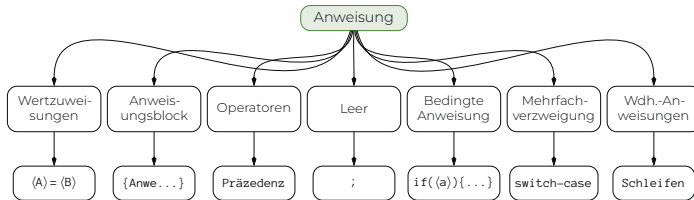


EIDI-KOMPAKT

Volle Version — 2.01.2022



Florian Sihler
Verteilte Systeme — Universität Ulm



1. Theoretische Grundlagen

- Der Algorithmusbegriff
- Algorithmen analysieren
- Konzeptionalisierung
- Übungsaufgaben

2. Java-Basics

- Wie ein Java-Programm entsteht
- Bezeichner & Variablen
- Komplexe Datentypen
- Subroutinen
- Zugriffsmodifikatoren
- Übungsaufgaben

3. Kontrollstrukturen und Arrays

- Fallunterscheidungen
- Schleifen
- Arrays
- Mehrdimensionale Konzepte
- Übungsaufgaben

4. OO-Konzepte

- Das Paradigma
- Klassen
- Enumerationen
- call-by-value/-reference
- Übungsaufgaben

5. Programmiertheorie

- Rekursion
- Laufzeitkomplexität
- Modellierung durch UML
- Zahlensysteme
- Übungsaufgaben

6. Weiterführende Konzepte

- Suchverfahren
- Sortierverfahren
- Übungsaufgaben

7. Dynamische Datenstrukturen

- Listen
- Stacks & Queues
- Bäume
- Graphen
- Übungsaufgaben

8. Java-Advanced

- Vererbung & Abstraktion
- Interfaces
- Fehlerbehandlungen
- Übungsaufgaben

DISCLAIMER



Der folgende Foliensatz erhebt keinen Anspruch auf vollständige Richtigkeit. Er wurde auf Basis der zugrundeliegenden Vorlesungsmaterialien erstellt und ist als unverbindliche Hilfe im Kontext seiner zusammenfassenden Natur zu verstehen. Hinzu kommt, dass die kapitelbasierte Einteilung der Vorlesung dort durchbrochen wurde, wo eine andere Gruppierung passender erschien. Ebenso wird ein Verständnis der Themen vorausgesetzt. So versuche ich es zwar zu vermeiden, Vorgriffe zu tätigen, scheue aber dennoch nicht vor ihnen zurück — sofern vonnöten.

Bei Anregungen oder Verbesserungsvorschlägen einfach melden!

Alle Grafiken wurden von mir, Florian Sihler, mithilfe von \LaTeX und TikZ erstellt. Ebenso wie das hier gezeigte Beamer-Layout. Für das Syntax-Highlighting wird das Paket `sopra-listings` verwendet. Weitere Links finden sich am Ende des Dokuments. florian.sihler@uni-ulm.de

„Viel Spaß beim Lernen!“

Theoretische Grundlagen



WAS IST EIN ALGORITHMUS?

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

(1) Wenn der Wert 42 übersteigt, (2) soll A ausgeführt werden. Ansonsten soll (3) B durchgeführt werden.

Textuell

WAS IST EIN ALGORITHMUS?

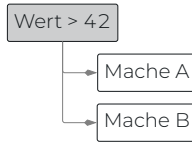
Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

(1) Wenn der *Wert* 42 übersteigt, (2) soll A ausgeführt werden. Ansonsten soll (3) B durchgeführt werden.

Textuell



Grafisch

WAS IST EIN ALGORITHMUS?

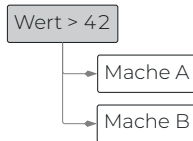
Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

(1) Wenn der *Wert* 42 übersteigt, (2) soll A ausgeführt werden. Ansonsten soll (3) B durchgeführt werden.

Textuell



Grafisch

```
1 ist Wert > 42 true:  
2 | Mache A;  
3 sonst:  
4 | Mache B;
```

Pseudocode

WAS IST EIN ALGORITHMUS?

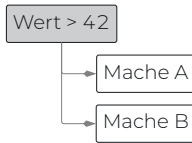
Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

(1) Wenn der *Wert* 42 übersteigt, (2) soll A ausgeführt werden. Ansonsten soll (3) B durchgeführt werden.

Textuell



Grafisch

```
1 ist Wert > 42 true:  
2 | Mache A;  
3 sonst:  
4 | Mache B;
```

Pseudocode

- Klassische Alltagsbeispiele: (Koch-)Rezepte, Gebrauchsanleitungen, ...

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:
PROZESS: Die Ausführung der Schritte eines Algorithmus.

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:
 - PROZESS: Die Ausführung der Schritte eines Algorithmus.
 - PROZESSOR: Der Ausführende (Mensch, Computer, ...).

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:
 - PROZESS: Die Ausführung der Schritte eines Algorithmus.
 - PROZESSOR: Der Ausführende (Mensch, Computer, ...).
 - ELEMENTAROPERATION: Eine einzelne, eindeutige Handlung.

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:
 - PROZESS: Die Ausführung der Schritte eines Algorithmus.
 - PROZESSOR: Der Ausführende (Mensch, Computer, ...).
 - ELEMENTAROPERATION: Eine einzelne, eindeutige Handlung.
- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:
 - PROZESS: Die Ausführung der Schritte eines Algorithmus.
 - PROZESSOR: Der Ausführende (Mensch, Computer, ...).
 - ELEMENTAROPERATION: Eine einzelne, eindeutige Handlung.
- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:
 - AUSFÜHRBARKEIT: Die Anleitung muss ausführ- und reproduzierbar sein.

