PGdP

Datentypen

Java VM

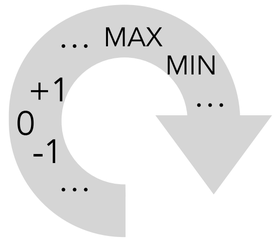
P-Aufgaben

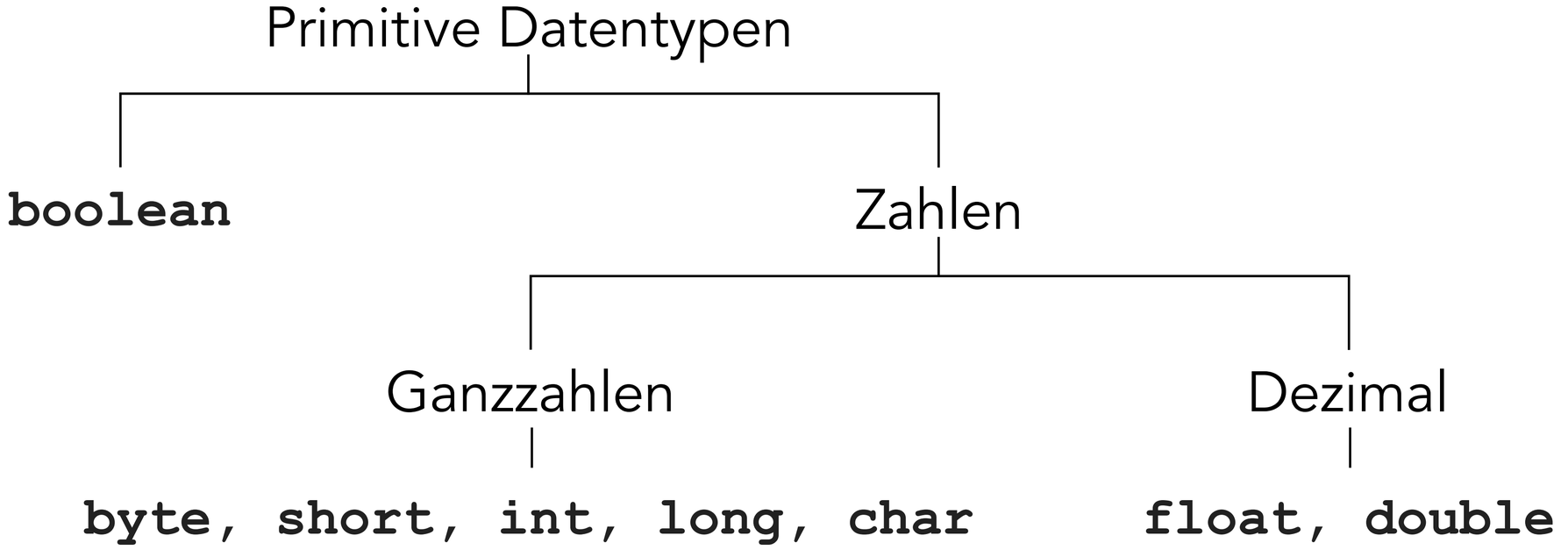
Folien: go.tum.de/904005

Datentypen

Achtung! Überlauf von Ganzzahlen (Bsp. Integer):

`Integer.MAX_VALUE + 1 = Integer.MIN_VALUE`





Datentypen

Ganzzahlen

Typ	Platz (in Bit)	Kleinster Wert	Größter Wert
byte	8	-128	127
short	16	-32.768	32.767
char	16	0	65635
int	32	-2.147.463.648	2.147.463.647
long	64	-2^{63} =-9.223.372.036.854.775.808	$2^{63}-1$ =9.223.372.036.854.775.807

Wahrheitswerte

boolean 8 **true** oder **false**

Datentypen

Character werden intern wie Zahlen behandelt. (Nächste Woche mehr dazu!)

Ganzzahlen

Typ	Platz (in Bit)	Kleinster Wert	Größter Wert
byte	8	-128	127
short	16	-32.768	32.767
char	16	0	65635
int	32	-2.147.463.648	2.147.463.647
long	64	-2^{63} =-9.223.372.036.854.775.808	$2^{63}-1$ =9.223.372.036.854.775.807

Wahrheitswerte

boolean	8	**true** oder **false**

Datentypen

Gleitkommazahlen

Typ	Platz (in Bit)	Kleinster Wert	Größter Wert
`float`	32	ca. $-3.4*10^{38}$	ca. $3.4*10^{38}$
`double`	64	ca. $-1.7*10^{308}$	ca. $1.7*10^{308}$

Achtung: Gleitkommazahlen sind keine Festkommazahlen!

