Deklaration legt man fest:
  - den Namen der Variablen
  - den Typ der Variablen
  - optional einen Anfangswert der Variablen
  - Beispiel: `int meineVariable = 5;`
- Syntax einer Variablendeklaration:



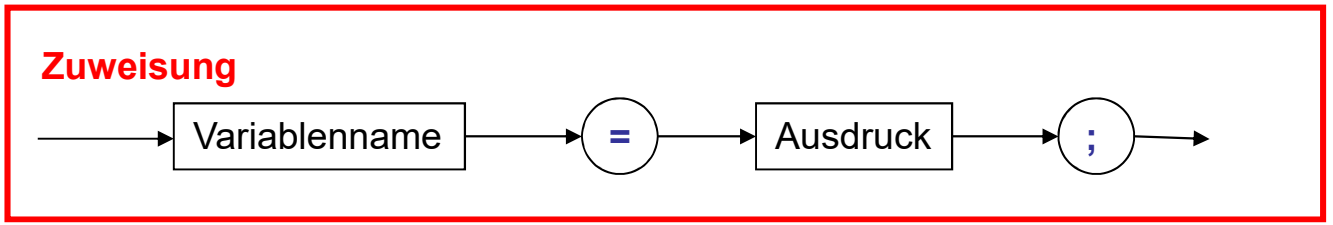
- Mit einer Variablen ist immer Hauptspeicherplatz verbunden, der bei der Deklaration automatisch vom Compiler reserviert wird und der immer den aktuellen Wert der Variablen enthält
- Aufgrund des Datentyps ist genau definiert, wie die gespeicherten Bits/Bytes des Variablenwertes interpretiert werden müssen

Operationen auf Variablen



- Auf Variablen gibt es zwei Operationen:

1. Nutzung des Namens auf der linken Seite einer Zuweisung: Ersetze den Wert dieser Variablen durch einen neuen Wert (auf rechter Seite)

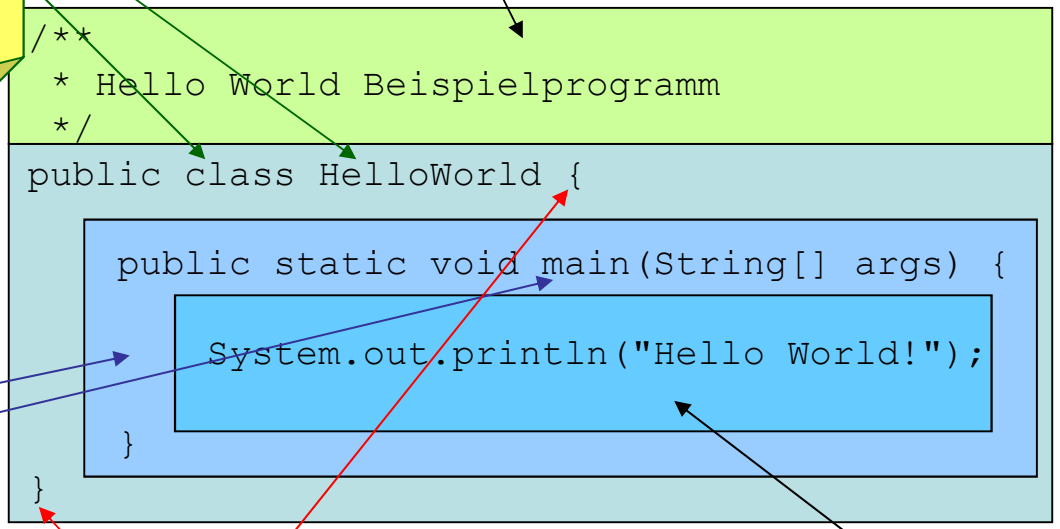


2. Nutzung des Namens in einem Ausdruck: Gib mir den aktuellen Wert dieser Variablen (der Wert in der Variablen bleibt erhalten)

Java Rahmenprogramm

Ein Programm ist in eine Klasse eingebettet.
Vorerst arbeiten wir immer nur mit einer Klasse (= ein Programm).
Der Name der Datei, in der diese Klasse gespeichert wird, muss mit dem Namen der Klasse (hinter `public class`) übereinstimmen.
Klassennamen beginnen vereinbarungsgemäß mit einem Großbuchstaben.

optionaler Kommentar
Kommentare ändern die Programmbedeutung nicht und können (fast) überall stehen



In einer Klasse kann es beliebig viele Methoden geben. Die Methode mit dem Namen main hat eine besondere Bedeutung: hier wird mit der Programmausführung begonnen

Geschweifte Klammern treten immer in Paaren auf und dienen der Markierung von Beginn/Ende eines Konstrukts.

Innerhalb einer Methode stehen Anweisungen (und Deklarationen).

Kommentare

- Kommentare dienen der Dokumentation und Erläuterung für den Leser
- Kommentare dürfen (fast) überall in einem Programm stehen
- Innerhalb eines Kommentars kann beliebiger Text stehen, der vom Compiler ignoriert wird

- Java kennt drei Kommentarformen
- 2 Formen von Implementierungskommentaren

```
/* hier kann beliebiger Text stehen,
   der auch über mehrere Zeilen gehen kann */
// Hier kann beliebiger Text stehen, der am Zeilenende aufhört
```

- Dokumentationskommentare

```
/**
 In solchen Kommentaren steht Text in einer bestimmten Form,
 der mit Hilfe von Werkzeugen (javadoc) zu Dokumentation
 in Form von html-Seiten umgewandelt werden kann
 */
```

