4. Effizienz von Algorithmen

Effizienz von Algorithmen, Random Access Machine Modell, Funktionenwachstum, Asymptotik [Cormen et al, Kap. 2.2,3,4.2-4.4 | Ottman/Widmayer, Kap. 1.1]

Effizienz von Algorithmen

Ziele

- Laufzeitverhalten eines Algorithmus maschinenunabhängig quantifizieren.
- Effizienz von Algorithmen vergleichen.
- Abhängigkeit von der Eingabegrösse verstehen.

Technologiemodell

Random Access Machine (RAM)

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.

Technologiemodell

Random Access Machine (RAM)

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit.

Technologiemodell

Random Access Machine (RAM)

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit.
- Elementare Operationen: Rechenoperation ($+$,$-$,$\cdot$,...) , Vergleichsoperationen, Zuweisung / Kopieroperation, Flusskontrolle (Sprünge)

Technologiemodell

Random Access Machine (RAM)

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit.
- Elementare Operationen: Rechenoperation ($+$,$-$,$\cdot$,...) , Vergleichsoperationen, Zuweisung / Kopieroperation, Flusskontrolle (Sprünge)
- Einheitskostenmodell: elementare Operation hat Kosten 1.

Technologiemodell

Random Access Machine (RAM)

- Ausführungsmodell: Instruktionen werden der Reihe nach (auf einem Prozessorkern) ausgeführt.
- Speichermodell: Konstante Zugriffszeit.
- Elementare Operationen: Rechenoperation ($+$,$-$,$\cdot$,...) , Vergleichsoperationen, Zuweisung / Kopieroperation, Flusskontrolle (Sprünge)
- Einheitskostenmodell: elementare Operation hat Kosten 1.
- Datentypen: Fundamentaltypen wie grössenbeschränkte Ganzzahl oder Fliesskommazahl.

Asymptotisches Verhalten

Genaue Laufzeit lässt sich selbst für kleine Eingabedaten kaum voraussagen.

- Betrachten das asymptotische Verhalten eines Algorithmus.
- Ignorieren alle konstanten Faktoren.

Beispiel

Eine Operation mit Kosten 20 ist genauso gut wie eine mit Kosten 1. Lineares Wachstum mit Steigung 5 ist genauso gut wie lineares Wachstum mit Steigung 1.

4.1 Funktionenwachstum

$\mathcal{O}$, Θ, Ω [Cormen et al, Kap. 3; Ottman/Widmayer, Kap. 1.1]

Oberflächlich

Verwende die asymptotische Notation zur Kennzeichnung der Laufzeit von Algorithmen

Wir schreiben $\Theta(n^2)$ und meinen, dass der Algorithmus sich für grosse n wie n^2 verhält: verdoppelt sich die Problemgrösse, so vervierfacht sich die Laufzeit.

Genauer: Asymptotische obere Schranke

Gegeben: Funktion $f : \mathbb{N} \to \mathbb{R}$.

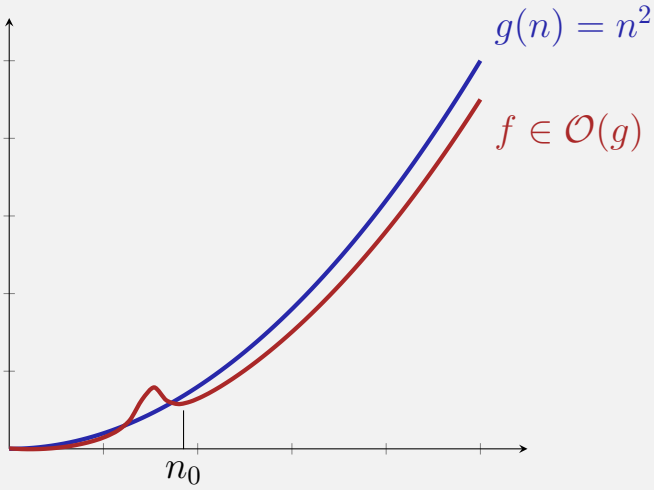
Definition:

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \; \forall n \geq n_0\}$$

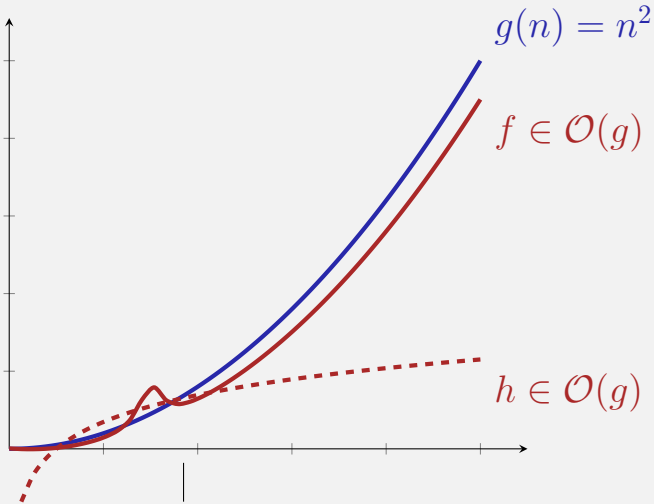
Schreibweise:

$$\mathcal{O}(g(n)) := \mathcal{O}(g(\cdot)) = \mathcal{O}(g).$$

Anschauung



Anschauung



Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} |\ \ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$		
$2n$		
$n^2 + 100n$		
$n + \sqrt{n}$		

Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} |\ \ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$		
$n^2 + 100n$		
$n + \sqrt{n}$		

Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} |\ \ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$		
$n + \sqrt{n}$		

Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} |\ \ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$		

Beispiele

$$\mathcal{O}(g) = \{f : \mathbb{N} \to \mathbb{R} |\ \ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n)\ \forall n \geq n_0\}$$

$f(n)$	$f \in \mathcal{O}(?)$	Beispiel
$3n + 4$	$\mathcal{O}(n)$	$c = 4, n_0 = 4$
$2n$	$\mathcal{O}(n)$	$c = 2, n_0 = 0$
$n^2 + 100n$	$\mathcal{O}(n^2)$	$c = 2, n_0 = 100$
$n + \sqrt{n}$	$\mathcal{O}(n)$	$c = 2, n_0 = 1$

Eigenschaft

$$f_1 \in \mathcal{O}(g), f_2 \in \mathcal{O}(g) \Rightarrow f_1 + f_2 \in \mathcal{O}(g)$$

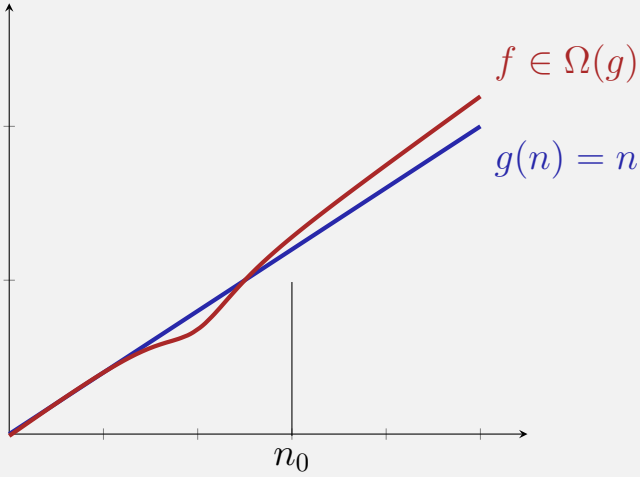
Umkehrung: Asymptotische untere Schranke

Gegeben: Funktion $f : \mathbb{N} \to \mathbb{R}$.