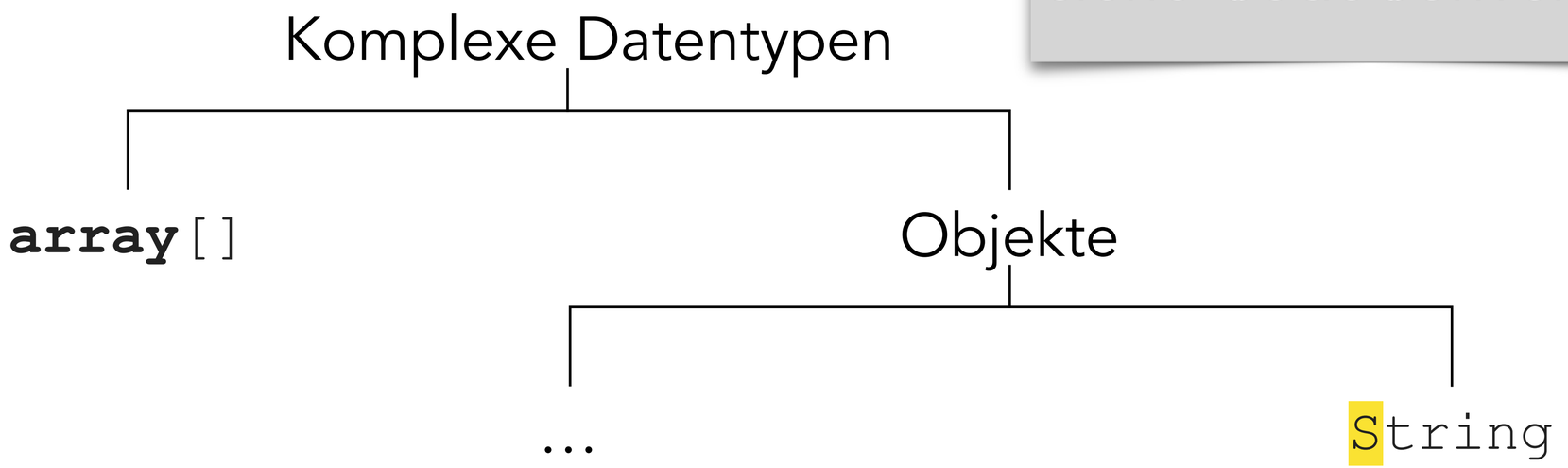
- Gleitkommazahlen verlieren an Genauigkeit!

`((0.1 + 0.1) == 0.2)` ≡ **`false`**

Datentypen

Komplexe Datentypen

String wird groß geschrieben, int klein.
siehe CodeConvention



Datentypen

String

- Strings sind Objekte
- es stehen Objektmethoden↗ zur Verfügung

Konkatenation

```
"H"+"a"+"ll"+"o" ⇒ "Hallo"

"Zahl: "+26 ⇒ "Zahl: 26"

1+6+" ist nicht "+6+1 ⇒ 7 ist nicht 61
```

wird zuerst als int ausgewertet

(Steht vor erster String Komponente)

wird als String ausgewertet

(Steht hinter erster String Komponente)

Java VM

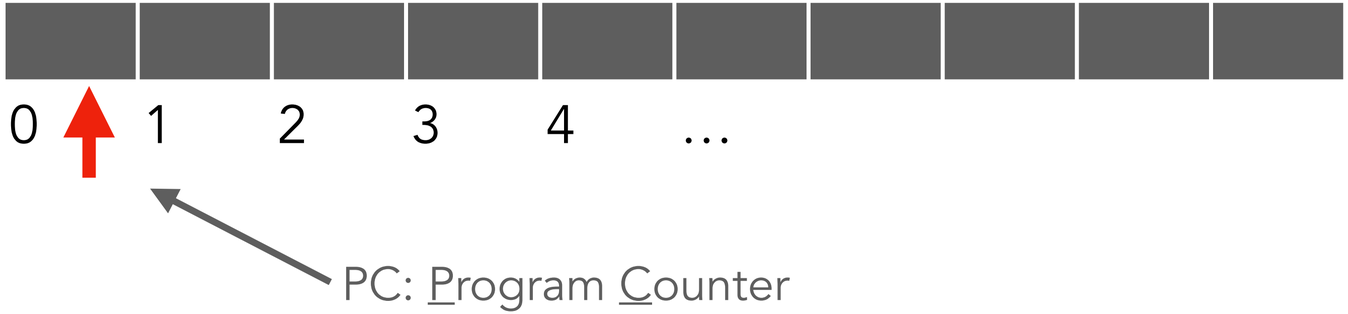
Funktionsweise

- Befehle, die **Argumente** benötigen, erwarten diese **oben auf dem Stack**
- Nach der Benutzung werden die Elemente vom Stack herunter geworfen
- Mögliche **Ergebnisse von Operationen** werden **oben auf dem Stack** abgelegt