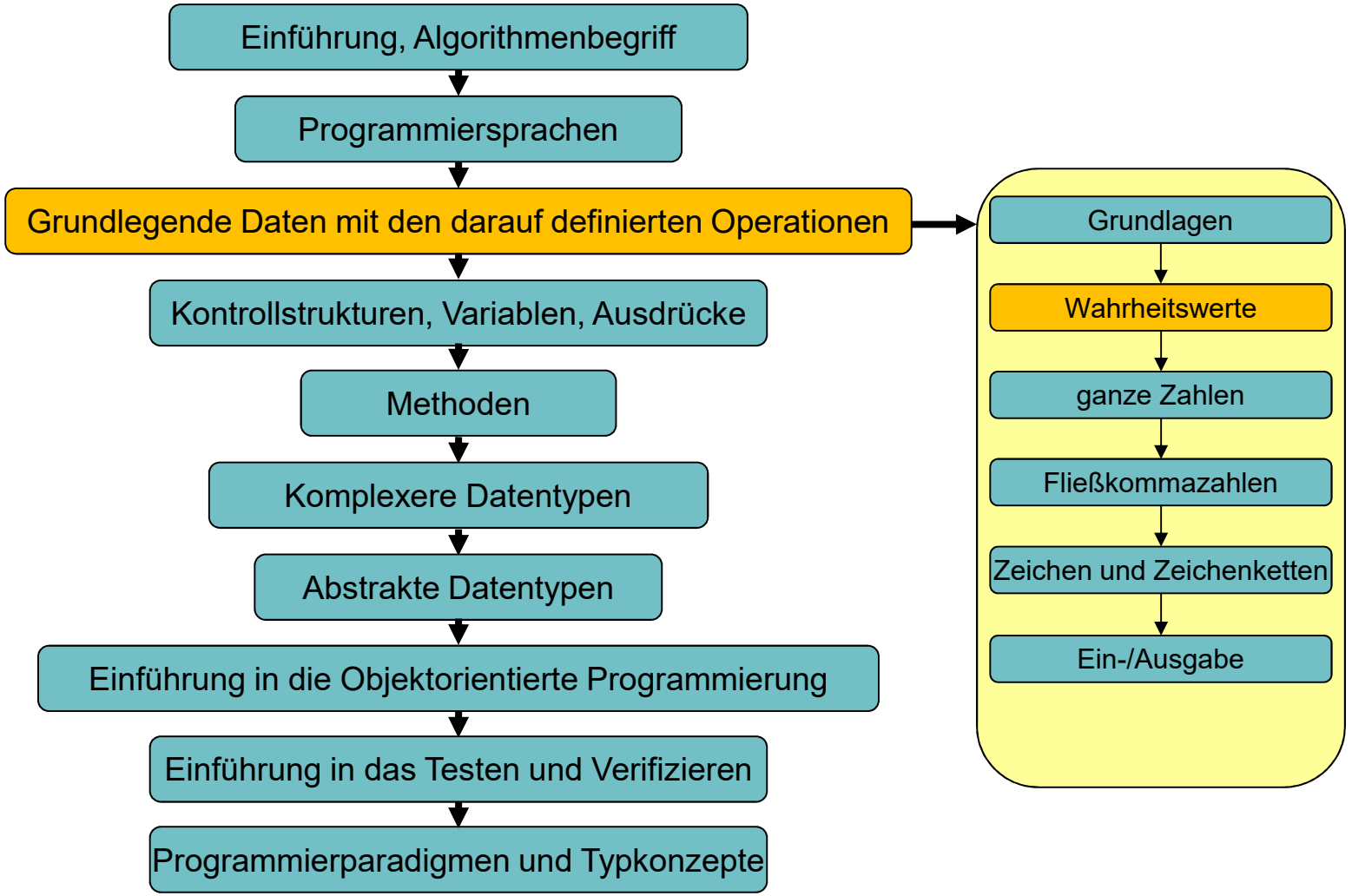
Zwischenstand

- Variablen enthalten genau einen Wert, der sich im Laufe der Programmausführung ändern kann.
- Variablen in Java müssen immer deklariert werden mit Angabe des Datentyps und des Namens der Variablen. Optional ist noch die Angabe eines Anfangswerts möglich.
- Einer Variablen kann man einen neuen Wert zuweisen oder mit dem aktuellen Wert arbeiten (damit rechnen).
- Java-Programme haben immer das gleiche Grundgerüst.

Reflektion

- Was würde sich für wen ändern, wenn man eine Variable nicht deklarieren müsste?
- Was sollte/könnte passieren, wenn man einer Variablen einen Wert zuweist, der nicht dem Typ der Variablen entspricht?

Inhalt dieser Veranstaltung



Wahrheitswerte

- Es gibt genau zwei Wahrheitswerte: B = {`true`, `false`}
- Nutzung in Programmiersprachen bei Vergleichen, Bedingungen,...
  - Selektion: `if (x > y)` ...
  - Iteration: `while (x > y)` ...

- Operationen auf Wahrheitswerten in der Mathematik/Logik:

Operation	math. Funktion	math. Beispiel
Negation	$\neg : B \to B$	$\neg$ `true`
Und	$\wedge : B \times B \to B$	`true` $\wedge$ `false`
Oder	$\vee : B \times B \to B$	`true` $\vee$ `false`
Exklusives Oder	xor : $B \times B \to B$	`true` xor `false`