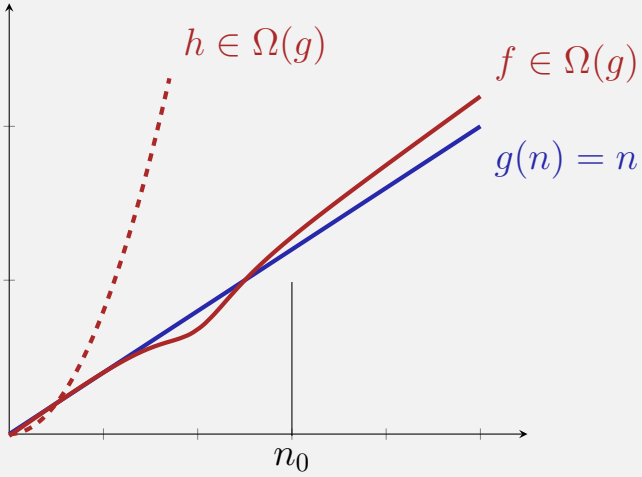
Definition:

$$\Omega(g) = \{f : \mathbb{N} \to \mathbb{R} | \\ \exists c > 0, n_0 \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n) \; \forall n \geq n_0\}$$

Beispiel



Beispiel



Asymptotisch scharfe Schranke

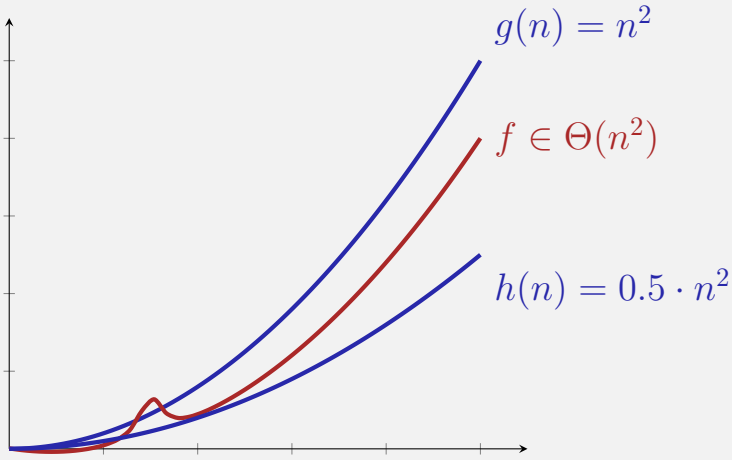
Gegeben Funktion $f : \mathbb{N} \to \mathbb{R}$.

Definition:

$$\Theta(g) := \Omega(g) \cap \mathcal{O}(g).$$

Einfache, geschlossene Form: Übung.

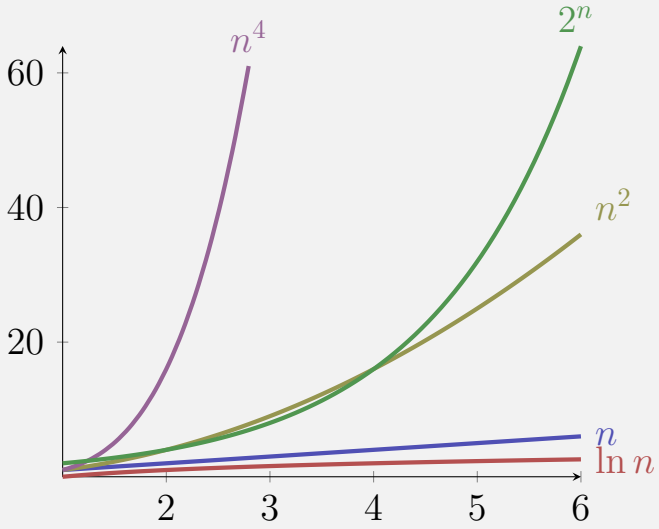
Beispiel



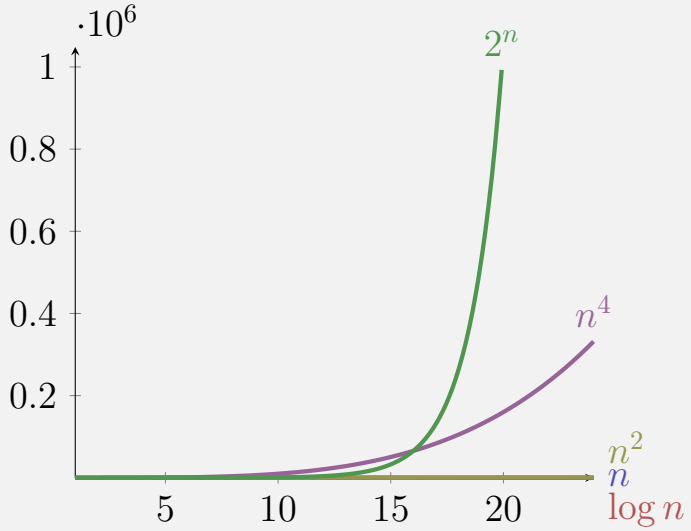
Wachstumsbezeichnungen

$\mathcal{O}(1)$	beschränkt	Array-Zugriff
$\mathcal{O}(\log \log n)$	doppelt logarithmisch	Binäre sortierte Suche interpoliert
$\mathcal{O}(\log n)$	logarithmisch	Binäre sortierte Suche
$\mathcal{O}(\sqrt{n})$	wie die Wurzelfunktion	Primzahltest (naiv)
$\mathcal{O}(n)$	linear	Unsortierte naive Suche
$\mathcal{O}(n \log n)$	superlinear / loglinear	Gute Sortieralgorithmen
$\mathcal{O}(n^2)$	quadratisch	Einfache Sortieralgorithmen
$\mathcal{O}(n^c)$	polynomial	Matrixmultiplikation
$\mathcal{O}(2^n)$	exponentiell	Travelling Salesman Dynamic Programming
$\mathcal{O}(n!)$	faktoriell	Travelling Salesman naiv

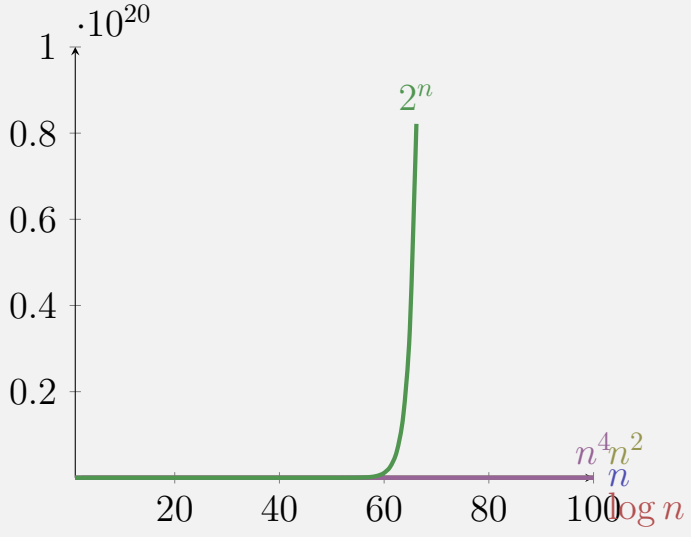
Kleine n



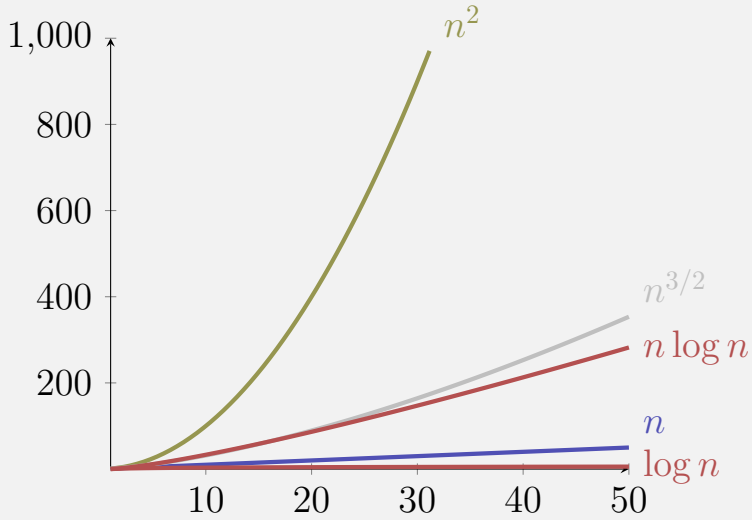
Grössere n



“Grosse” n



Logarithmen!



Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$				
n	$1\mu s$				
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$				
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$				
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$				
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$				
n^2	$1\mu s$				
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
n^2	$1\mu s$				
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$				

Zeitbedarf

Annahme: 1 Operation = $1\mu s$.

Problemgrösse	1	100	10000	10^6	10^9
$\log_2 n$	$1\mu s$	$7\mu s$	$13\mu s$	$20\mu s$	$30\mu s$
n	$1\mu s$	$100\mu s$	$1/100s$	$1s$	17 Minuten
$n \log_2 n$	$1\mu s$	$700\mu s$	$13/100\mu s$	$20s$	8.5 Stunden
n^2	$1\mu s$	$1/100s$	1.7 Minuten	11.5 Tage	317 Jahrhund.
2^n	$1\mu s$	10^{14} Jahrh.	$\approx \infty$	$\approx \infty$	$\approx \infty$

Eine gute Strategie?

... dann kaufe ich mir eben eine neue Maschine!

Eine gute Strategie?

... dann kaufe ich mir eben eine neue Maschine! Wenn ich heute ein Problem der Grösse n lösen kann, dann kann ich mit einer 10 oder 100 mal so schnellen Maschine...

Komplexität	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$		
n		
n^2		
2^n		

Eine gute Strategie?

... dann kaufe ich mir eben eine neue Maschine! Wenn ich heute ein Problem der Grösse n lösen kann, dann kann ich mit einer 10 oder 100 mal so schnellen Maschine...

Komplexität	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$	$n \to n^{10}$	$n \to n^{100}$
n		
n^2		
2^n		

Eine gute Strategie?

... dann kaufe ich mir eben eine neue Maschine! Wenn ich heute ein Problem der Grösse n lösen kann, dann kann ich mit einer 10 oder 100 mal so schnellen Maschine...

Komplexität	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$	$n \to n^{10}$	$n \to n^{100}$
n	$n \to 10 \cdot n$	$n \to 100 \cdot n$
n^2		
2^n		

Eine gute Strategie?

... dann kaufe ich mir eben eine neue Maschine! Wenn ich heute ein Problem der Grösse n lösen kann, dann kann ich mit einer 10 oder 100 mal so schnellen Maschine...

Komplexität	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$	$n \to n^{10}$	$n \to n^{100}$
n	$n \to 10 \cdot n$	$n \to 100 \cdot n$
n^2	$n \to 3.16 \cdot n$	$n \to 10 \cdot n$
2^n		

Eine gute Strategie?

... dann kaufe ich mir eben eine neue Maschine! Wenn ich heute ein Problem der Grösse n lösen kann, dann kann ich mit einer 10 oder 100 mal so schnellen Maschine...

Komplexität	(speed $\times 10$)	(speed $\times 100$)
$\log_2 n$	$n \to n^{10}$	$n \to n^{100}$
n	$n \to 10 \cdot n$	$n \to 100 \cdot n$
n^2	$n \to 3.16 \cdot n$	$n \to 10 \cdot n$
2^n	$n \to n + 3.32$	$n \to n + 6.64$

Beispiele

Beispiele

- $n \in \mathcal{O}(n^2)$

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau: $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich: Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau: $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich: Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch:

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ ist korrekt

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ ist korrekt

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ ist korrekt
- $\Theta(n) \subseteq \Theta(n^2)$

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ ist korrekt
- $\Theta(n) \subseteq \Theta(n^2)$ ist falsch:

Beispiele

- $n \in \mathcal{O}(n^2)$ korrekt, aber ungenau:
  $n \in \mathcal{O}(n)$ und sogar $n \in \Theta(n)$.
- $3n^2 \in \mathcal{O}(2n^2)$ korrekt, aber unüblich:
  Konstanten weglasssen: $3n^2 \in \mathcal{O}(n^2)$.
- $2n^2 \in \mathcal{O}(n)$ ist falsch: $\frac{2n^2}{cn} = \frac{2}{c}n \underset{n\to\infty}{\to} \infty$!
- $\mathcal{O}(n) \subseteq \mathcal{O}(n^2)$ ist korrekt
- $\Theta(n) \subseteq \Theta(n^2)$ ist falsch: $n \notin \Omega(n^2) \supset \Theta(n^2)$

5. Java - Sprachkonstrukte I

Namen und Bezeichner, Variablen, Zuweisungen, Konstanten, Datentypen, Operationen, Auswerten von Ausdrücken, Konversionen, Kontrollfluss, Klassen, statische Methoden

Namen und Bezeichner

Ein Programm (also Klasse) braucht einen Namen

```
public class SudokuSolver { ...
```

- Konvention für Klassennamen: *CamelCase* benützen $\rightarrow$ Wörter sind zusammengeschrieben mit jeweils einem Grossbuchstaben

Erlaubte Namen für “Entitäten” im Programm:

- Namen beginnen mit einem *Buchstaben* oder _ oder *$*
- Danach Folge von *Buchstaben*, *Zahlen* oder _ oder *$*

Namen - was ist erlaubt

_myName

me@home

TheCure

strictfp

49ers

__AN$WE4_1S_42__

side-swipe

$bling$

Ph.D’s

Namen - was ist erlaubt

`_myName`	`me@home`
`TheCure`	`strictfp`
`49ers`	`__AN$WE4_1S_42__`
`side-swipe`	`$bling$`
`Ph.D's`	

Namen - was ist erlaubt

`_myName`	`me@home`
`TheCure`	`strictfp` ?! `49ers`
`__AN$WE4_1S_42__`	`side-swipe`
`$bling$`	`Ph.D's`

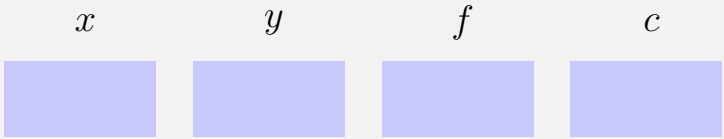
Schlüsselwörter

Folgende Wörter werden von der Sprache bereits benützt und können deshalb nicht als Namen benützt werden:

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

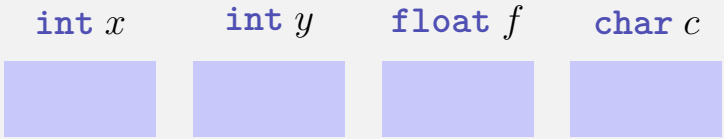
Variablen und Konstanten

- *Behälter* für einen Wert.
- Haben einen *Datentyp* und einen *Namen*
- Der Datentyp bestimmt, welche Art von Werten in der Variable erlaubt sind



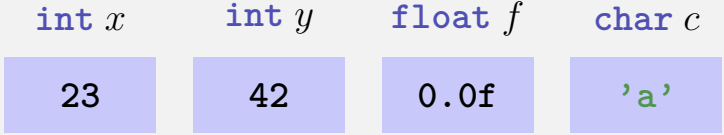
Variablen und Konstanten

- *Behälter* für einen Wert.
- Haben einen *Datentyp* und einen *Namen*
- Der Datentyp bestimmt, welche Art von Werten in der Variable erlaubt sind



Variablen und Konstanten

- *Behälter* für einen Wert.
- Haben einen *Datentyp* und einen *Namen*
- Der Datentyp bestimmt, welche Art von Werten in der Variable erlaubt sind



Variablen und Konstanten

- *Behälter* für einen Wert.
- Haben einen *Datentyp* und einen *Namen*
- Der Datentyp bestimmt, welche Art von Werten in der Variable erlaubt sind

`int` x	`int` y	`float` f	`char` c
`23`	`42`	`0.0f`	`'a'`

Deklaration in Java:

```
int x = 23, y = 42;
float f;
char c = 'a';
```

Variablen und Konstanten

- *Behälter* für einen Wert.
- Haben einen *Datentyp* und einen *Namen*
- Der Datentyp bestimmt, welche Art von Werten in der Variable erlaubt sind

`int` x	`int` y	`float` f	`char` c
`23`	`42`	`0.0f`	`'a'`

Deklaration in Java:

```
int x = 23, y = 42;
float f;
char c = 'a';
```

Initialisierung

Konstanten

- Schlüsselwort `final`
- Der Wert der Variable kann genau einmal gesetzt werden.