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:
 - PROZESS: Die Ausführung der Schritte eines Algorithmus.
 - PROZESSOR: Der Ausführende (Mensch, Computer, ...).
 - ELEMENTAROPERATION: Eine einzelne, eindeutige Handlung.
- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:
 - AUSFÜHRBARKEIT: Die Anleitung muss ausführ- und reproduzierbar sein.
 - ENDLICHKEIT: Er muss mit endlich viel Text beschreibbar sein.

ALGORITHMUSEIGENSCHAFTEN

- Zu Beginn gilt es einige **Begriffe** zu klären:
 - PROZESS: Die Ausführung der Schritte eines Algorithmus.
 - PROZESSOR: Der Ausführende (Mensch, Computer, ...).
 - ELEMENTAROPERATION: Eine einzelne, eindeutige Handlung.
- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:
 - AUSFÜHRBARKEIT: Die Anleitung muss ausführ- und reproduzierbar sein.
 - ENDLICHKEIT: Er muss mit endlich viel Text beschreibbar sein.
 - VERARBEITUNG: Ein Algorithmus erhält Eingabedaten E , hält lokale Daten D und erzeugt Ausgabedaten A .

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

Termination

Ein Algorithmus terminiert, wenn er nach endlich vielen Schritten zum Ende kommt.

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

Ein Algorithmus terminiert, wenn er nach endlich vielen Schritten zum Ende kommt.

Partielle Korrektheit

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

Ein Algorithmus terminiert, wenn er nach endlich vielen Schritten zum Ende kommt.

Partielle Korrektheit

Wenn der Algorithmus für eine korrekte Eingabe terminiert, ist die erzeugte Ausgabe korrekt.

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

Ein Algorithmus terminiert, wenn er nach endlich vielen Schritten zum Ende kommt.

Partielle Korrektheit

Wenn der Algorithmus für eine korrekte Eingabe terminiert, ist die erzeugte Ausgabe korrekt.

Totale Korrektheit

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

Ein Algorithmus terminiert, wenn er nach endlich vielen Schritten zum Ende kommt.

Partielle Korrektheit

Wenn der Algorithmus für eine korrekte Eingabe terminiert, ist die erzeugte Ausgabe korrekt.

Totale Korrektheit

Der Algorithmus terminiert und ist partiell korrekt.

EIGENSCHAFTEN FÜR DIE ANALYSE, II

Determinismus

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

- Die Begriffe sind zu unterscheiden!

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

- Die Begriffe sind zu unterscheiden!
- Es können verschiedene Wege zum selben Ziel führen.

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

- Die Begriffe sind zu unterscheiden!
- Es können verschiedene Wege zum selben Ziel führen.
- (Sinnfreies) Beispiel:

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

- Die Begriffe sind zu unterscheiden!
- Es können verschiedene Wege zum selben Ziel führen.
- (Sinnfreies) Beispiel: Der Algorithmus kann jedes Element eines Arrays quadrieren. Die Reihenfolge wählt er zufällig aus!

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

- Die Begriffe sind zu unterscheiden!
- Es können verschiedene Wege zum selben Ziel führen.
- (Sinnfreies) Beispiel: Der Algorithmus kann jedes Element eines Arrays quadrieren. Die Reihenfolge wählt er zufällig aus!
 - › Dieser Algorithmus *determiniert*,

Determinismus

Der Ablauf ist *eindeutig*. Jeder Anweisung folgt (bei gleichen Voraussetzungen) die selbe.

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

- Die Begriffe sind zu unterscheiden!
- Es können verschiedene Wege zum selben Ziel führen.
- (Sinnfreies) Beispiel: Der Algorithmus kann jedes Element eines Arrays quadrieren. Die Reihenfolge wählt er zufällig aus!
 - › Dieser Algorithmus *determiniert*, ist aber *nicht-deterministisch*!

SCHEMA EINES ALGORITHMUS

SCHEMA EINES ALGORITHMUS



SCHEMA EINES ALGORITHMUS



- Ein Algorithmus benötigt (formal):

SCHEMA EINES ALGORITHMUS



- Ein Algorithmus benötigt (formal):
 - Vorbedingungen (wie „positive ganze Zahl“)

SCHEMA EINES ALGORITHMUS



- Ein Algorithmus benötigt (formal):
 - Vorbedingungen (wie „positive ganze Zahl“)
 - Nachbedingungen (wie „negative Fließkommazahl“)

SCHEMA EINES ALGORITHMUS



- Ein Algorithmus benötigt (formal):
 - Vorbedingungen (wie „positive ganze Zahl“)
 - Nachbedingungen (wie „negative Fließkommazahl“)
- Algorithmen fundieren oft auf einem Problem der realen Welt und bedienen sich Mechaniken, wie der Abstraktion, zur Lösung.

SCHEMA EINES ALGORITHMUS



- Ein Algorithmus benötigt (formal):
 - Vorbedingungen (wie „positive ganze Zahl“)
 - Nachbedingungen (wie „negative Fließkommazahl“)
- Algorithmen fundieren oft auf einem Problem der realen Welt und bedienen sich Mechaniken, wie der Abstraktion, zur Lösung.
- (Java-)Programme sind eine Darstellungsform von Algorithmen.

SCHEMA EINES ALGORITHMUS



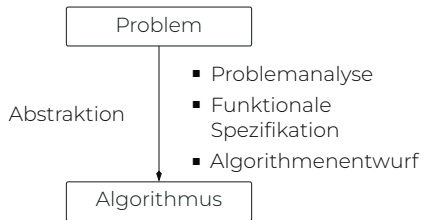
- Ein Algorithmus benötigt (formal):
 - Vorbedingungen (wie „positive ganze Zahl“)
 - Nachbedingungen (wie „negative Fließkommazahl“)
- Algorithmen fundieren oft auf einem Problem der realen Welt und bedienen sich Mechaniken, wie der Abstraktion, zur Lösung.
- (Java-)Programme sind eine Darstellungsform von Algorithmen. Diese kann von Maschinen interpretiert und ausgeführt werden.