Wahrheitswerte in Java

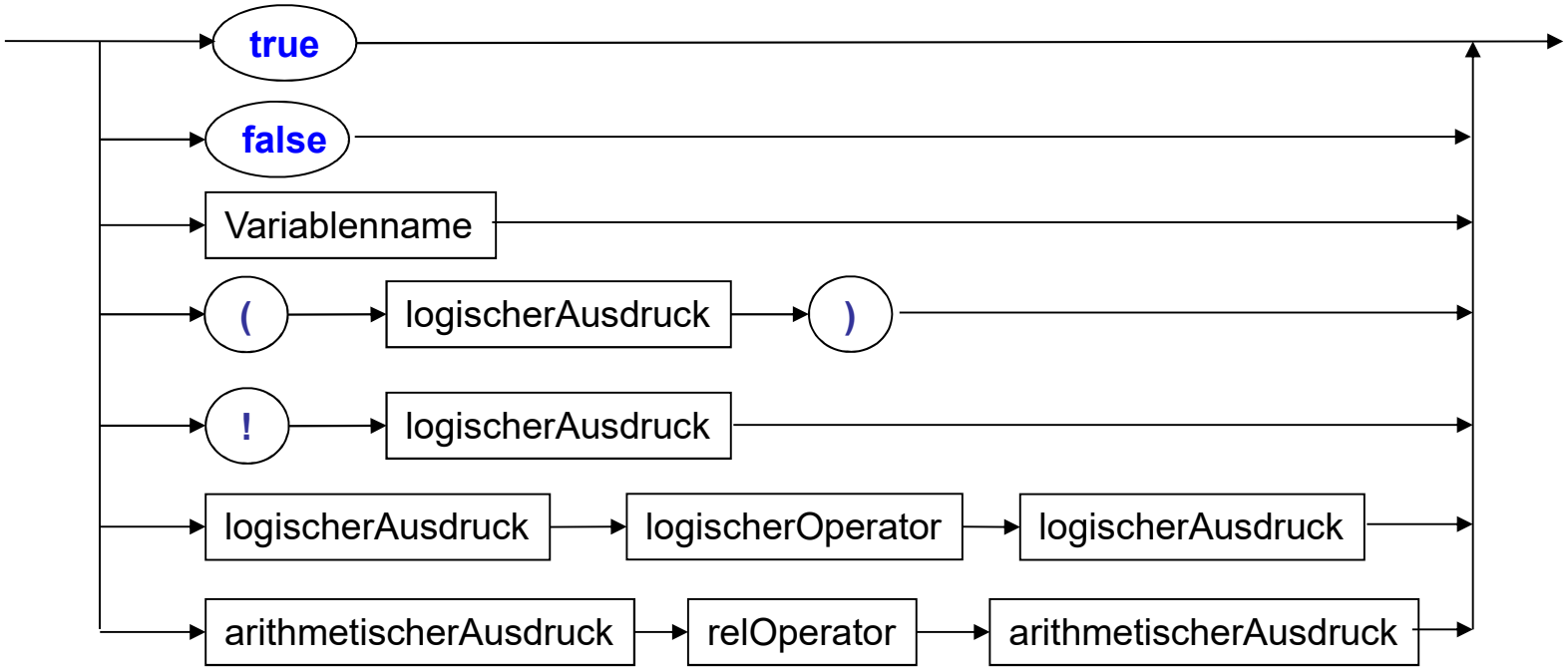
- Wahrheitswerte werden in Java mit 1 Byte gespeichert
- Darstellung durch Bitfolge: 00000000_2=`false`, 00000001_2=`true`
- Die beiden Wahrheitswerte können direkt durch die Schlüsselworte `true` und `false` im Programm angegeben werden (Konstante / Literal)
- Oder sie entstehen als Resultat einer Berechnung (Beispiel: `3 < 4`)

- Java-Operationen auf Wahrheitswerten:

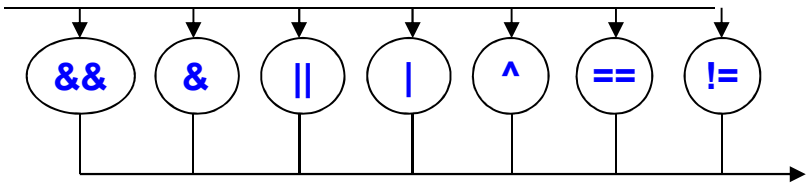
Operation	math. Funktion	math. Beispiel	Java Operator	Java Beispiel
Negation	$\neg : B \to B$	$\neg$ `true`	`!`	`!true`
Und	$\wedge : B \times B \to B$	`true` $\wedge$ `false`	`&` und `&&`	`true && false`
Oder	$\vee : B \times B \to B$	`true` $\vee$ `false`	`|` und `||`	`true || false`
Exklusives Oder	xor : $B \times B \to B$	`true` xor `false`	`^`	`true ^ false`
(Un)Gleichheit			`==` und `!=`	`true == false`

Logische Ausdrücke in Java

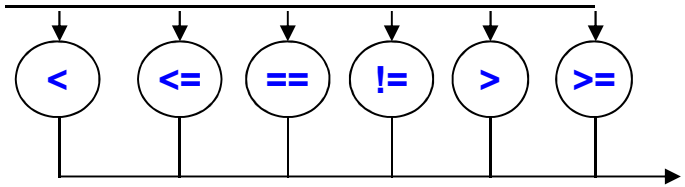
logischerAusdruck



logischerOperator



relOperator



Beispiele zur konkreten Nutzung

`x,y,z` seien drei Variablen von einem Zahlentyp mit den Werten `3,5,7`

Beispielausdruck	Wert des Beispiels	Test worauf?
`x < y`	`true`	x ist kleiner als y
`(x < y) && (x < z)`	`true`	x ist kleiner als y und z
`(x < y) && (y < z)`	`true`	y liegt zwischen x und z
`(0 <= x) && (x <= 9)`	`true`	x liegt im Intervall [0,9]
`!((0 <= x) && (x <= 9))`	`false`	x liegt außerhalb des Intervalls [0,9]
`(x < 0) || (x > 9)`	`false`	x liegt außerhalb des Intervalls [0,9]

Der aus der Mathematik bekannte Ausdruck `0 < x < 9` kann so in Java nicht formuliert werden. Die korrekte Notation dafür ist `(0 < x) && (x < 9)`