Beispiel

```
final int maxSize = 100;
```

Tip: Benützen Sie `final` immer, es sei denn der Wert muss tatsächlich veränderlich sein.

Standardtypen

Datentyp	Definition	Wertebereich	Initialwert
`byte`	8-bit Ganzzahl	$-128, \ldots, 127$	`0`
`short`	16-bit Ganzzahl	$-32'768, \ldots, 32'767$	`0`
`int`	32-bit Ganzzahl	$-2^{31}, \ldots, 2^{31} - 1$	`0`
`long`	64-bit Ganzzahl	$-2^{63}, \ldots, 2^{63} - 1$	`0L`
`float`	32-bit Fliesskommazahl	$\pm 1.4E^{-45}, \ldots, \pm 3.4E^{+38}$	`0.0f`
`double`	64-bit Fliesskommazahl	$\pm 4.9E^{-324}, \ldots, \pm 1.7E^{+308}$	`0.0d`
`boolean`	Wahrheitswert	`true`, `false`	`false`
`char`	Unicode-16 Zeichen	`'\u0000'`,...,`'a'`,`'b'`,...,`'\uFFFF'`	`'\u0000'`
`String`	Zeichenkette	∞	`null`

Typen und Speicherbelegung

Zur Erinnerung: Speicherzellen enthalten 1 Byte $=$ 8 bit

Typen und Speicherbelegung

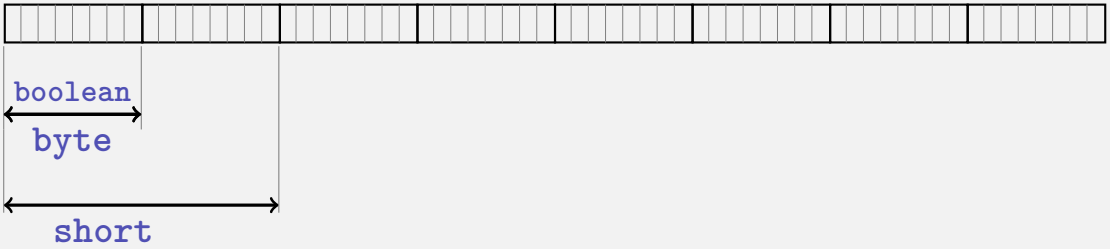
Zur Erinnerung: Speicherzellen enthalten 1 Byte $=$ 8 bit

Typen und Speicherbelegung

Zur Erinnerung: Speicherzellen enthalten 1 Byte $=$ 8 bit

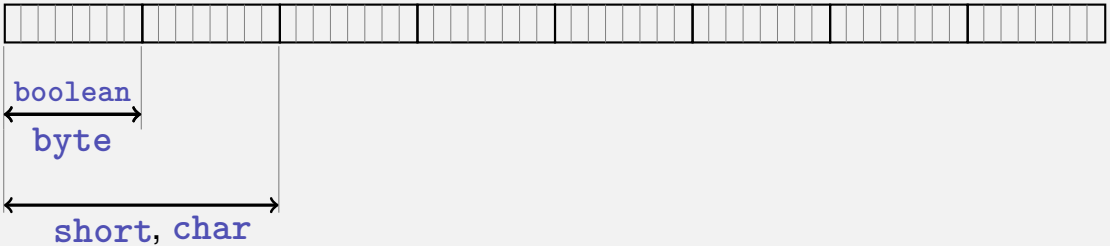
Typen und Speicherbelegung

Zur Erinnerung: Speicherzellen enthalten 1 Byte $=$ 8 bit



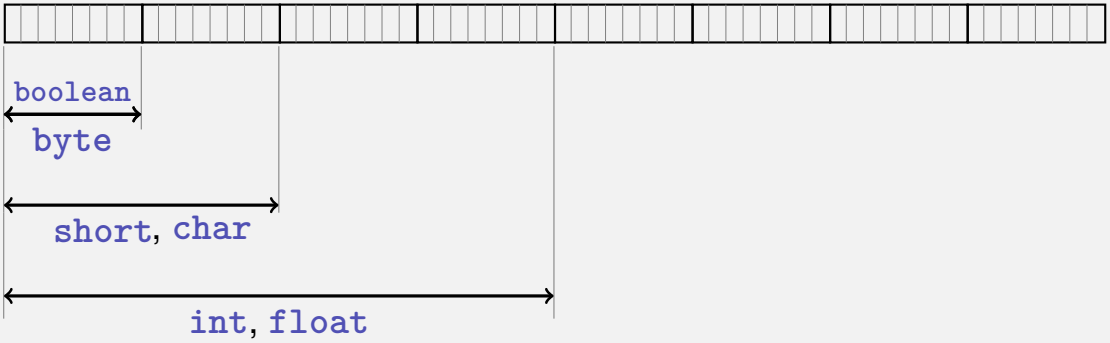
Typen und Speicherbelegung

Zur Erinnerung: Speicherzellen enthalten 1 Byte $=$ 8 bit



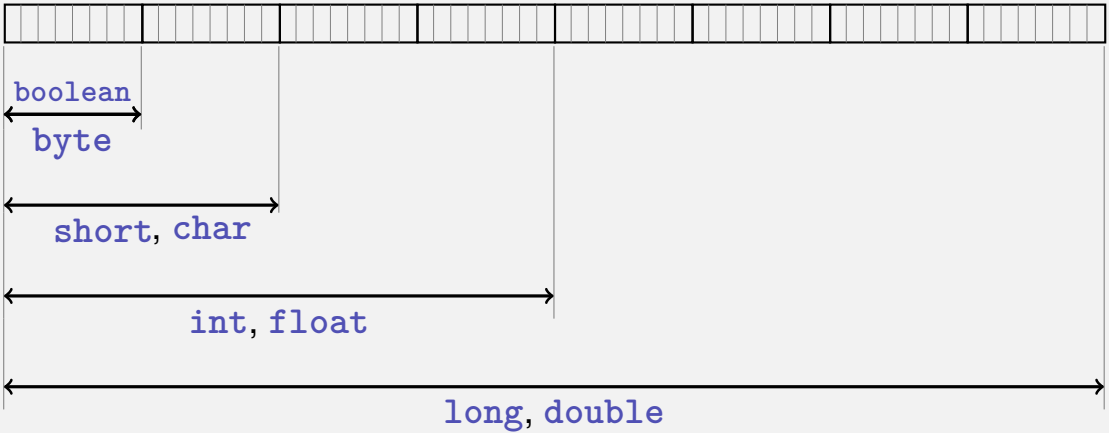
Typen und Speicherbelegung

Zur Erinnerung: Speicherzellen enthalten 1 Byte $=$ 8 bit



Typen und Speicherbelegung

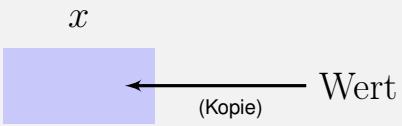
Zur Erinnerung: Speicherzellen enthalten 1 Byte = 8 bit



Wertzuweisungen

Kopiert einen Wert in die Variable x

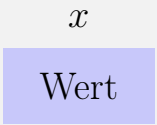
- In Pseudocode: $x \leftarrow \text{Wert}$
- In Java: $x = \text{Wert}$



Wertzuweisungen

Kopiert einen Wert in die Variable x

- In Pseudocode: $x \leftarrow \text{Wert}$
- In Java: $x = \text{Wert}$

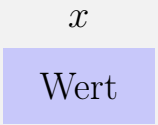


Wert

Wertzuweisungen

Kopiert einen Wert in die Variable x

- In Pseudocode: $x \leftarrow \text{Wert}$
- In Java: $x = \text{Wert}$



Wert

"=" ist der Zuweisungsoperator *und nicht ein Vergleich!*

`int y = 42` ist also Deklaration + Wertzuweisung in einem.