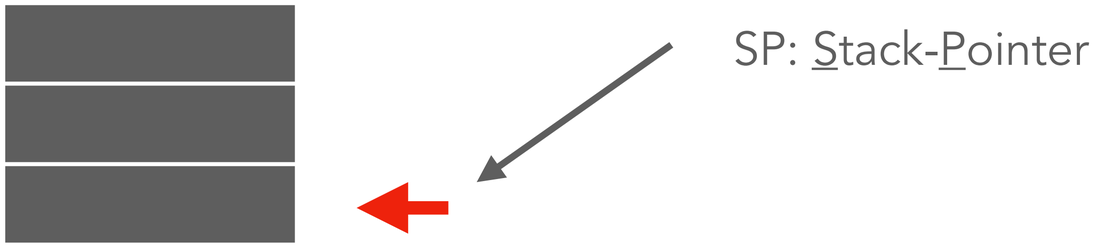
Java VM

Code und Stack

- Code liegt in JVM Befehlen vor

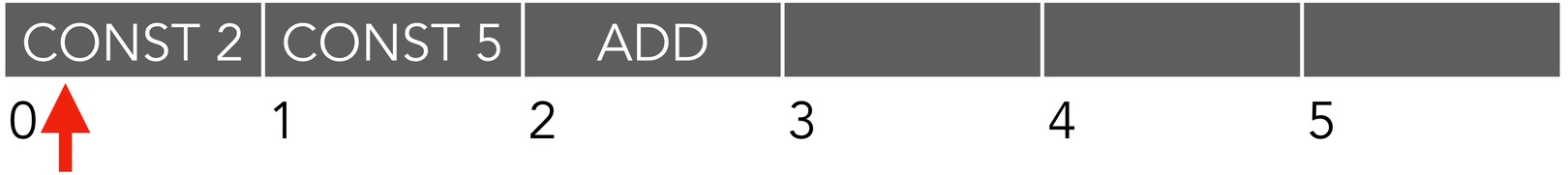


- Stack mit Speicher



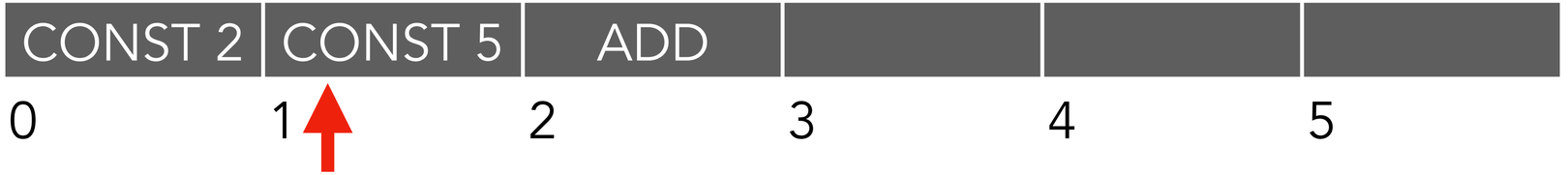
Java VM

Binäre Operationen (Bsp. Addiern)



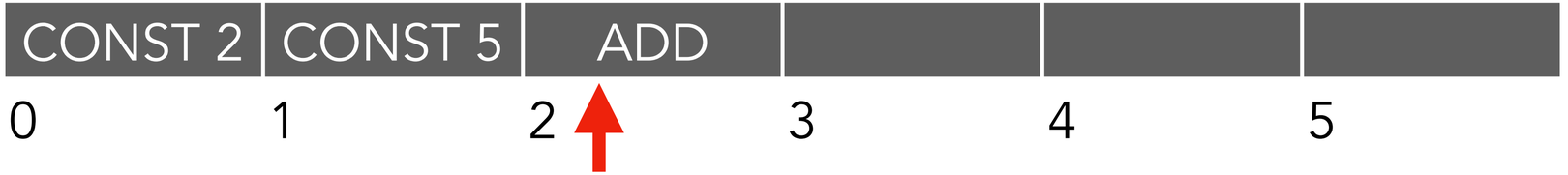
Java VM

Binäre Operationen (Bsp. Addiern)



Java VM

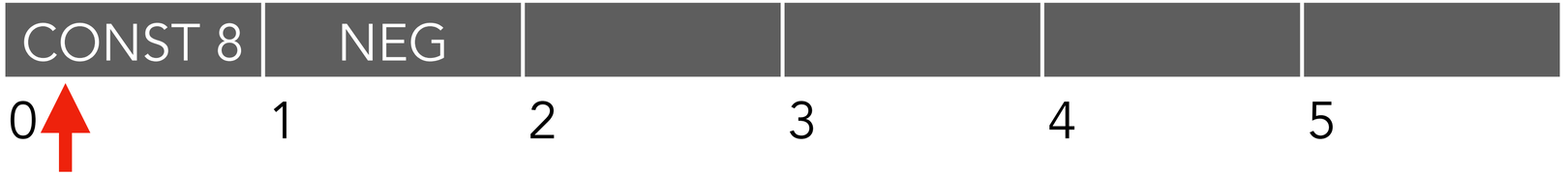
Binäre Operationen (Bsp. Addiern)



ADD konsumiert zwei Elemente vom Stack und legt ein neues Element ab.