Beispielprogramm

```
public class BoolBeispiel {
    public static void main(String[] args) {

        // zwei Variablen vom Typ boolean deklarieren
        boolean b1;
        boolean b2;

        // diesen beiden Variablen jeweils einen Wert zuweisen
        b1 = true;
        b2 = false;

        // Test, ob 3 kleiner als 4 ist. Das Ergebnis speichern wir in b1
        b1 = (3 < 4);

        // Negation des Wertes, der in der Variablen b1 steht
        // Und-Verknuepfung mit dem Test, ob 4 kleiner als 3 ist
        b2 = !b1 && (4 < 3);

    }
}
```

Siehe Syntaxdiagramm „logischerAusdruck“

Siehe Syntaxdiagramm „Variablendeklaration“

Siehe Syntaxdiagramm „Zuweisung“

Beispielprogramm mit Ausgabe

```
public class BoolBeispiel {
    public static void main(String[] args) {
        // zwei Variablen vom Typ boolean deklarieren
        boolean b1, b2;

        // diesen beiden Variablen jeweils einen Wert zuweisen
        b1 = true;
        b2 = false;

        // Wir geben das Ergebnis einiger Operationen aus
        System.out.println("Der Wert von b1 ist " + b1);
        System.out.println("Der Wert von b2 ist " + b2);

        // Test, ob 3 kleiner als 4 ist. Das Ergebnis speichern wir in b1
        b1 = (3 < 4);

        // Negation des Wertes, der in der Variablen b1 steht
        // Und-Verknuepfung mit dem Test, ob 4 kleiner als 3 ist
        b2 = !b1 && (4 < 3);

        System.out.println("Der Wert von b1 ist jetzt " + b1);
        System.out.println("Der Wert von b2 ist jetzt " + b2);
    }
}
```

Ausgabe von Werten auf dem Bildschirm

- Mit `System.out.println(...);` läßt sich eine Zeichenkette auf dem Bildschirm ausgeben
- Innerhalb der Klammern steht genau ein Ausdruck
- Dieser Ausdruck wird ausgewertet und das Ergebnis (ein Wert) als Zeichenkette auf dem Bildschirm ausgegeben und diese Ausgabezeile anschließend umgebrochen
- An dieser Stelle schon vorab einige erste Anmerkungen zu Fällen, die auftreten können:
  - Ein String (Zeichenkette) wird notiert als `"text"`. Der Wert dieses Strings ist der Text zwischen den Anführungszeichen, der von `System.out.println` genau so auf dem Bildschirm ausgegeben wird.
  - Man kann zwei Strings mit `+` zu einem zusammenhängenden String verketten.
    Beispiel: `"abc" + "def"` ergibt `"abcdef"`
  - Wenn ein Operand in einem +-Ausdruck ein String ist, so wird der andere Operand automatisch in einen String umgewandelt
    Beispiel: `"text" + 3` ergibt `"text3"`

Zwischenstand

- Der Java-Typ `boolean` entspricht der Menge der Wahrheitswerte mit den üblichen Operationen darauf.
- Ein logischer Ausdruck setzt sich aus Konstanten, (logischen) Variablen und logischen Operatorausdrücken zusammen.
- Logische Ausdrücke werden z.B. in Bedingungen genutzt (falls … dann …).
- Mit `System.out.println` kann man in Java eine Ausgabe auf dem Bildschirm bewirken.

Reflektion

- Formulieren Sie die Bedingung, dass ein Wert x weder 11 noch 13 ist.
- Formulieren Sie dies konkret in Java.
- Wieso könnte `System.out.println` für Sie in der Programmentwicklung hilfreich sein?