Wertzuweisungen

Beispiele

```
int a = 3;
double b;

b = 3.141;

int c = a = 0;

String name = "Inf";
```

Wertzuweisungen

Beispiele

```
int a = 3;
double b;

b = 3.141;

int c = a = 0;

String name = "Inf";
```

Eine *verschachtelte* Zuweisung: Der Ausdruck `a = 0` speichert den Wert `0` in Variable `a`. *und gibt den Wert danach zurück*

Arithmetische Binäre Operatoren

Infix Notation: $x \text{ op } y$ mit folgenden Operatoren

op: $+ \quad - \quad * \quad / \quad \%$

Modulo

Arithmetische Binäre Operatoren

Infix Notation: $x \text{ op } y$ mit folgenden Operatoren

op: `+` `-` `*` `/` `%`

Modulo

Division x / y: Ganzzahldivision falls x und y Ganzzahlen sind.

Arithmetische Binäre Operatoren

Infix Notation: $x \text{ op } y$ mit folgenden Operatoren

op: `+` `-` `*` `/` `%`

Division x `/` y: Ganzzahldivision falls x und y Ganzzahlen sind.

Beispiele

Ganzzahldivision und Modulo

- `5 / 3` ergibt 1 `-5 / 3` ergibt -1
- `5 % 3` ergibt 2 `-5 % 3` ergibt -2

Arithmetische Zuweisung

```
x = x + y
```

```
x += y
```

Analog für $-$, $*$, `/`, `%`

Arithmetische Zuweisung

```
x = x + y
```

```
x += y
```

Beispiele:

```
x -= 3;       // x = x - 3
name += "x"   // name = name + "x"
num *= 2;     // num = num * 2
```

Analog für $-$, $*$, **/**, **%**

Arithmetische Unäre Operatoren

Prefix Notation: + x oder $-$ x

Arithmetische Unäre Operatoren

Prefix Notation: + x oder $-\ x$

Unäre Operatoren binden stärker als binäre.

Arithmetische Unäre Operatoren

Prefix Notation: + x oder $-$ x

Unäre Operatoren binden stärker als binäre.

Beispiele

Angenommen `x` ist 3

- 2 $*$ $-$`x` ergibt -6
- $-$`x` $-$ `+1` ergibt -4

Inkrement/Dekrement Operatoren

Inkrement Operatoren `++x` und `x++` haben den gleichen *Effekt*:

$$x \leftarrow x + 1.$$

Aber unterschiedliche Rückgabewerte:

- *Präfixoperator* `++x` gibt *neuen* Wert zurück:
  `a = ++x;` $\iff$ `x = x + 1; a = x;`
- *Postfixoperator* `x++` gibt den *alten* Wert zurück:
  `a = x++;` $\iff$ `temp = x; x = x + 1; a= temp;`

Analog für `x--` und `--x`.

Inkrement/Dekrement Operatoren

Beispiele

Angenommen `x` ist initial 2

- `y = ++x * 3`
- `y = x++ * 3`

Inkrement/Dekrement Operatoren

Beispiele

Angenommen `x` ist initial 2

- `y = ++x * 3` ergibt: `x` ist 3 und `y` ist 9
- `y = x++ * 3`

Inkrement/Dekrement Operatoren

Beispiele

Angenommen `x` ist initial 2

- `y = ++x * 3` ergibt: `x` ist 3 und `y` ist 9
- `y = x++ * 3` ergibt: `x` ist 3 und `y` ist 6

Ausdrücke (Expressions)

- repräsentieren *Berechnungen*
- sind entweder *primär*
- oder *zusammengesetzt* . . .
- . . . aus anderen Ausdrücken, mit Hilfe von *Operatoren*
- sind statisch typisiert

Analogie: Baukasten

Ausdrücke (Expressions)

Beispiele

primär: “`-4.1d`” oder “`x`” oder `"Hi"`

zusammengesetzt: “`x + y`” oder “`f * 2.1f`”

Der Typ von “`12 * 2.1f`” ist `float`

Celsius zu Fahrenheit

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        System.out.print("Celsius: ");
        Scanner input = new Scanner(System.in);
        int celsius = input.nextInt();
        float fahrenheit = 9 * celsius / 5 + 32;
        System.out.println("Fahrenheit: " + fahrenheit);
    }
}
```

Celsius zu Fahrenheit

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        System.out.print("Celsius: ");
        Scanner input = new Scanner(System.in);
        int celsius = input.nextInt();
        float fahrenheit = 9 * celsius / 5 + 32;
        System.out.println("Fahrenheit: " + fahrenheit);
    }
}
```

Beispiel: 15° Celsius sind 59° Fahrenheit

Celsius zu Fahrenheit - Analyse

```
9 * celsius / 5 + 32
```

- Arithmetischer Ausdruck,

Celsius zu Fahrenheit - Analyse

```
9 * celsius / 5 + 32
```

- Arithmetischer Ausdruck,
- drei Literale, eine Variable, drei Operatorsymbole

Celsius zu Fahrenheit - Analyse

```
9 * celsius / 5 + 32
```

- Arithmetischer Ausdruck,
- drei Literale, eine Variable, drei Operatorsymbole

Celsius zu Fahrenheit - Analyse

```
9 * celsius / 5 + 32
```

- Arithmetischer Ausdruck,
- drei Literale, eine Variable, drei Operatorsymbole

Celsius zu Fahrenheit - Analyse

```
9 * celsius / 5 + 32
```

- Arithmetischer Ausdruck,
- drei Literale, eine Variable, drei Operatorsymbole

Wie ist der Ausdruck geklammert?

Regel 1: Präzedenz

Multiplikative Operatoren (`*`, `/`, `%`) haben höhere Präzedenz ("binden stärker") als additive Operatoren (`+`, `-`)

Regel 1: Präzedenz

Multiplikative Operatoren (`*`, `/`, `%`) haben höhere Präzedenz ("binden stärker") als additive Operatoren (`+`, `-`)

Beispiel

```
9 * celsius / 5 + 32
```

bedeutet

```
(9 * celsius / 5) + 32
```

Regel 2: Assoziativität

Arithmetische Operatoren (`*`, `/`, `%`, `+`, `-`) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts

Regel 2: Assoziativität

Arithmetische Operatoren (`*`, `/`, `%`, `+`, `-`) sind linksassoziativ: bei gleicher Präzedenz erfolgt Auswertung von links nach rechts

Beispiel

```
9 * celsius / 5 + 32
```

bedeutet

```
((9 * celsius) / 5) + 32
```

Regel 3: Stelligkeit

Unäre Operatoren $+$, $-$ vor binären $+$, $-$.

Regel 3: Stelligkeit

Unäre Operatoren +, − vor binären +, −.

Beispiel

```
9 * celsius / + 5 + 32
```

bedeutet

```
9 * celsius / (+5) + 32
```

Klammerung

Jeder Ausdruck kann mit Hilfe der

- Assoziativitäten
- Präzedenzen
- Stelligkeiten

der beteiligten Operatoren eindeutig geklammert werden.