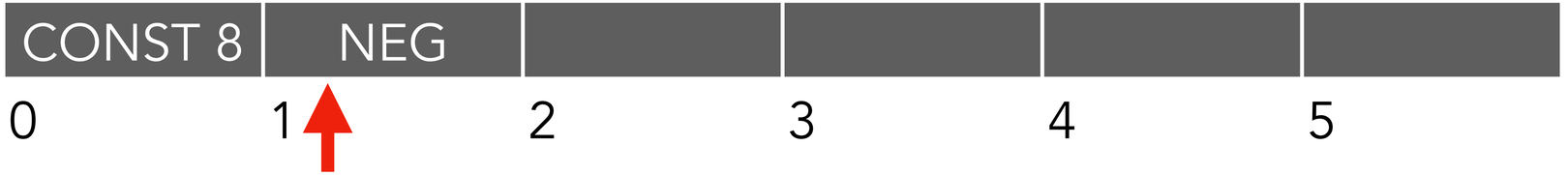
Java VM

Unäre Operationen (Bsp. Negieren)



Java VM

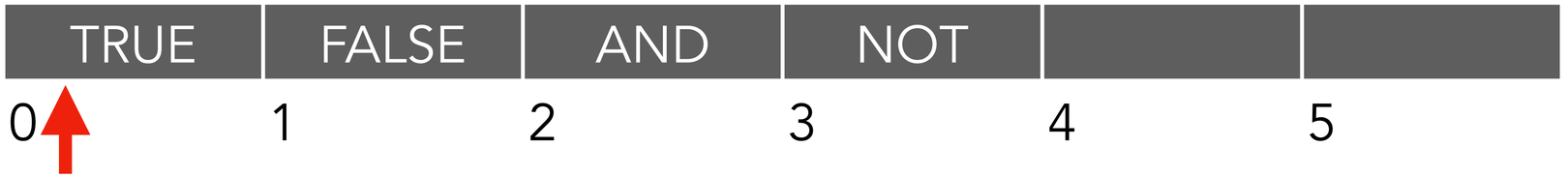
Unäre Operationen (Bsp. Negieren)



NEG konsumiert ein Element vom Stack und legt ein neues Element ab.

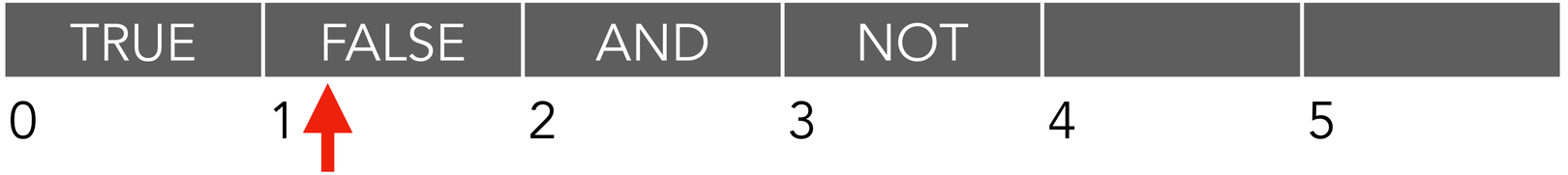
Java VM

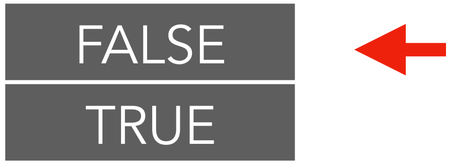
Vergleichsoperationen (Bsp. Cond ¬(true ∧ false))



Java VM

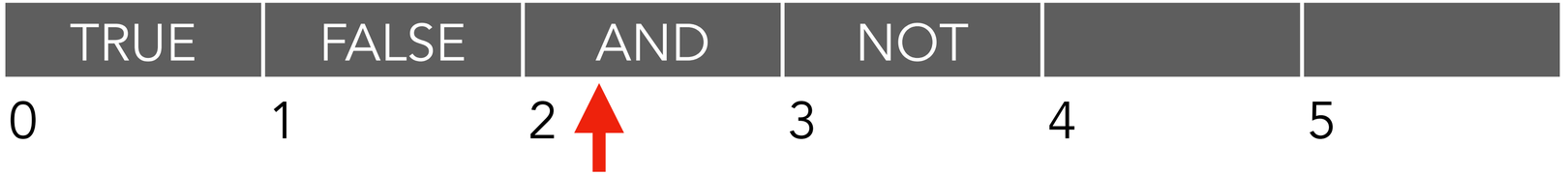
Vergleichsoperationen (Bsp. Cond ¬(true ∧ false))





Java VM

Vergleichsoperationen (Bsp. Cond ¬(true ∧ false))



AND konsumiert zwei Element vom Stack und legt ein neues Element ab.

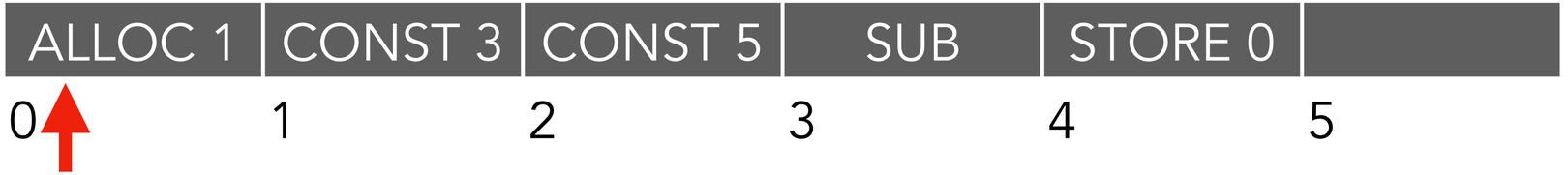
Java VM

Vergleichsoperationen (Bsp. Cond ¬(true ∧ false))

NOT konsumiert ein Element vom Stack und legt ein neues Element ab.