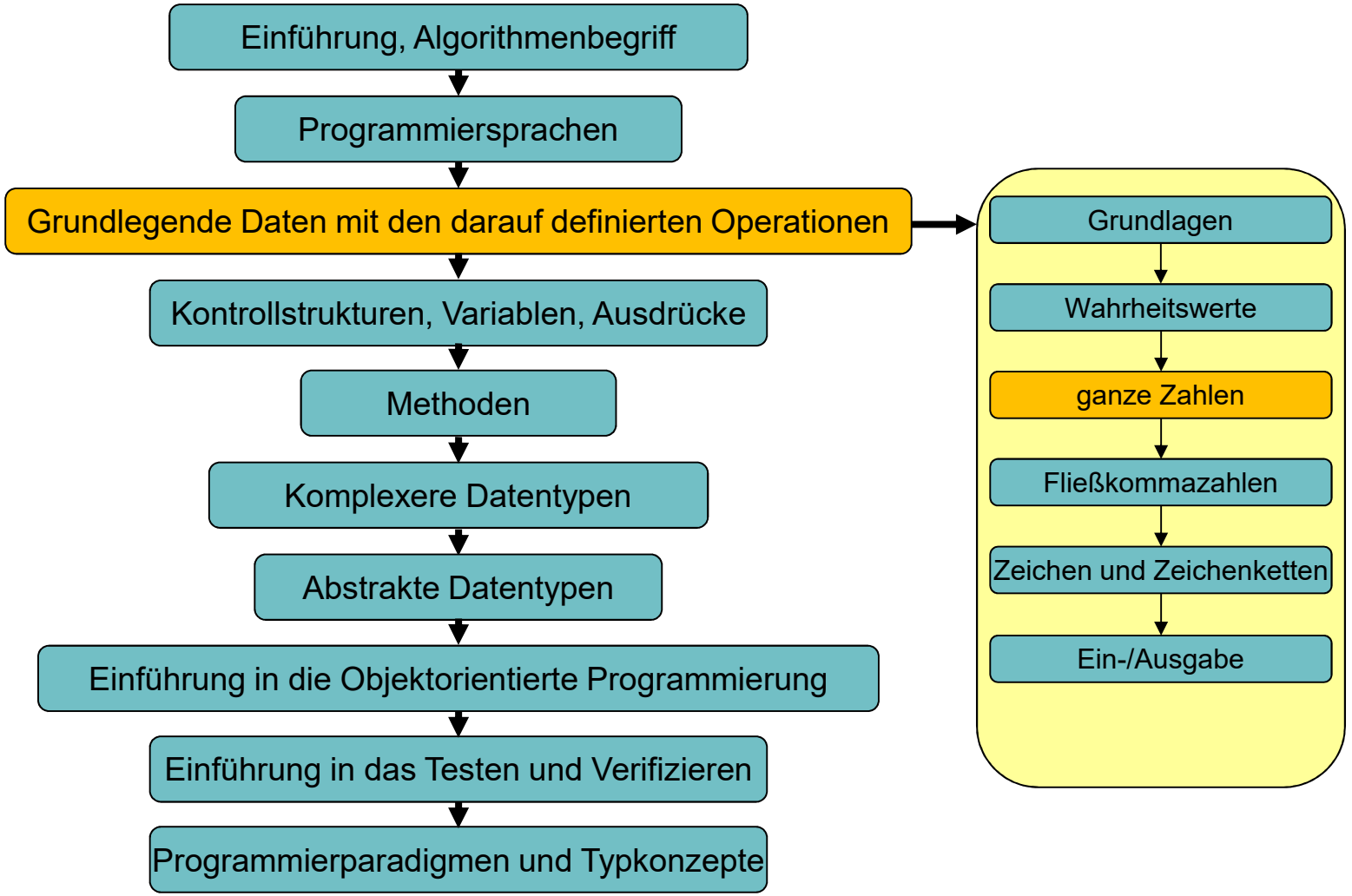
Zusammenfassung

- Bits und Bytes in einem Rechner zur Darstellung von Werten
- Datentypen in Programmiersprachen (Wertemenge mit Operationen und Kodierung)
- Eine Variable speichert genau ein Datum

- Steckbrief Wahrheitswerte:

Name	`boolean`
Wertemenge	{true, false}
Kodierung	1 Byte, 00000000_2=false, 00000001_2=true
Konstanten	`true`, `false`
Operationen	`!`, `&`, `&&`, `|`, `||`, `^`
Nutzung	in Bedingungen (Selektion, Iteration)
Besonderheiten	–

Inhalt dieser Veranstaltung



Zahlen

- In der Mathematik gibt es nicht "die Zahlen"
- Zahlenmengen N, Q, R, C,... mit ihren jeweiligen Eigenschaften (Wertebereich, Operationen, Abgeschlossenheit,...)
- Je nachdem, was man benötigt, sucht man den geeigneten Zahlenraum aus und arbeitet in diesem

- In Programmiersprachen ist dies ähnlich. Hier werden wir ganze Zahlen und reelle Zahlen unterscheiden.
- Fragen, die zu klären sind:
  - Wie kann ich natürliche und ganze Zahlen darstellen? Beispiel: 3, -3
  - Wie lassen sich Brüche darstellen? Beispiel: 3/4
  - Wie lassen sich "Kommazahlen" darstellen? Beispiel: 0,1
  - Wie lassen sich irrationale Zahlen darstellen? Beispiel: π
  - Welche Operationen gibt es auf den jeweiligen Zahlenmengen?

Startpunkt: natürliche Zahlen

- Frage: wie kann ich natürliche Zahlen mit Bits kodieren?

- Ein Problem: es gibt unendliche viele natürlich Zahlen
- Mögliche Näherungslösung: starte bei 0 und nehme nächste noch nicht genutzte Bitkombination für nächste noch nicht kodierte Zahl

Zahl	0	1	2	3	4	5	6	7	8	...
Kodierung	0	1	10	11	100	101	110	111	1000	...

- Vorteile:
  - Nur so viele Bits wie notwendig werden verwendet
  - Größere Zahlen sind (mit vielen Bits) darstellbar. Aber irgendwann ist natürlich Schluss, weil der vorhandene Speicher nicht mehr ausreicht.

- Nachteil: variable Anzahl an Bits für Zahlen. Aber Prozessoren haben Register fester Länge!

Zweiter Ansatz für natürliche Zahlen

- Berücksichtigung, dass Register feste Länge haben: nehme eine feste Anzahl an Bits zur Kodierung
- Vorteil: feste Länge
- Nachteil: Wertebereich ist beschränkt (zur Erinnerung: mit n Bits lassen sich genau 2^n verschiedene Werte kodieren)
- Mit n Bits ließen sich also natürliche Zahlen aus dem Intervall $[0,2^n-1]$ darstellen

n=	Wertebereich	würde reichen für
8	[0, 255]	Anzahl Studierende in einer Übungsgruppe
16	[0, 65535]	Entfernungsangaben in km auf Erde (Äquator ca. 40.000 km)
32	[0, 4.294.967.295]	Einwohner Deutschland (ca. 82 Mio)
64	$[0, \approx 1{,}8 \cdot 10^{19}]$	Schuldenstand Bund Deutschland (ca. 1,7 Billionen €)
128	$[0, \approx 3{,}4 \cdot 10^{38}]$	Anzahl Moleküle in 1 Liter 4° warmen Wasser ($33 \cdot 10^{24}$)
?		Wassermoleküle auf der Erde (Wasservorräte Erde ca. $1{,}4 \cdot 10^{21}$ Liter mal $33 \cdot 10^{24}$ Moleküle/l)
?		Atome auf der Erde, im Sonnensystem, Weltall,...