Ausdrucksbäume

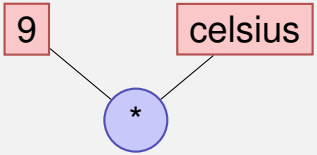
Klammerung ergibt Ausdrucksbaum

```
9 * celsius / 5 + 32
```

Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

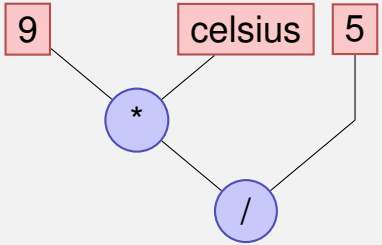
```
(9 * celsius) / 5 + 32
```



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

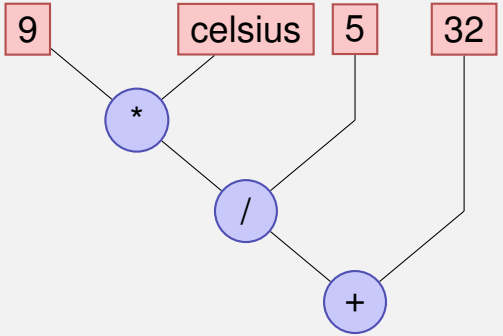
```
((9 * celsius) / 5) + 32
```



Ausdrucksbäume

Klammerung ergibt Ausdrucksbaum

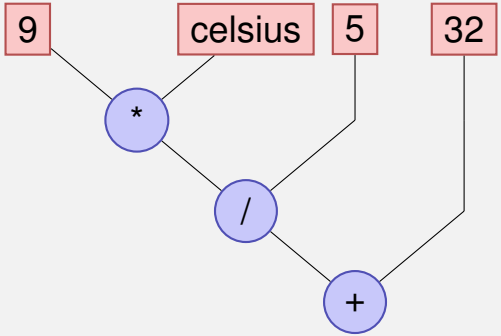
```
(((9 * celsius) / 5) + 32)
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

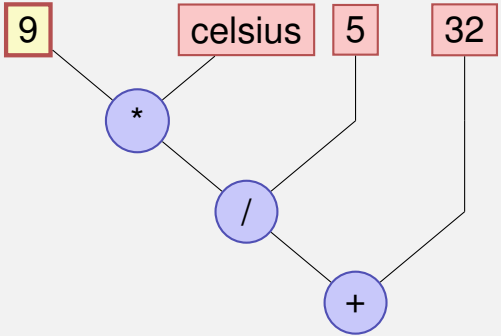
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

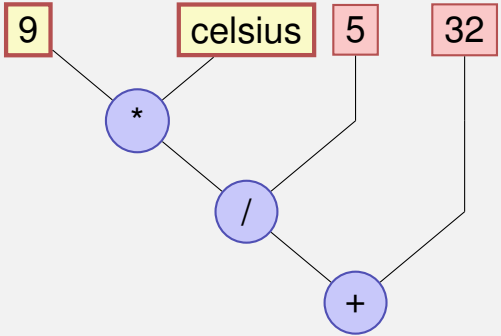
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

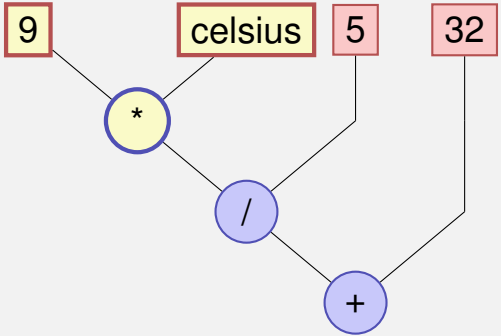
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

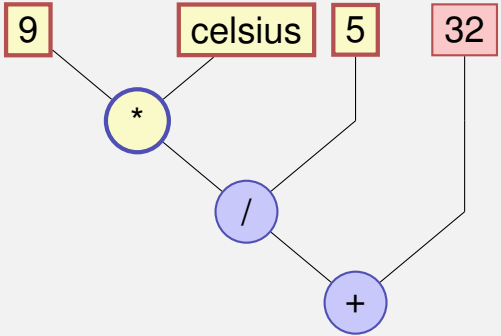
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

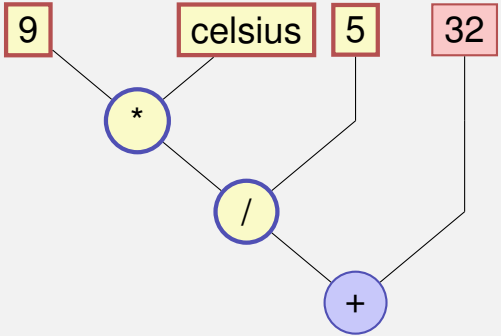
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

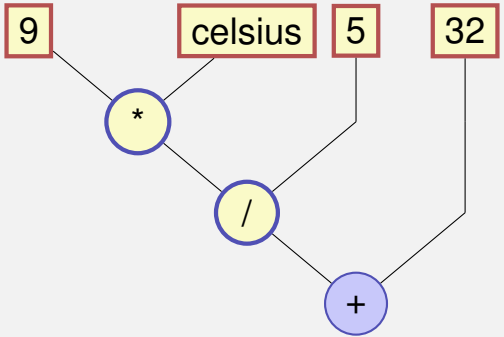
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

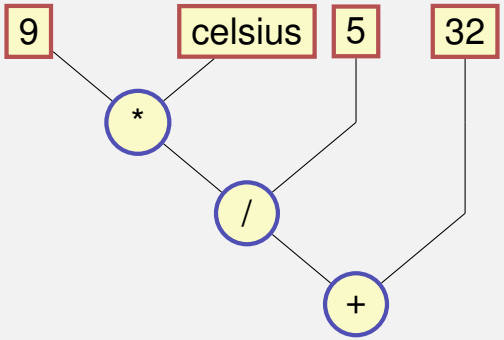
```
9 * celsius / 5 + 32
```



Auswertungsreihenfolge

"Von oben nach unten" im Ausdrucksbaum

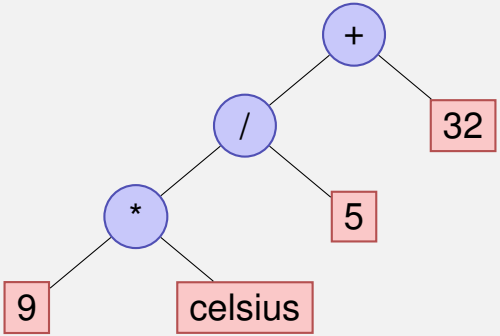
```
9 * celsius / 5 + 32
```



Ausdrucksbäume – Notation

Üblichere Notation: Wurzel oben

```
9 * celsius / 5 + 32
```



Typsystem

Java hat ein *statisches* Typsystem:

- Alle Typen müssen deklariert werden
- Falls möglich wird Typisierung wird vom Compiler geprüft ...
- ... ansonsten zur Laufzeit

Vorteil eines statischen Typsystems

- *Fail-fast* Fehler im Programm werden oft schon vom Compiler gefunden
- Verständlicher Code

Typfehler

Beispiel

```
int pi_ungenau;
float pi = 3.14f;

pi_ungenau = pi;
```

Compiler Fehler:

```
./Root/Main.java:12: error: incompatible types: possible lossy conversion from float to int
       pi_ungenau = pi;
                    ^
```

Explizite Typkonvertierung

Beispiel

```
int pi_ungenau;
float pi = 3.14f;

pi_ungenau =        pi;
```

Explizite Typkonvertierung

Beispiel

```
int pi_ungenau;
float pi = 3.14f;

pi_ungenau = (int) pi;
```

Explizite Typkonvertierung mit Typecasting: (typ)

Explizite Typkonvertierung

Beispiel

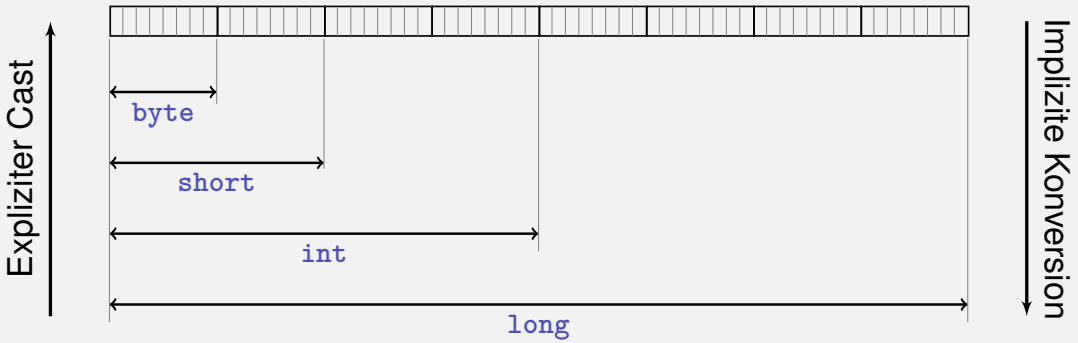
```
int pi_ungenau;
float pi = 3.14f;