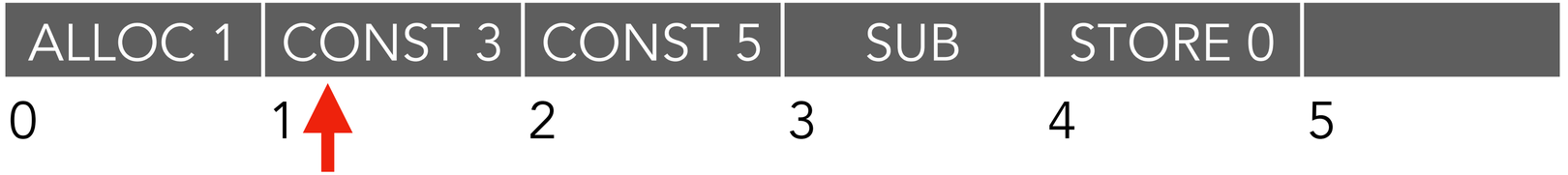
Java VM

Zuweisung (Reservierung von Speicher (**int** a;), Zuweisung $3 - 5$)



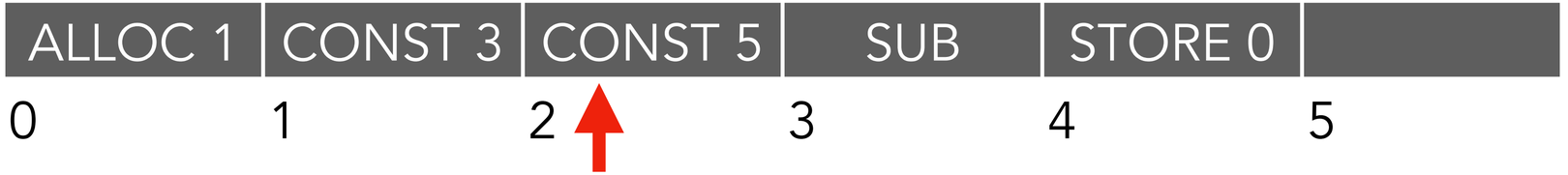
Java VM

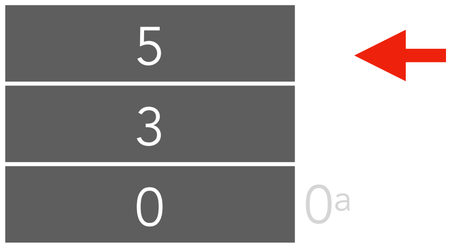
Zuweisung (Reservierung von Speicher (**int** a;), Zuweisung $3 - 5$)



Java VM

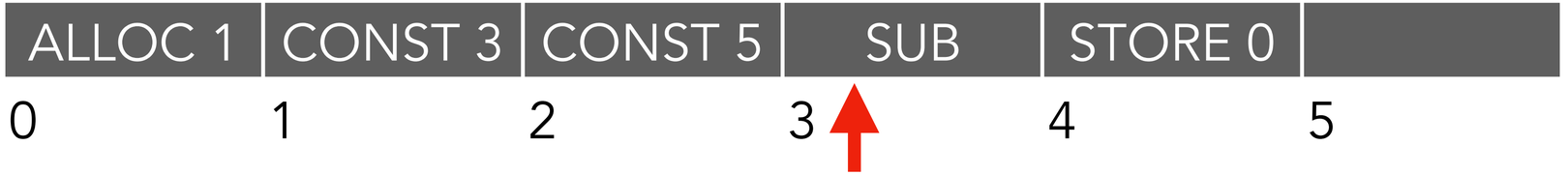
Zuweisung (Reservierung von Speicher (**int** a;), Zuweisung $3 - 5$)

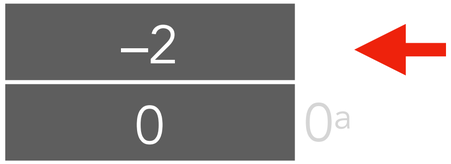




Java VM

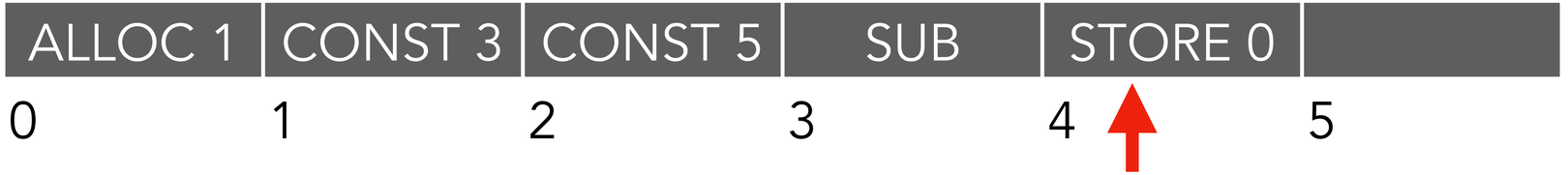
Zuweisung (Reservierung von Speicher (**int** a;), Zuweisung $3 - 5$)





Java VM

Zuweisung (Reservierung von Speicher (**int** a;), Zuweisung $3 - 5$)