SCHEMA EINES ALGORITHMUS, II

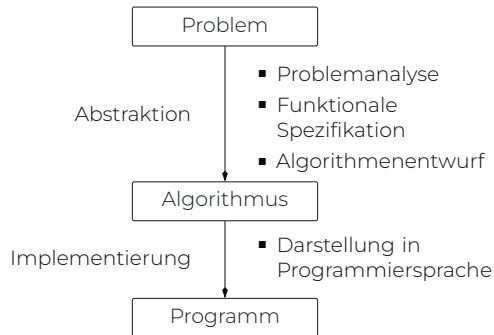
SCHEMA EINES ALGORITHMUS, II

Problem

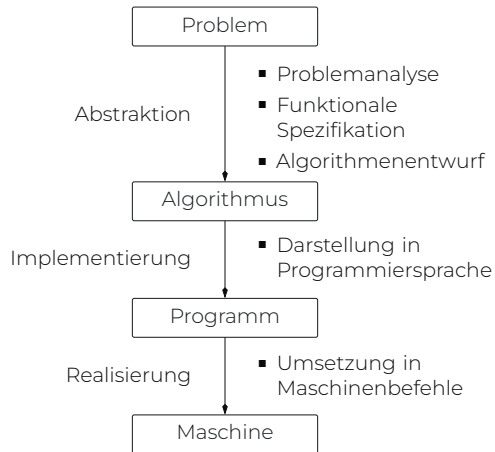
SCHEMA EINES ALGORITHMUS, II



SCHEMA EINES ALGORITHMUS, II



SCHEMA EINES ALGORITHMUS, II



WIE EIN PROGRAMM ENTSTEHT

WIE EIN PROGRAMM ENTSTEHT

- Ein Programm fällt nicht aus den Wolken (leider).

WIE EIN PROGRAMM ENTSTEHT

- Ein Programm fällt nicht aus den Wolken (leider).
- Im Softwareengineering gliedert man den Prozess meist in sechs Phasen.

WIE EIN PROGRAMM ENTSTEHT

- Ein Programm fällt nicht aus den Wolken (leider).
- Im Softwareengineering gliedert man den Prozess meist in sechs Phasen.
- Die sechste Phase (Dokumentation) läuft dabei parallel zu den anderen.

WIE EIN PROGRAMM ENTSTEHT

- Ein Programm fällt nicht aus den Wolken (leider).
- Im Softwareengineering gliedert man den Prozess meist in sechs Phasen.
- Die sechste Phase (Dokumentation) läuft dabei parallel zu den anderen.
- Die Dokumentation bezeichnet nicht nur die exakte Beschreibung aller für die Umsetzung notwendiger Komponenten,

WIE EIN PROGRAMM ENTSTEHT

- Ein Programm fällt nicht aus den Wolken (leider).
- Im Softwareengineering gliedert man den Prozess meist in sechs Phasen.
- Die sechste Phase (Dokumentation) läuft dabei parallel zu den anderen.
- Die Dokumentation bezeichnet nicht nur die exakte Beschreibung aller für die Umsetzung notwendiger Komponenten, sondern schließt auch Beschreibungen des Codes,

WIE EIN PROGRAMM ENTSTEHT

- Ein Programm fällt nicht aus den Wolken (leider).
- Im Softwareengineering gliedert man den Prozess meist in sechs Phasen.
- Die sechste Phase (Dokumentation) läuft dabei parallel zu den anderen.
- Die Dokumentation bezeichnet nicht nur die exakte Beschreibung aller für die Umsetzung notwendiger Komponenten, sondern schließt auch Beschreibungen des Codes, sowie des Ablaufs selbst mit ein!