pi_ungenau = (int) pi;
```

Explizite Typkonvertierung mit Typecasting: `(typ)`

- Statisch typkorrekt, Compiler happy
- Laufzeitverhalten: Je nach Situation

  Hier: Genauigkeitsverlust: $3.14 \Rightarrow 3$
- Kann das Programm zur Laufzeit zum Absturz bringen!

Typ Konvertierung - Anschaulich für Ganzzahlen



Potentieller Informationsverlust bei explizitem Cast, da weniger Speicher zur Verfügung.

Typkonversion bei binären Operationen

Bei einer binären Operation mit numerischen Operanden von verschiedenem Typ werden die Operanden nach folgenden Regeln konvertiert

- Haben beide Operanden denselben Typ, findet keine Konversion statt
- Ist einer der Operanden `double`, so wird der andere nach `double` konvertiert
- Ist einer der Operanden `float`, so wird der andere nach `float` konvertiert
- Ist einer der Operanden `long`, so wird der andere nach `long` konvertiert
- Ansonsten: Beide Operanden werden nach `int` konvertiert

Celsius zu Fahrenheit: Typanalyse

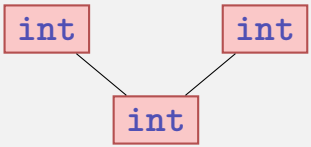
Welcher Typ hat das Ergebnis?

```
9    *  celsius / 5  +  32
```

Celsius zu Fahrenheit: Typanalyse

Welcher Typ hat das Ergebnis?

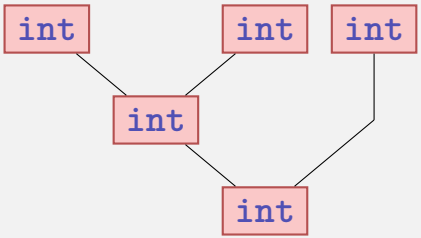
```
9    *  celsius / 5  +  32
```



Celsius zu Fahrenheit: Typanalyse

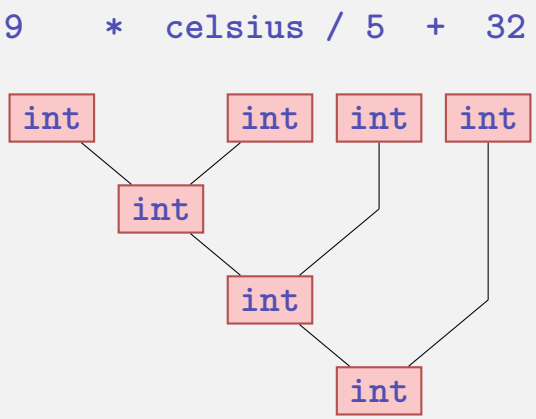
Welcher Typ hat das Ergebnis?

```
9    *  celsius / 5  +  32
```



Celsius zu Fahrenheit: Typanalyse

Welcher Typ hat das Ergebnis?



Celsius zu Fahrenheit: Typanalyse

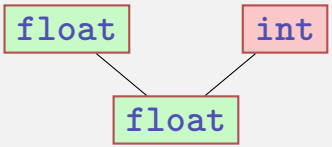
Besser: Berechnung und Ergebnis als `float`

```
9f   *  celsius / 5  +  32
```

Celsius zu Fahrenheit: Typanalyse

Besser: Berechnung und Ergebnis als `float`

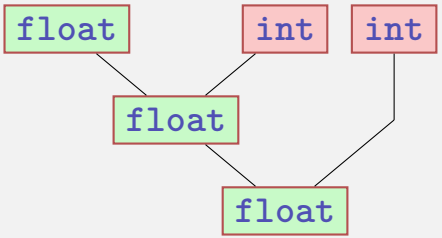
```
9f   *  celsius / 5  +  32
```



Celsius zu Fahrenheit: Typanalyse

Besser: Berechnung und Ergebnis als `float`

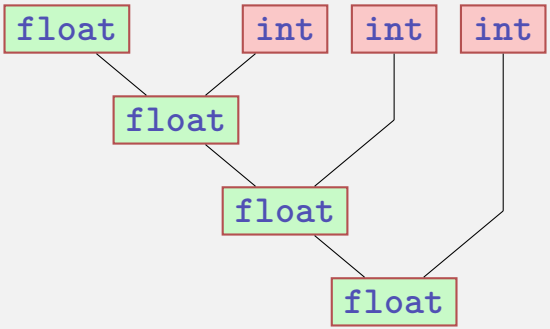
```
9f   *  celsius / 5  +  32
```



Celsius zu Fahrenheit: Typanalyse

Besser: Berechnung und Ergebnis als `float`

```
9f   *  celsius / 5  +  32
```



Anweisungen (Statements)

Eine Anweisung ist . . .

- vergleichbar mit einem Satz in der natürlichen Sprache.
- eine komplette Ausführungseinheit.
- immer mit einem *Semikolon* abgeschlossen.

Beispiel

```
f = 9f * celsius / 5 + 32 ;
```

Anweisungenarten

Gültige Anweisungen sind:

- Wertzuweisungen
- Inkrement / Dekrement Ausdrücke
- Methodenaufrufe
- Objekterzeugungs-Ausdrücke (noch nicht gesehen)

Anweisungenarten

Beispiele

```
aValue = 8933.234;                // Wertzuweisung

aValue++;                         // Inkrement

System.out.println("Awesome"); // Methodenaufruf
```

Blöcke

Ein Block ist . . .

- eine Gruppe von Anweisungen
- überall erlaubt wo Anweisungen erlaubt sind
- durch geschweiften Klammern markiert

```
{
    Anweisung
    Anweisung
    ⋮
}
```

Kontrollstrukturen

Die Abfolge der Ausführung eines Programms wird durch Anweisungen gesteuert.

Im einfachsten Fall:

Sequenz: Lineare Abfolge von Anweisungen:

Anweisung
Anweisung

Kontrollstrukturen - Sequenz

Beispiel

```
int fahrenheit = 9f * celsius / 5 + 32;
System.out.println("Fahrenheit: " + fahrenheit);
```

Kontrollstrukturen - If Anweisung

optional

```
if ( Bedingung )
    Anweisung
else
    Anweisung
```

Bedingung: Boolsche Ausdrücke

- Ein Ausdruck der zu `true` oder `false` ausgewertet wird.
- Verwendet boolschen Operatoren in zusammengesetzten Ausdrücken.

Beispiele

```
scanner.hasNext() // Hat es noch ein Zeichen?

x > 3 || y > 3    // Ist x oder y groesser 3?
```

Vergleichsoperatoren

Infix Notation: $x \text{ op } y$ mit folgenden Operatoren

op: `<` `>` `<=` `>=` `==` `!=`

Nicht zu verwechseln mit Zuweisungsoperator “`=`”

- Angewandt auf zwei vergleichbare Operanden
- Resultat ist vom Typ `boolean`
- `<`, `>`, `<=` und `>=` binden stärker als `==` und `!=`

Vergleichsoperatoren

Beispiele

```
x >= 5      // ergibt true, falls x groesser oder gleich
            // 5 ist, ansonsten false

a < b  ==  b > a      // Kommutativgesetz fuer Vergleiche
```

Boolsche Binäre Operatoren

Infix Notation: $x \operatorname{op} y$ mit folgenden Operatoren

- `&&` : Konjunktion (logisches “und”)
- `||` : Disjunktion (logisches “oder”)

Präzedenz:

- `&&` vor `||`
- Beide binden schwächer als Vergleichsoperatoren

Shortcut Evaluation

Wie wird dieser Ausdruck ausgewertet? ...

```
d > 0  &&  n/d > 1
```

Shortcut Evaluation

Wie wird dieser Ausdruck ausgewertet? ...