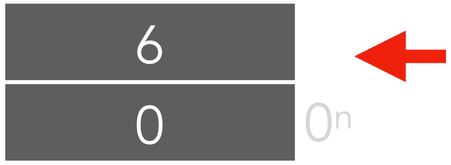
Java VM

Kleines. Programm (Einlesen, falls $(n < 5)$, write$(n + 1)$


```
1 int n;
2 n = readInt();
3 if(n < 5)
4   write(n+1);
```

Java VM

Kleines. Programm (Einlesen (6), falls $(n < 5)$, write$(n + 1)$




```
1 int n;
2 n = readInt();
3 if(n < 5)
4   write(n+1);
```

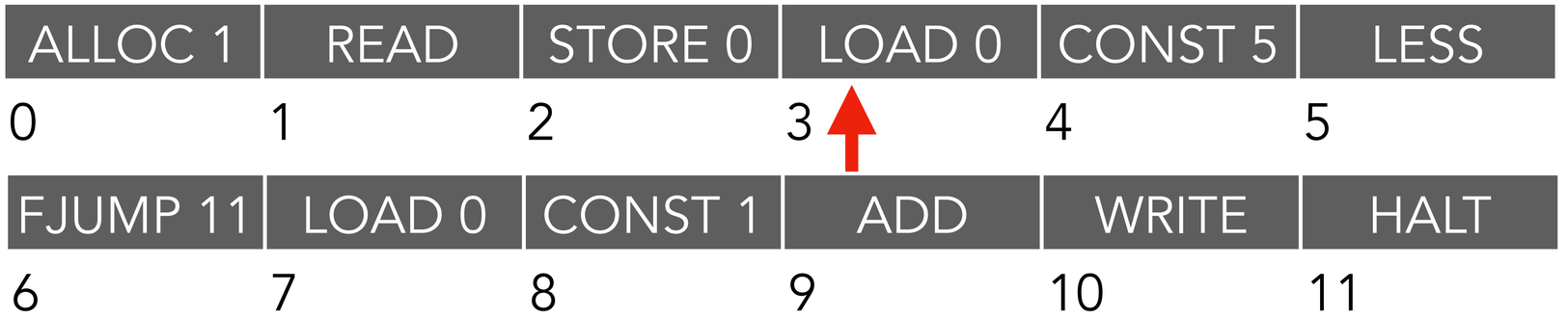
Java VM

Kleines. Programm (Einlesen (6), falls $(n < 5)$, write$(n + 1)$


```
1 int n;
2 n = readInt();
3 if(n < 5)
4   write(n+1);
```

Java VM

Kleines. Programm (Einlesen (6), falls $(n < 5)$, write$(n + 1)$




```
1 int n;
2 n = readInt();
3 if(n < 5)
4   write(n+1);
```

Java VM

Kleines. Programm (Einlesen (6), falls $(n < 5)$, write$(n + 1)$


```
1 int n;
2 n = readInt();
3 if(n < 5)
4   write(n+1);
```

Java VM

Kleines. Programm (Einlesen (6), falls $(n < 5)$, write$(n + 1)$


```
1 int n;
2 n = readInt();
3 if(n < 5)
4   write(n+1);
```

Java VM

Kleines. Programm (Einlesen (6), falls $(n < 5)$, write$(n + 1)$

ALLOC 1	READ	STORE 0	LOAD 0	CONST 5	LESS
0	1	2	3	4	5

FJUMP 11	LOAD 0	CONST 1	ADD	WRITE	HALT
6	7	8	9		


```
1 int
2 n =
3 if
4   w
```

FJUMP konsumiert einen booleschen Wert und verändert (falls false) den PC auf i.

Java VM

Kleines. Programm (Einlesen (6), falls $(n < 5)$, write$(n + 1)$

ALLOC 1	READ	STORE 0	LOAD 0	CONST 5	LESS
0	1	2	3	4	5

FJUMP 11	LOAD 0	CONST 1	ADD	WRITE	HALT
6	7	8	9	10	11

6	0 n

```
1 int n;
2 n = readInt();
3 if(n < 5)
4   write(n+1);
```

Java VM

Befehlsübersicht

int Operatoren	`NEG, ADD, SUB, MUL, DIV, MOD`
boolean Operatoren	`NOT, AND, OR`
Vergleichs Operatoren	`LESS, LEQ, EQ, NEQ`
Laden von Konstanten	`CONST` *i*`, TRUE, FALSE`
Speicher Operationen	`LOAD` *i*`, STORE` *i*
Sprung Befehle	`JUMP` *i*`, FJUMP` *i*
IO (in/out) Befehle	`READ, WRITE`
Reservierung Speicher	`ALLOC` *i*
Beenden d. Programms	`HALT`

Java VM

Befehlsübersicht

int Operatoren	`NEG`
boolean Operatoren	`NOT`
Vergleichs Operatoren	`LES`
Laden von Konstanten	`CON`
Speicher Operationen	`LOAD` *i*, `STORE` *i*
Sprung Befehle	`JUMP` *i*, `FJUMP` *i*
IO (in/out) Befehle	`READ`, `WRITE`
Reservierung Speicher	`ALLOC` *i*
Beenden d. Programms	`HALT`