WIE EIN PROGRAMM ENTSTEHT, II

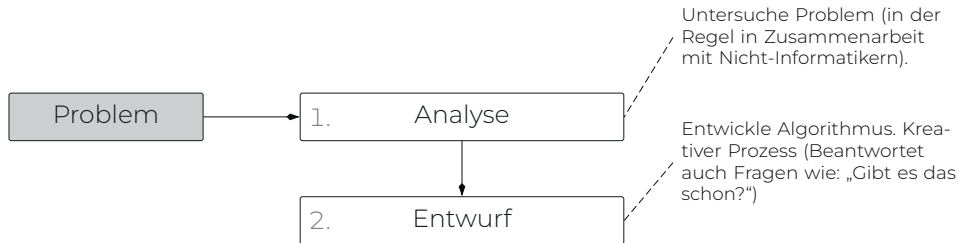
WIE EIN PROGRAMM ENTSTEHT, II

Problem

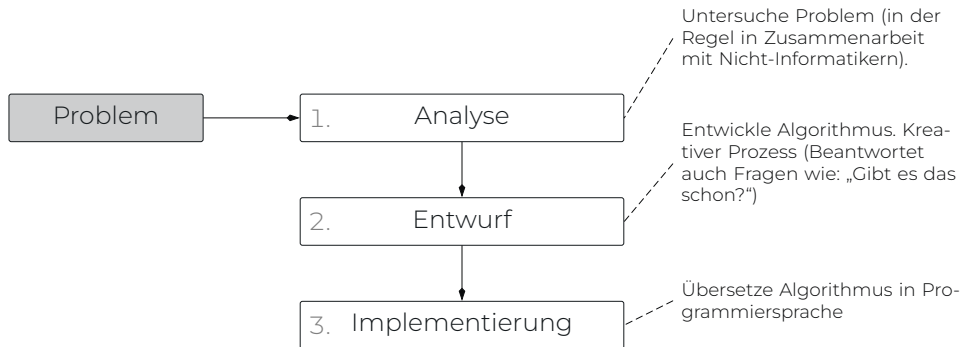
WIE EIN PROGRAMM ENTSTEHT, II



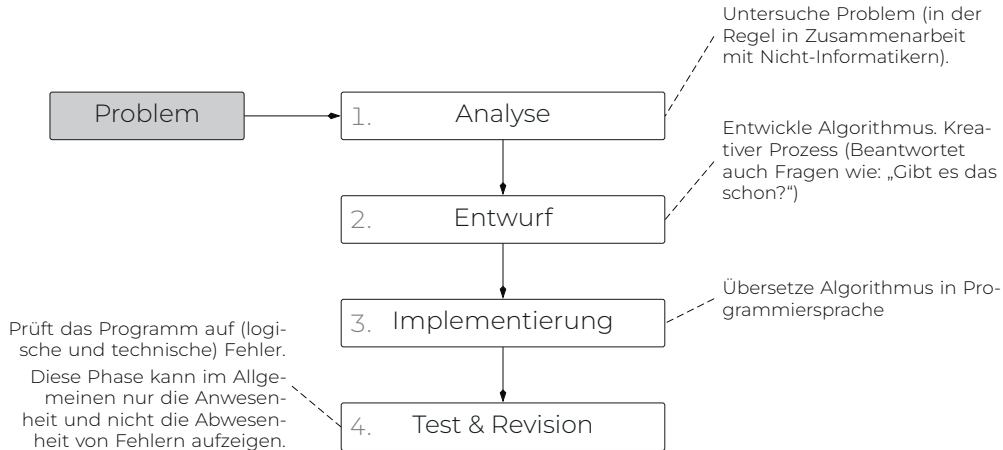
WIE EIN PROGRAMM ENTSTEHT, II



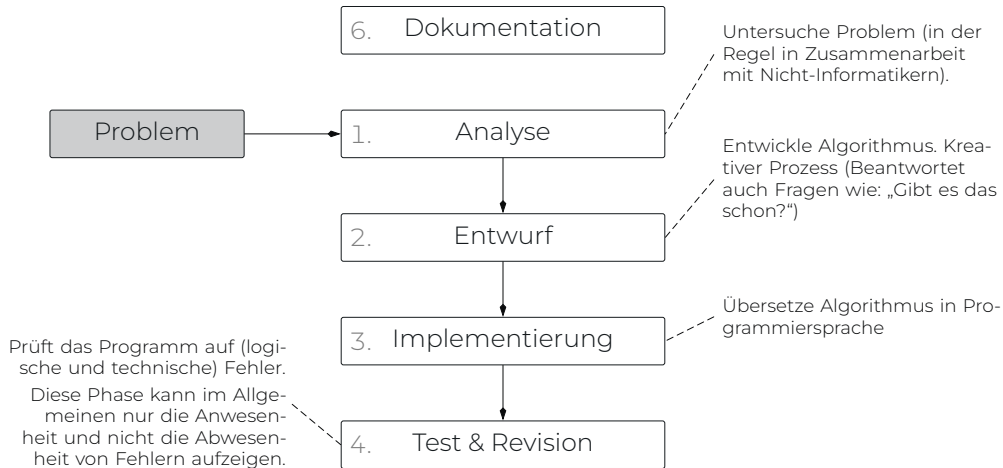
WIE EIN PROGRAMM ENTSTEHT, II



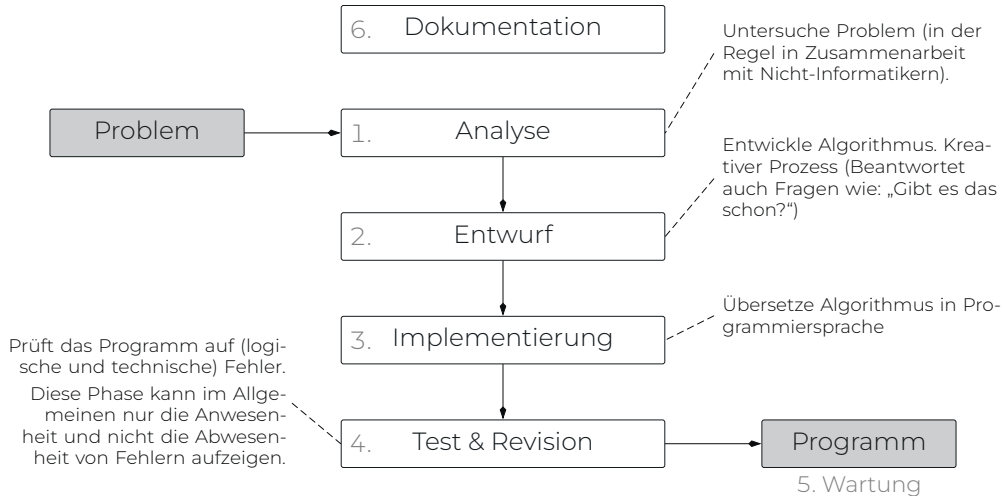
WIE EIN PROGRAMM ENTSTEHT, II



WIE EIN PROGRAMM ENTSTEHT, II



WIE EIN PROGRAMM ENTSTEHT, II



ÜBUNGSAUFGABE

Aufgabe 1: Algorithmus beschreiben

(2 Minuten)

Aufgabe 1: Algorithmus beschreiben

(2 Minuten)

Beschreiben Sie einen Algorithmus, der das maximale Element einer Datenmenge bestimmt.

Aufgabe 1: Algorithmus beschreiben

(2 Minuten)

Beschreiben Sie einen Algorithmus, der das maximale Element einer Datenmenge bestimmt.

Hinweis: Auf den Elementen muss keine „numerische“ Ordnung definiert sein. Sie können aber vergleichen, ob ein Element größer als ein anderes ist.

Lösung 1: Algorithmus beschreiben

Wir betrachten den textuellen Ansatz:

Lösung 1: Algorithmus beschreiben

Wir betrachten den textuellen Ansatz:

1. Keine Eingabedaten? → Fertig.

Lösung 1: Algorithmus beschreiben

Wir betrachten den textuellen Ansatz:

1. Keine Eingabedaten? → Fertig.
2. Setze Maximum auf erstes Element der Eingabedaten.

Lösung 1: Algorithmus beschreiben

Wir betrachten den textuellen Ansatz:

1. Keine Eingabedaten? → Fertig.
2. Setze Maximum auf erstes Element der Eingabedaten.
3. Tue für jedes weitere Element der Eingabedaten:

Lösung 1: Algorithmus beschreiben

Wir betrachten den textuellen Ansatz:

1. Keine Eingabedaten? → Fertig.
2. Setze Maximum auf erstes Element der Eingabedaten.
3. Tue für jedes weitere Element der Eingabedaten:
 - 3.1 Ist das zu betrachtende Element „größer“ als das bisherige Maximum?