Stellenwertsystem

- Zur Erinnerung: Dezimalsystem
- Die Zahl 4321 wird im Dezimalsystem (Basis 10) beschrieben durch die Ziffernfolge 4,3,2,1 und der Wert errechnet sich aus $4 \cdot 1000 + 3 \cdot 100 + 2 \cdot 10 + 1 \cdot 1$ oder $4 \cdot 10^3 + 3 \cdot 10^2 + 2 \cdot 10^1 + 1 \cdot 10^0 = 4321_{10}$

- Ein Zahlensystem, in dem einer Ziffer in Abhängigkeit ihrer Position eine Wertigkeit zugeordnet wird, nennt man Stellenwertsystem

- Menschen haben 10 Finger, deshalb Basis 10 gebräuchlich (Dezimalsystem)
- Bits haben 2 mögliche Zustände. Deshalb jetzt: Basis 2 (Dualsystem)
- Der Wert der Zahl 1011_2 errechnet sich durch $1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 8_{10} + 0_{10} + 2_{10} + 1_{10} = 11_{10}$

- Oder allgemein für eine Basis B und Ziffern $z_{n-1} \ldots z_0$:

$$Wert(z_{n-1}z_{n-2}\ldots z_1 z_0)_B = z_{n-1} \cdot B^{n-1} + z_{n-2} \cdot B^{n-2} + \ldots + z_1 \cdot B^1 + z_0 \cdot B^0 = \sum_{i=0}^{n-1} z_i \cdot B^i$$

Einige Werte im Dualsystem

Bitfolge	Wert
000	$0 = 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
001	$1 = 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
010	$2 = 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
011	$3 = 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$
100	$4 = 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
101	$5 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
110	$6 = 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
111	$7 = 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

Umwandlung von Dual nach Dezimal: ausrechnen des Polynoms

$$Wert(z_{n-1}z_{n-2}...z_1z_0)_B = z_{n-1} \cdot B^{n-1} + z_{n-2} \cdot B^{n-2} + ... + z_1 \cdot B^1 + z_0 \cdot B^0 = \sum_{i=0}^{n-1} z_i \cdot B^i$$

Umwandlung von Dezimal nach Dual

- Aufgabe: bestimme für einen gegebenen Wert die Ziffern zur jeweiligen Stelligkeit

- Beispiel im Dezimalsystem:

$1234 / 10^3$	=	1 Rest 234
$234 / 10^2$	=	2 Rest 34
$34 / 10^1$	=	3 Rest 4
$4 / 10^0$	=	4 Rest 0

- Beispiel im Dualsystem (fortgesetzte Division durch 2-Potenzen):

$1234 / 2^{10}$	= 1234 / 1024	= 1 Rest 210
$210 / 2^9$	= 210 / 512	= 0 Rest 210
$210 / 2^8$	= 210 / 256	= 0 Rest 210
$210 / 2^7$	= 210 / 128	= 1 Rest 82
$82 / 2^6$	= 82 / 64	= 1 Rest 18
$18 / 2^5$	= 18 / 32	= 0 Rest 18
$18 / 2^4$	= 18 / 16	= 1 Rest 2
$2 / 2^3$	= 2 / 8	= 0 Rest 2
$2 / 2^2$	= 2 / 4	= 0 Rest 2
$2 / 2^1$	= 2 / 2	= 1 Rest 0
$0 / 2^0$	= 0 / 1	= 0 Rest 0

Oktal- und Hexadezimal

- Bitfolgen mit 32 oder 64 Bits sind für Menschen schwer lesbar
- Bei einem Wert in Dezimaldarstellung ist schwer ersichtlich, welche Bits 0 oder 1 sind für eine Darstellung zur Basis 2
- Deshalb:
  - Oktaldarstellung zur Basis 8 und Ziffern 0,...,7
  - Hexadezimaldarstellung zur Basis 16 und Ziffern 0,...,9,A,...,F

- Umwandlungen:
  - Oktal / Hexadezimal $\rightarrow$ Dezimal: Ausrechnen Polynom
  - Dezimal $\rightarrow$ Oktal: fortgesetzte Division durch 8-Potenzen
  - Dezimal $\rightarrow$ Hexadezimal: fortgesetzte Division durch 16-Potenzen

- Vorteil der beiden Darstellungen:
  - Oktal- und Hexadezimaldarstellungen lassen auf einfachere Weise erkennen, welche Bits an/aus sind
  - Sie sind damit besser geeignet zur Darstellung von Konstanten, in denen Bits eine Rolle spielen

Beispiele

- Dezimal → Hexadezimal

$1234 / 16^2$	$= 1234 / 256$	$= 4$ Rest 210
$210 / 16^1$	$= 210 / 16$	$= D$ Rest 2 ($D_{16} = 13_{10}$)
$2 / 16^0$	$= 2 / 1$	$= 2$ Rest 0

- Dezimal → Oktal

$1234 / 8^3$	$= 1234 / 512$	$= 2$ Rest 210
$210 / 8^2$	$= 210 / 64$	$= 3$ Rest 18
$18 / 8^1$	$= 18 / 8$	$= 2$ Rest 2
$2 / 8^0$	$= 2 / 1$	$= 2$ Rest 0

Ganze Zahlen

- Erster Ansatz: verwende Darstellung natürlicher Zahlen und füge ein Bit für das Vorzeichen hinzu
- Nachteile:
  - Die Zahl 0 hat zwei Darstellungen (+0, -0)
  - a + (-b) müsste anders behandelt werden als a-b

- Deshalb: Zweierkomplementdarstellung

$$Wert(z_{n-1}z_{n-2}...z_1z_0)_2 = -z_{n-1}\cdot 2^{n-1} + \sum_{i=0}^{n-2} z_i \cdot 2^i$$