JUMP wird **immer ausgeführt** und konsumiert nichts vom Stack. **FJUMP** konsumiert einen Wert vom Stack und wird **nur ausgeführt, falls der Wert FALSE ist.**

Java VM

Hinweise:

- Keine Anweisung für 'GREATER'

  `greater ≡ NOT LEQ`

- Schleifen, wie if mit 'JUMP' am ENDE

  `loop ≡ condition`

  `FJUMP hinter Ende der Schleife`

  `body`

  ~~`TRUE`~~ `//ohne TRUE, das war falsch!`

  `JUMP condition`

Java VM

- Selektion (if) mit Alternative (else)

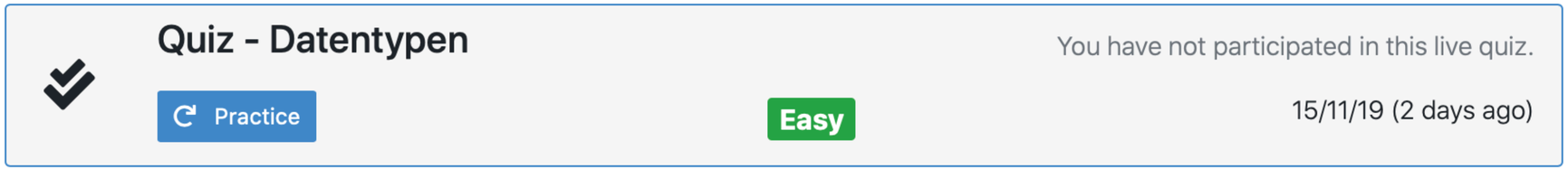
```
condition
FJUMP Beginn 'else' Teil
'true' Teil
TRUE //ohne TRUE, das war falsch!
JUMP hinter Ende 'else Teil'
'else' Teil
```

- *Zuweisung*

```
expr
STORE Adresse Variable
```

P05.01

Quiz zu Datentypen



Zuhause bearbeiten, Besprechung/Fragen nächste Woche!

P05.02

Recap: Rekursion

- Rekursionsanfang
- Rekursionsschritt
- Abbruchbedingung

Tail-Rekursion funktioniert läuft in der Realität ohne Stack! (Compiler optimiert)

Bsp: Fakultät

$$f(n) = \begin{cases} n = 1 & 1 \\ n > 1 & n \cdot f(n-1) \end{cases}$$

P05.02

Binomialkoeffizient (Rekursion)

$$\binom{n}{k} = \begin{cases} n == k & 1 \\ k == 0 & 1 \\ n > k & \binom{n-1}{k-1} + \binom{n-1}{k} \end{cases} \qquad \forall n, q : n \geq k \geq 0$$