```
d > 0 && n/d > 1
```

... im Fall von `d == 0` ?

Shortcut Evaluation

Annahme: `d == 0`

Problem: Die Auswertung des rechten Operandes würde zu einer Division durch 0 führen.

$\Rightarrow$ *Solche Programmfehler führen zum Abbruch der Auswertung!*

Shortcut Evaluation $\Rightarrow$ Lazy Evaluation

Annahme: `d == 0`

Lösung: Faule Auswertung! Nur Auswerten was wirklich nötig ist.

```
d > 0 && irgend_etwas
```

Shortcut Evaluation $\Rightarrow$ Lazy Evaluation

Annahme: `d == 0`

Lösung: Faule Auswertung! Nur Auswerten was wirklich nötig ist.

```
d > 0 && irgend_etwas
```

$\Rightarrow$

```
false && irgend_etwas
```

Shortcut Evaluation $\Rightarrow$ Lazy Evaluation

Annahme: `d == 0`

Lösung: Faule Auswertung! Nur Auswerten was wirklich nötig ist.

```
d > 0 && irgend_etwas
```

$\Rightarrow$

```
false && irgend_etwas
```

$\Rightarrow$

```
false
```

Shortcut Evaluation $\Rightarrow$ Lazy Evaluation

Annahme: `d == 0`

Lösung: Faule Auswertung! Nur Auswerten was wirklich nötig ist.

```
d > 0 && irgend_etwas
```

$\Rightarrow$

```
false && irgend_etwas
```

$\Rightarrow$

```
false
```

Shortcut-Evaluation: Berechnung des Ausdrucks abbrechen sobald Ergebnis feststeht.

Boolscher unärer Operator: Negation

Prefix Notation: ! b

Die Negation binden stärker als alle anderen boolschen Operatoren, inklusive Vergleichsoperatoren

Boolscher unärer Operator: Negation

Beispiel

```
! d > 0  || n/d == 0
```

Ohne Klammerung: Compiler Fehler

```
./Root/src/Main.java:11: error: bad operand type int for unary operator '!'
        boolean b = !d > 0 || n/d == 0;
                    ^
```

Boolscher unärer Operator: Negation

Beispiel

```
!(d > 0) || n/d == 0
```

Kontrollstrukturen - Verwendung von Blöcken

In Kontrollstrukturen sollten immer Blöcke verwendet werden.[5]

```
if ( Bedingung ) {
    Anweisung
} else if ( Bedingung ){
    Anweisung
} else {
    Anweisung
}
```

[5] Wir weichen von dieser Regel sporadisch aus Gründen des Platzmangels auf übersichtlichen Folien ab.

Kontrollstrukturen - Verwendung von Blöcken

Beispiele

```
if (anzahl > 0)
    mean = sum / anzahl; ssq = sumsq / anzahl;


System.out.println(ssq);
```

Kontrollstrukturen - Verwendung von Blöcken

Beispiele

```
if (anzahl > 0) {
    mean = sum / anzahl;
}
ssq = sumsq / anzahl;
System.out.println(ssq);
```

Kontrollstrukturen: Schleifen

Es gibt mehrere Schleifen-Anweisungen

- `while`
- `do ... while`
- `for`
- und noch weitere Konstrukte (kommen später)

Kontrollstrukturen - While Schleife

```
while ( Bedingung )
    Anweisung
```

Kontrollstrukturen - While Schleife

Beispiele

```
int num = 143;
int factor = 2;

while (num % factor != 0){
       factor++;
}

System.out.println(factor + "|" + num);
```

Kontrollstrukturen - Do . . . While Schleife

```
do
   Anweisung
while ( Bedingung )
```

Kontrollstrukturen - Do . . . While Schleife

Beispiele

```
Scanner scanner = new Scanner(System.in);
String password;
do{
        System.out.print("enter magic word:");
        password = scanner.next();
} while (!password.equals("geheim"));
System.out.println("Door open.");
scanner.close();
```

Kontrollstrukturen - For Schleife

```
for ( Initialisierung ; Bedingung ; Fortschritt )
    Anweisung
```

$\Leftrightarrow$

```
Initialisierung
while ( Bedingung ) {
    Anweisung
    Fortschritt
}
```

Kontrollstrukturen - Do . . . While Schleife

Beispiele

```
for (int x = 1; x <= 128; x *= 2) {
        System.out.println("x= " + x);
}
```

$\Leftrightarrow$

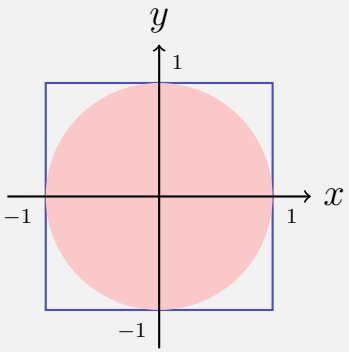
```
int x = 1;
while (x <= 128){
    System.out.println("x= " + x);
    x *= 2;
}
```

Ein Beispiel: Monte Carlo Simulation

Monte Carlo Simulation: Verwendung des Zufalls zum Lösen von Problemen. Breites Anwendungsfeld in der angewandten Mathematik und sämtlichen Natur- und Ingenieurswissenschaften.

π schätzen mit Monte Carlo Simulation

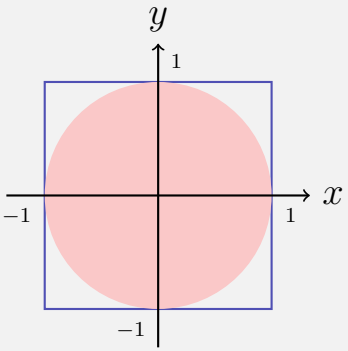
$$\frac{\text{Kreisfläche}}{\text{Fläche des Quadrates}} = \frac{\pi}{4}$$



π schätzen mit Monte Carlo Simulation

$$\frac{\text{Kreisfläche}}{\text{Fläche des Quadrates}} = \frac{\pi}{4}$$

Idee: Simuliere Zufallsvariable mit Gleichverteilung auf dem Einheitsquadrat $[0,1] \times [0,1]$.

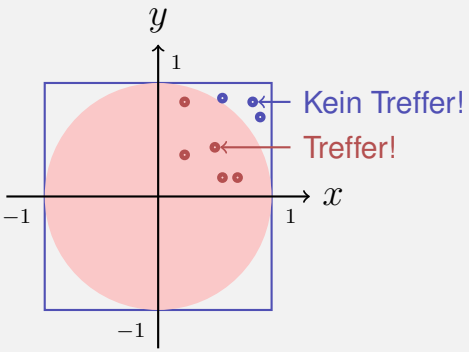


π schätzen mit Monte Carlo Simulation

$$\frac{\text{Kreisfläche}}{\text{Fläche des Quadrates}} = \frac{\pi}{4}$$

Idee: Simuliere Zufallsvariable mit Gleichverteilung auf dem Einheitsquadrat $[0, 1] \times [0, 1]$.

$$\frac{\text{Anzahl Treffer}}{\text{Anzahl Versuche}} \cdot 4 \approx \pi.$$



Aufgabe

- Experiment: Schätze π mit obigem Monte-Carlo Verfahren
- Führe das Experiment durch für Anzahl Versuche $1, 2, 4, \ldots, 4096$
- Jeweils Ausgabe der Anzahl Versuche und der Schätzung von π.

Hinweis: eine auf $[0, 1)$ uniformverteilte Pseudo-Zufallszahl bekommt man mit `Math.random()`.

π schätzen mit Monte Carlo Simulation

```
public class Pi {
  public static void main(String[] args){
    for (int trials = 1; trials <= 4096; trials*=2){
      int hits = 0;
      for (int i = 0; i<trials; ++i){
        double x = Math.random();
        double y = Math.random();
        if (x * x + y * y <= 1){
              hits++; }
      }
      double pi = (double)hits / trials * 4;
      System.out.println("trials=" + trials + ", pi=" + pi);
    }
  }
}
```