- Alle Zahlen im Intervall $[-2^{n-1},+2^{n-1}-1]$ sind darstellbar (2^n verschiedene Werte)
- Oberstes Bit hat eine Vorzeichenfunktion (1=negative Zahl)
- Die Gesamtanzahl n an Bits der Darstellung spielt jetzt eine Rolle (vorher waren zum Beispiel führende Nullen kein Problem)

Beispiel

n=3 Bits werden zur Darstellung gewählt

Bitfolge	Wert
000	$0 = -0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
001	$1 = -0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
010	$2 = -0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
011	$3 = -0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$
100	$-4 = -1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0$
101	$-3 = -1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$
110	$-2 = -1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0$
111	$-1 = -1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$

Zwischenstand

- Auch in Rechnern / Programmiersprachen gibt es verschiedene Arten/Typen von Zahlen.
- Zur Darstellung von ganzen Zahlen nutzt man die Zweierkomplement-darstellung gemäß der Formel $Wert(z_{n-1}z_{n-2}...z_1z_0)_2 = -z_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} z_i \cdot 2^i$
- Eine Umrechnung eines Werts von einem Zahlensystemen zu einem anderen kann nach dem gleichen Schema erfolgen (z.B. Dezimal, Binär, Oktal, Hexadezimal).

Reflektion

- Wieviele verschiedene Werte lassen sich mit 3 Oktalziffern darstellen? Mit 2 Hexadezimalziffern?
- Hat die Zahl mit der Darstellung 110_2 einen positiven oder negativen Wert? Wieso? Die Zahl 001_2?

Ganze Zahlen in Java

- In Prozessoren wird grundsätzlich die Zweierkomplementdarstellung zur Darstellung von ganzen Zahlen genutzt (positive und negative Zahlen)
- Prozessoren unterstützen Rechenbefehle auf 8, 16, 32 und 64 Bit Zahlen

- Frage: wovon hängt die Wahl der Anzahl von Bits zur Darstellung eines Wertes ab?
- Antwort: es kommt darauf an...
  - Ausreichend Bits zur Darstellung aller auftretenden Werte
  - Nicht zu viele Bits, dass Speicherplatz nicht unnötig verschwendet wird

- Deshalb gibt es in Java mehrere Datentypen für ganzzahlige Werte
- Der Programmierer muss entscheiden, welcher dieser Datentypen für die jeweilige Nutzung sinnvoll ist

- Später: ganzzahlige Datentypen werden auch zur Darstellung von Bit-Folgen genutzt

Ganzzahlige Datentypen in Java

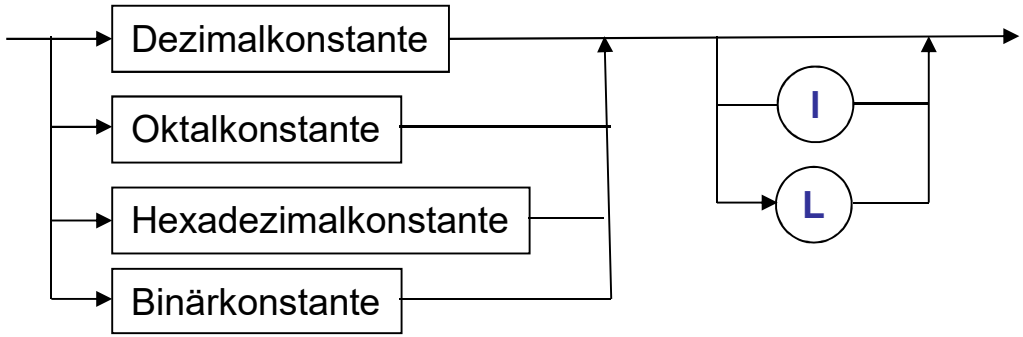
Datentyp	Anzahl Bits	Anzahl Bytes	Wertebereich
`byte`	8	1	[-128, +127]
`short`	16	2	[-32.768, +32.767]
`int`	32	4	[-2.147.483.648, +2.147.483.647]
`long`	64	8	[-9.223.372.036.854.755.808, +9.223.372.036.854.755.807]

- Übereinkunft: wenn nichts dagegen spricht (zu kleiner Wertebereich, zu hoher Speicherbedarf), verwendet man im Programm für ganzzahlige Werte den Datentyp `int`

- Fragen:
  - Wenn ich im Programm 3 schreibe, von welchem Typ ist dies? `int`
  - Und wie kann ich explizit eine Zahl zum Beispiel des Datentyps `long` angeben? siehe Syntax nächste Folie

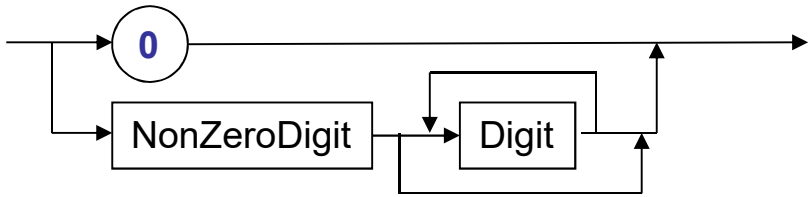
Syntax ganzzahliger Konstanten

Ganzzahlkonstante

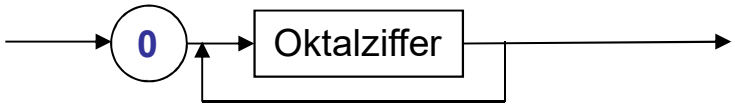


Hinweis: Unterstriche innerhalb einer Zahlenangabe möglich (hier nicht aufgeführt)