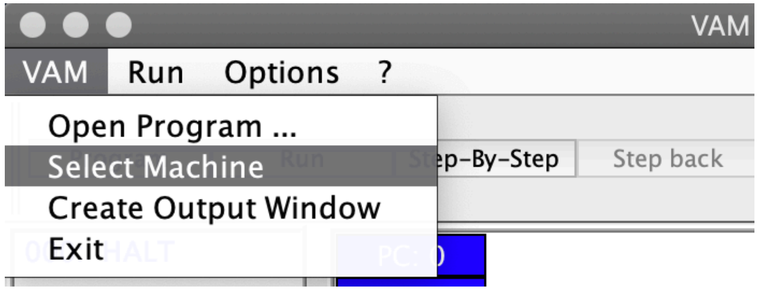
```
static int bino(int n, int k)
```

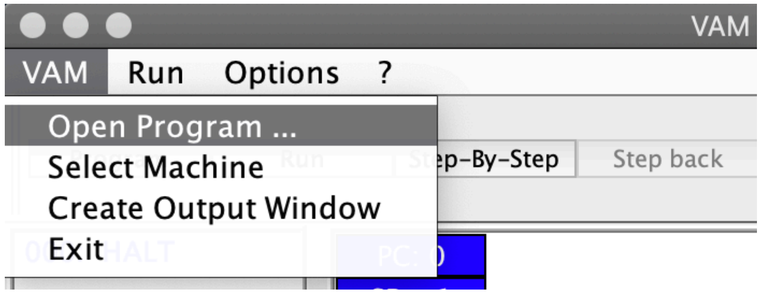
Visualisierung↗

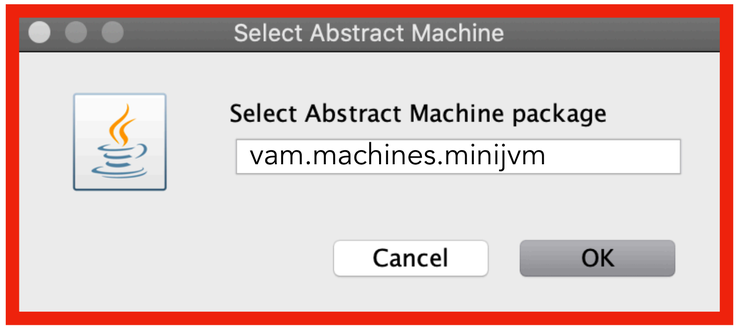
P05.04

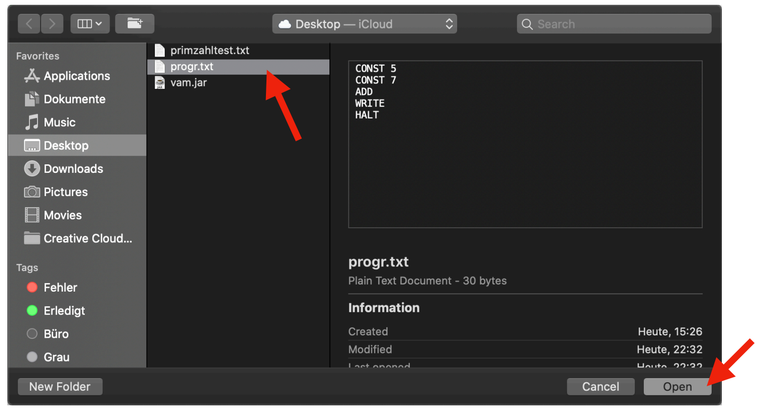
Java VM, Primzahlen

- Übersicht der Befehle
- Programm zum Testen









P05.04

Java VM, Primzahlen

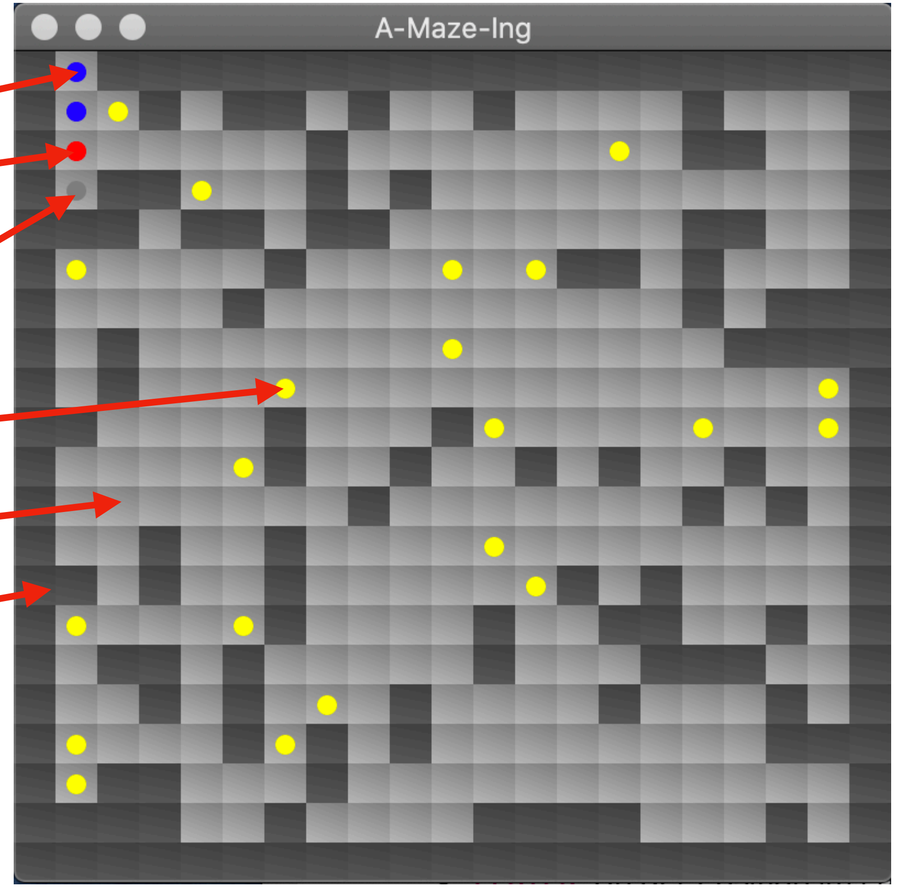
```
1  int nummer,teiler;
2  boolean prim;
3  nummer = 5;
4  teiler = 2;
5  prim = true;
6  while(teiler*teiler <=
7    nummer){
8    if(nummer%teiler == 0){
9      prim = false;
10   }
11   teiler = teiler + 1;
12 }
```

```
13 if(nummer <= 1) {
14   prim = false;
15 }
16 write(prim);
17
18
19
20
21
22
23
24
```

P05.03

Pinguinlabyrinth

- OLD_PATH_ACTIVE
- PLAYER
- OLD_PATH_DONE
- PENGUIN
- FREE
- WALL



P05.03

Pinguinlabyrinth Idee: Methode gibt zurück, wie viele Pinguine auf dem besuchten Feld gefunden wurden.

```
if maxDistance < 0 then return 0 //Ende
if nicht mehr im Labyrinth then return 0 //Ende
if Feld bereits besucht then return 0 //Ende
if Feld ist Wand then return 0 //Ende
if Feld ist Pinguin then Speicher neuen Pinguin,
  erhöhe maxDistance um 3
Feld betreten, draw, Feld auf besucht (aktiv)
Speichern von neuen Pinguinen aus walk //rekursiv
Feld auf besucht (fertig), return Anzahl Pinguine
```