Lösung 1: Algorithmus beschreiben

Wir betrachten den textuellen Ansatz:

1. Keine Eingabedaten? → Fertig.
2. Setze Maximum auf erstes Element der Eingabedaten.
3. Tue für jedes weitere Element der Eingabedaten:
 - 3.1 Ist das zu betrachtende Element „größer“ als das bisherige Maximum?
JA: Ersetze Maximum durch dieses Element.

Lösung 1: Algorithmus beschreiben

Wir betrachten den textuellen Ansatz:

1. Keine Eingabedaten? → Fertig.
2. Setze Maximum auf erstes Element der Eingabedaten.
3. Tue für jedes weitere Element der Eingabedaten:
 - 3.1 Ist das zu betrachtende Element „größer“ als das bisherige Maximum?
JA: Ersetze Maximum durch dieses Element.
NEIN: Tue nichts.

ÜBUNGSAUFGABE

Aufgabe 2: Charakteristika

(2 Minuten)

Aufgabe 2: Charakteristika

(2 Minuten)

Ist der folgende Algorithmus für die Berechnung des Modulo $a \bmod b$ (total) korrekt?

Aufgabe 2: Charakteristika

(2 Minuten)

Ist der folgende Algorithmus für die Berechnung des Modulo $a \bmod b$ (total) korrekt?

```
public static int simpleMod(int a, int b){
    while(a >= b)
        a -= b;
    return a;
}
```


Lösung 2: Charakteristika

Der Algorithmus ist nicht total korrekt.

Lösung 2: Charakteristika

Der Algorithmus ist nicht total korrekt. So liefert er für ein negatives b kein korrektes Ergebnis,

Lösung 2: Charakteristika

Der Algorithmus ist nicht total korrekt. So liefert er für ein negatives b kein korrektes Ergebnis, da hier $a \neq b$ zu einem Überlauf führt (bis dahin ist stets $a \geq b$). Weiter terminiert er nicht für ein $b = 0$ mit einem $a > 0$ (da dann von a immer nur 0 abgezogen wird, was es nicht weiter verkleinert).

Lösung 2: Charakteristika

Der Algorithmus ist nicht total korrekt. So liefert er für ein negatives b kein korrektes Ergebnis, da hier $a \neq b$ zu einem Überlauf führt (bis dahin ist stets $a \geq b$). Weiter terminiert er nicht für ein $b = 0$ mit einem $a > 0$ (da dann von a immer nur 0 abgezogen wird, was es nicht weiter verkleinert).
Ferner ist er auch nicht partiell korrekt.

Lösung 2: Charakteristika

Der Algorithmus ist nicht total korrekt. So liefert er für ein negatives b kein korrektes Ergebnis, da hier $a \neq b$ zu einem Überlauf führt (bis dahin ist stets $a \geq b$). Weiter terminiert er nicht für ein $b = 0$ mit einem $a > 0$ (da dann von a immer nur 0 abgezogen wird, was es nicht weiter verkleinert).

Ferner ist er auch nicht partiell korrekt. So liefert er für ein negatives a bei einem positiven b einfach nur a zurück.

ÜBUNGSAUFGABE

Aufgabe 3: Korrektheitsbeweis

(4 Minuten)

Zeigen Sie die totale Korrektheit des folgenden Algorithmus:

Aufgabe 3: Korrektheitsbeweis

(4 Minuten)

Zeigen Sie die totale Korrektheit des folgenden Algorithmus:

Algorithm 3: Finden des Minimum

Input: $a, b, c \in \mathbb{N}$

Output: Minimum of a, b, c

```
1 elems := (a, b, c);  
2 min := a;  
3 foreach elem in elems do  
4   | if elem < min then min := elem ;  
5 end  
6 return min;
```

Lösung 3: Korrektheitsbeweis

Wir zeigen die Termination und die partielle Korrektheit getrennt:

Lösung 3: Korrektheitsbeweis

Wir zeigen die Termination und die partielle Korrektheit getrennt:

- Das for-each aus Zeile 3 endet sicher nach drei Durchläufen. Es gibt keine weiteren Sprünge (wie Rekursion, ...): der Algorithmus *terminiert* für alle $a, b, c \in \mathbb{N}$.

Lösung 3: Korrektheitsbeweis

Wir zeigen die Termination und die partielle Korrektheit getrennt:

- Das for-each aus Zeile 3 endet sicher nach drei Durchläufen. Es gibt keine weiteren Sprünge (wie Rekursion, ...): der Algorithmus *terminiert* für alle $a, b, c \in \mathbb{N}$.
- Hier kann man alle Fälle $a = b = c, a > b = c, \dots$ einzeln prüfen. Sonst:

Lösung 3: Korrektheitsbeweis

Wir zeigen die Termination und die partielle Korrektheit getrennt:

- Das for-each aus Zeile 3 endet sicher nach drei Durchläufen. Es gibt keine weiteren Sprünge (wie Rekursion, ...): der Algorithmus *terminiert* für alle $a, b, c \in \mathbb{N}$.
- Hier kann man alle Fälle $a = b = c, a > b = c, \dots$ einzeln prüfen. Sonst:
 - $a = \min\{a, b, c\}$ so initialisiert Zeile 2 *min* schon korrekt, $b \geq a$ und $c \geq a$.