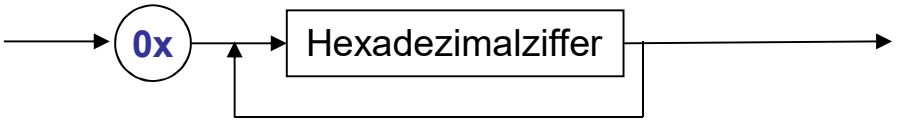
Dezimalkonstante



Oktalkonstante

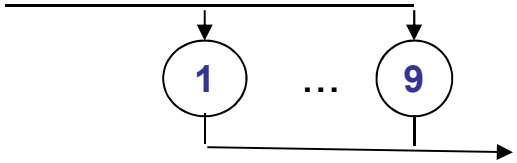


Hexadezimalkonstante

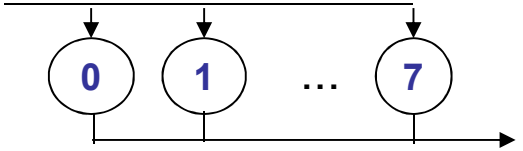


Binärkonstante

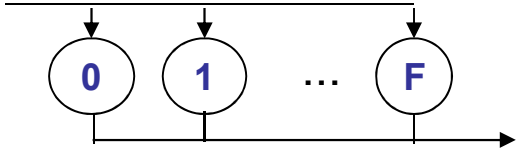
NonZeroDigit



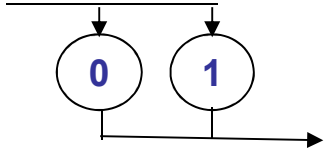
Oktalziffer



Hexadezimalziffer



Binärziffer



Beispiele

- Dezimalangabe:
  - `0, 3, -32777` (int)
  - `0l, 3L` (long)
- Oktalangabe:
  - `00, 01, 017` (int)
  - `00l, 017L` (long)
- Hexadezimalangabe:
  - `0x1, 0x17` (int)
  - `0x1l, 0x17L` (long)
- Binärangabe:
  - `0b1, 0b10` (int)
  - `0b1l, 0b10L` (long)

- Oktal-, Hexadezimal- und Binärkonstanten sind u.a. sinnvoll anwendbar im Zusammenhang mit Bitoperationen

Operationen auf ganzzahligen Werten

arithmetische Operationen:

Operator	Beispiel	Wert Beispiel	Wirkung
+	+3	+3	Vorzeichenplus
-	-3	-3	Vorzeichenminus
+	3+4	7	Addition
-	3-4	-1	Subtraktion
*	3*4	12	Multiplikation
/	5/3	1	ganzzahlige Division
%	5%3	2	Rest bei ganzzahliger Division (Modulo)

arithmetische Vergleichsoperationen (liefern Ergebnis vom Typ `boolean`):

Operator	Beispiel	Wert Beispiel	Wirkung
<	3<4	true	kleiner
<=	3<=4	true	kleiner oder gleich
==	3==4	false	gleich
!=	3!=4	true	ungleich
>	3>4	false	größer
>=	5>=3	true	größer oder gleich

Beispiel

```
/**
 * Beispiel zu Operationen aus dem Datentyp int
 */
public class IntBeispiel {
    public static void main(String[] args) {

        // zwei Variablen vom Typ int definieren
        int i1, i2;

        // diesen beiden Variablen Werte zuweisen
        i1 = 3;
        i2 = i1 + 8;

        // Wir geben das Ergebnis einiger Operationen aus
        System.out.println("i1 * i2: " + (i1 * i2));
        System.out.println("11 / 3: " + (11 / 3));
        System.out.println("11 % 3: " + (11 % 3));
    }
}
```

Anwendung der Modulo-Operation

- Aufgabe: Quersumme einer 2-stelligen Dezimalzahl ermitteln

- Idee:
  - Eine beliebige natürliche Zahl modulo 10 liefert mir die letzte Dezimalziffer
  - Eine Zahl dividiert durch 10 "wirft" die letzte Dezimalziffer weg

```
public class Quersumme {
    public static void main(String[] args) {
        int zahl = 21;    // Beispielzahl
        int summe = 0;     // Quersumme

        // letzte Ziffer
        summe = summe + (zahl % 10);          0+1=1
        zahl = zahl / 10;                     2

        // vorletzte Ziffer
        summe = summe + (zahl % 10);          1+2=3
    }
}
```

Schaltjahre

- Aufgabenstellung: bestimme, ob ein Jahr ein Schaltjahr ist
- Ein Jahr ist ein Schaltjahr, wenn die Jahreszahl durch 4 teilbar ist. Ist sie zusätzlich durch 100 teilbar, so ist das Jahr doch kein Schaltjahr. Ist die Jahreszahl aber zusätzlich durch 400 teilbar, so ist das Jahr doch ein Schaltjahr.
- Lösung:

```
public class Schaltjahr {
    public static void main(String[] args) {
        // Beispiel fuer ein zu ueberpruefendes Jahr
        int jahr = 2013;
        boolean istSchaltjahr;

        /* Eine Jahrzahl ist durch 4 und nicht durch 100 teilbar
           oder sie ist durch 400 teilbar
        */
        istSchaltjahr = ( (   (jahr % 4) == 0)
                            && (!((jahr % 100) == 0)))
                          || ((jahr % 400) == 0);

        System.out.println("Das Jahr " + jahr
                           + " ist ein Schaltjahr? " + istSchaltjahr);
    }
}
```

Zwischenstand

- Java (und Prozessoren) kennen Ganzzahlenmengen unterschiedlicher Größe
- Die interne Darstellung ist immer die Zweierkomplementdarstellung
- Für die Angabe in einem Programm gibt es mehrere Möglichkeiten
- Es existieren als Operationen u.a. die üblichen Grundrechenarten
- Ganzzahlige Division und Modulo-Bildung sind zu beachten

Reflektion

- Wie könnte man zu einer beliebigen Zahl statt der Quersumme der Dezimalziffern die Quersumme der Binärziffern ermitteln?