Lösung 3: Korrektheitsbeweis

Wir zeigen die Termination und die partielle Korrektheit getrennt:

- Das for-each aus Zeile 3 endet sicher nach drei Durchläufen. Es gibt keine weiteren Sprünge (wie Rekursion, ...): der Algorithmus *terminiert* für alle $a, b, c \in \mathbb{N}$.
- Hier kann man alle Fälle $a = b = c$, $a > b = c$, ... einzeln prüfen. Sonst:
 - $a = \min\{a, b, c\}$ so initialisiert Zeile 2 *min* schon korrekt, $b \geq a$ und $c \geq a$.
 - $b = \min\{a, b, c\}$ so überschreibt Zeile 3 *min* für b , da $b \leq a$ und $c \geq b$.

Lösung 3: Korrektheitsbeweis

Wir zeigen die Termination und die partielle Korrektheit getrennt:

- Das for-each aus Zeile 3 endet sicher nach drei Durchläufen. Es gibt keine weiteren Sprünge (wie Rekursion, ...): der Algorithmus *terminiert* für alle $a, b, c \in \mathbb{N}$.
- Hier kann man alle Fälle $a = b = c$, $a > b = c$, ... einzeln prüfen. Sonst:
 - $a = \min\{a, b, c\}$ so initialisiert Zeile 2 *min* schon korrekt, $b \geq a$ und $c \geq a$.
 - $b = \min\{a, b, c\}$ so überschreibt Zeile 3 *min* für b , da $b \leq a$ und $c \geq b$.
 - $c = \min\{a, b, c\}$ so überschreibt Zeile 3 *min* für c , da $c \leq a$ und $b \geq c$.

ÜBUNGSaufGABE

Aufgabe 4: Deterministische Determiniertheit

(2 Minuten)

Nennen Sie je ein Beispiel für einen Algorithmus der:

Aufgabe 4: Deterministische Determiniertheit

(2 Minuten)

Nennen Sie je ein Beispiel für einen Algorithmus der:

- i) nicht-deterministisch ist und determiniert.

Aufgabe 4: Deterministische Determiniertheit

(2 Minuten)

Nennen Sie je ein Beispiel für einen Algorithmus der:

- i) nicht-deterministisch ist und determiniert.
- ii) sowohl deterministisch ist, als auch determiniert.

Lösung 4: Deterministische Determiniertheit

Lösung 4: Deterministische Determiniertheit

- i) Suche, ob ein Element in den Eingabedaten vorkommt.

Lösung 4: Deterministische Determiniertheit

- i) Suche, ob ein Element in den Eingabedaten vorkommt. Die Elemente werden vom Algorithmus in einer zufälligen Reihenfolge verglichen.

Lösung 4: Deterministische Determiniertheit

- i) Suche, ob ein Element in den Eingabedaten vorkommt. Die Elemente werden vom Algorithmus in einer zufälligen Reihenfolge verglichen.
- ii) Ein Algorithmus der eine Zahl verdoppelt, indem er sie mit 2 multipliziert.

Java-Basics



VOM TEXT ZUM PROGRAMM

VOM TEXT ZUM PROGRAMM

Programmtext

VOM TEXT ZUM PROGRAMM



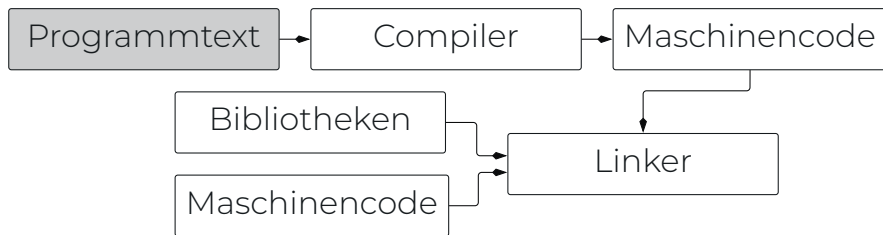
VOM TEXT ZUM PROGRAMM



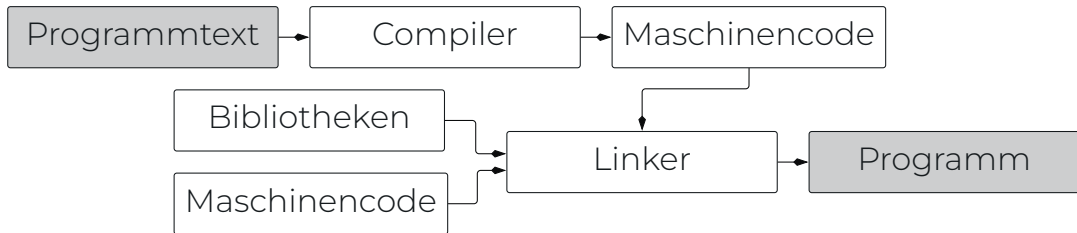
VOM TEXT ZUM PROGRAMM



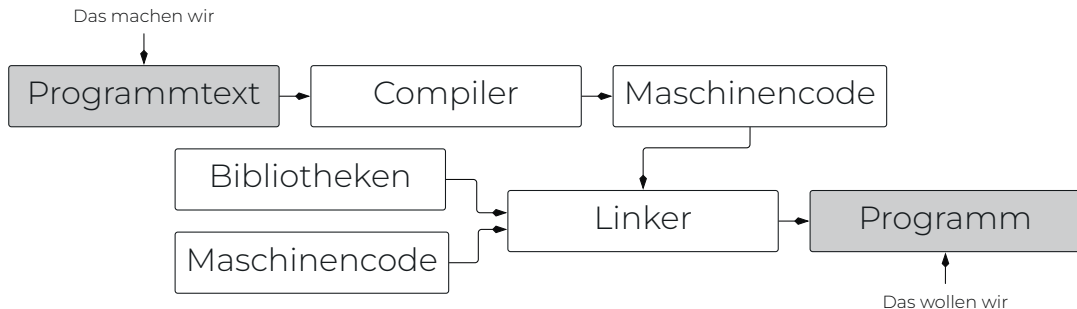
VOM TEXT ZUM PROGRAMM



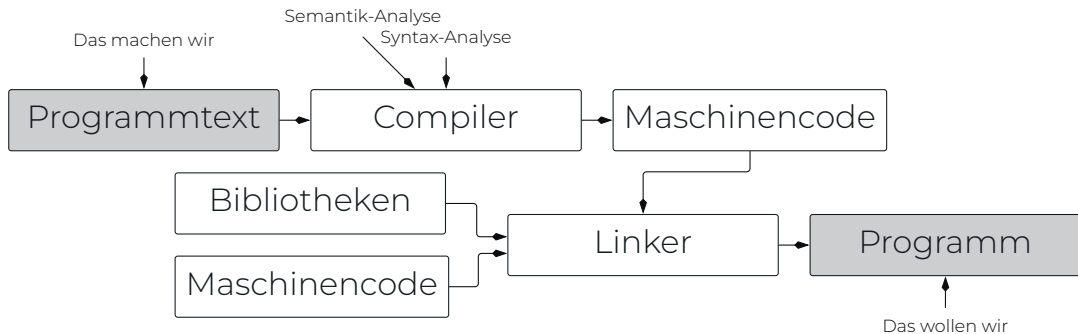
VOM TEXT ZUM PROGRAMM



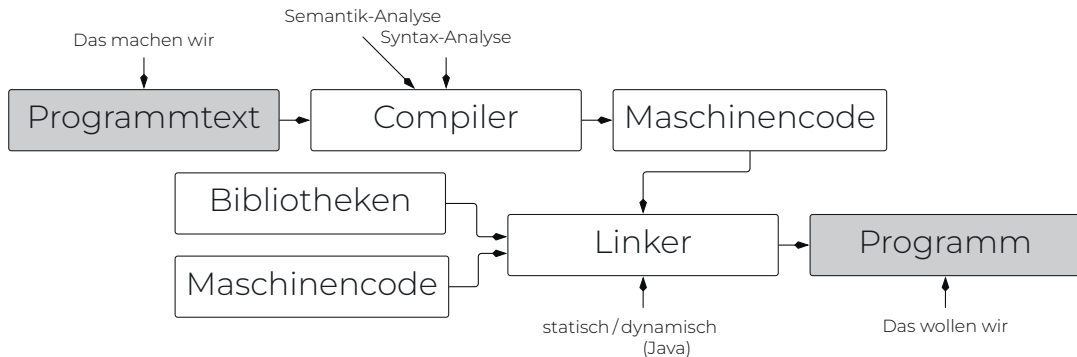
VOM TEXT ZUM PROGRAMM



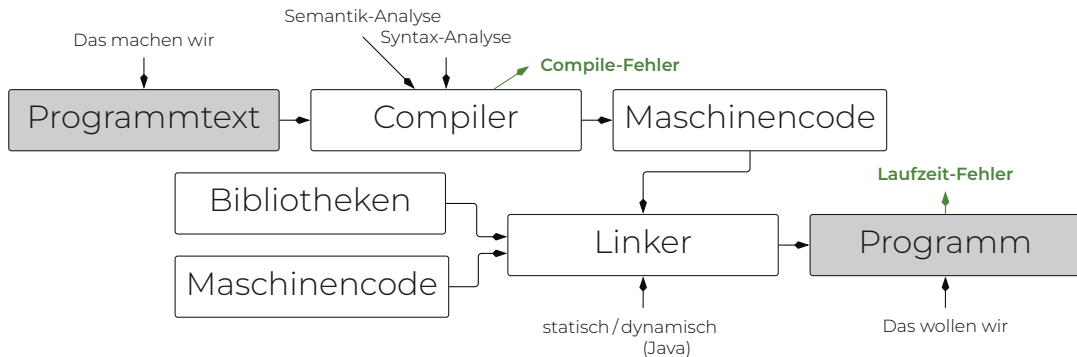
VOM TEXT ZUM PROGRAMM



VOM TEXT ZUM PROGRAMM



VOM TEXT ZUM PROGRAMM



VOM TEXT ZUM PROGRAMM, II

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode*

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).
- Dieser Code kann vom Java-Interpreter ausgeführt werden:

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).
- Dieser Code kann vom Java-Interpreter ausgeführt werden: **java** `<Name>`

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).
- Dieser Code kann vom Java-Interpreter ausgeführt werden: **java** `<Name>`
- Ein paar wichtige Punkte:

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).
- Dieser Code kann vom Java-Interpreter ausgeführt werden: **java** `<Name>`
- Ein paar wichtige Punkte:
 - Bei den `.class`-Dateien handelt es sich um *keine* `.zip`-Dateien!

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).
- Dieser Code kann vom Java-Interpreter ausgeführt werden: **java** `<Name>`
- Ein paar wichtige Punkte:
 - Bei den `.class`-Dateien handelt es sich um *keine* `.zip`-Dateien!
 - Die `.jar`-Dateien von Java sind `.zip`-Archive.

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).
- Dieser Code kann vom Java-Interpreter ausgeführt werden: **java** `<Name>`
- Ein paar wichtige Punkte:
 - Bei den `.class`-Dateien handelt es sich um *keine* `.zip`-Dateien!
 - Die `.jar`-Dateien von Java sind `.zip`-Archive.
 - Der Interpreter (**java**) ist sowohl im Java-Runtime-Environment (JRE) als auch im Java-Development-Kit (JDK) enthalten.

VOM TEXT ZUM PROGRAMM, II

- Mit **javac** `<Name>.java` übersetzt der Java-Compiler die Datei in *Java-Bytecode* (`<Name>.class`).
- Dieser Code kann vom Java-Interpreter ausgeführt werden: **java** `<Name>`
- Ein paar wichtige Punkte:
 - Bei den `.class`-Dateien handelt es sich um *keine* `.zip`-Dateien!
 - Die `.jar`-Dateien von Java sind `.zip`-Archive.
 - Der Interpreter (**java**) ist sowohl im Java-Runtime-Environment (JRE) als auch im Java-Development-Kit (JDK) enthalten.
 - Der Compiler wird (in der Regel) nur mit der JDK ausgeliefert.

GÜLTIGE JAVA-BEZEICHNER

GÜLTIGE JAVA-BEZEICHNER

- Java stellt folgende Regeln an Bezeichner für Variablen, Klassen, ...

GÜLTIGE JAVA-BEZEICHNER

- Java stellt folgende Regeln an Bezeichner für Variablen, Klassen, ...
 - Diese dürfen nur aus Buchstaben, Ziffern, dem Unterstrich (_) und dem Dollarzeichen (\$) bestehen.

GÜLTIGE JAVA-BEZEICHNER

- Java stellt folgende Regeln an Bezeichner für Variablen, Klassen, ...
 - Diese dürfen nur aus Buchstaben, Ziffern, dem Unterstrich (_) und dem Dollarzeichen (\$) bestehen.
 - Ein Bezeichner darf *nicht* mit einer Ziffer beginnen.

GÜLTIGE JAVA-BEZEICHNER

- Java stellt folgende Regeln an Bezeichner für Variablen, Klassen, ...
 - Diese dürfen nur aus Buchstaben, Ziffern, dem Unterstrich (`_`) und dem Dollarzeichen (`$`) bestehen.
 - Ein Bezeichner darf *nicht* mit einer Ziffer beginnen.
 - Schlüsselbegriffe (wie `int`, `double`, ...) dürfen nicht als Bezeichner verwendet werden.

VARIABLEN UND WERTZUWEISUNGEN

VARIABLEN UND WERTZUWEISUNGEN

```
int zahl = -42, superZahl = 4;  
zahl = 42;
```

VARIABLEN UND WERTZUWEISUNGEN

```
int zahl = -42, superZahl = 4;  
zahl = 42;
```

- Java ist eine *streng typisierte* Sprache.

VARIABLEN UND WERTZUWEISUNGEN

```
int zahl = -42, superZahl = 4;  
zahl = 42;
```

- Java ist eine *streng typisierte* Sprache. Jede Variable kann nur Daten eines (bestimmten) Typs speichern.

VARIABLEN UND WERTZUWEISUNGEN

```
int zahl = -42, superZahl = 4;  
zahl = 42;
```

- Java ist eine *streng typisierte* Sprache. Jede Variable kann nur Daten eines (bestimmten) Typs speichern.
- Wir unterscheiden zwischen *primitiven* und *komplexen* Datentypen.

VARIABLEN UND WERTZUWEISUNGEN

```
int zahl = -42, superZahl = 4;  
zahl = 42;
```

- Java ist eine *streng typisierte* Sprache. Jede Variable kann nur Daten eines (bestimmten) Typs speichern.
- Wir unterscheiden zwischen *primitiven* und *komplexen* Datentypen.
PRIMITIVE: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`
Hinweis: Stack, *call-by-value*

VARIABLEN UND WERTZUWEISUNGEN

```
int zahl = -42, superZahl = 4;  
zahl = 42;
```

- Java ist eine *streng typisierte* Sprache. Jede Variable kann nur Daten eines (bestimmten) Typs speichern.
- Wir unterscheiden zwischen *primitiven* und *komplexen* Datentypen.
 - PRIMITIVE: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`
Hinweis: Stack, *call-by-value*
 - KOMPLEXE: `String`, `Random`, ...
Hinweis: Heap, „*call-by-reference*“

KONVERTIERUNG PRIMITIVER DATENTYPEN

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren.

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \leftarrow `short` \leftarrow `int` \leftarrow `long` \leftarrow `float` \leftarrow `double`

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \leftarrow `short` \leftarrow `int` \leftarrow `long` \leftarrow `float` \leftarrow `double`

Sowie:

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \prec `short` \prec `int` \prec `long` \prec `float` \prec `double`

Sowie:

`char` \prec `int`

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \prec `short` \prec `int` \prec `long` \prec `float` \prec `double`

Sowie:

`char` \prec `int`

- So kann Java einen `char` implizit in einen `int` umwandeln

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \prec `short` \prec `int` \prec `long` \prec `float` \prec `double`

Sowie:

`char` \prec `int`

- So kann Java einen `char` implizit in einen `int` umwandeln (da der Wertebereich größer ist).

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \prec `short` \prec `int` \prec `long` \prec `float` \prec `double`

Sowie:

`char` \prec `int`

- So kann Java einen `char` implizit in einen `int` umwandeln (da der Wertebereich größer ist). Umgekehrt allerdings nicht.

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \prec `short` \prec `int` \prec `long` \prec `float` \prec `double`

Sowie:

`char` \prec `int`

- So kann Java einen `char` implizit in einen `int` umwandeln (da der Wertebereich größer ist). Umgekehrt allerdings nicht.
- **Floats und Doubles unterliegen einem Rundungsproblem.**

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \leftarrow `short` \leftarrow `int` \leftarrow `long` \leftarrow `float` \leftarrow `double`

Sowie:

`char` \leftarrow `int`

- So kann Java einen `char` implizit in einen `int` umwandeln (da der Wertebereich größer ist). Umgekehrt allerdings nicht.
- **Floats und Doubles unterliegen einem Rundungsproblem.**
- Unter der Einschränkung des Wertebereichs kann durch `(<Datentyp >) <Ausdruck >` eine explizite Konvertierung auch in umgekehrte Richtung erfolgen.

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

`byte` \leftarrow `short` \leftarrow `int` \leftarrow `long` \leftarrow `float` \leftarrow `double`

Sowie:

`char` \leftarrow `int`

- So kann Java einen `char` implizit in einen `int` umwandeln (da der Wertebereich größer ist). Umgekehrt allerdings nicht.
- **Floats und Doubles unterliegen einem Rundungsproblem.**
- Unter der Einschränkung des Wertebereichs kann durch `(<Datentyp >) <Ausdruck >` eine explizite Konvertierung auch in umgekehrte Richtung erfolgen. Beispiel: `(byte)42`

ÜBUNGSAUFGABE

Aufgabe 5: Implizite Typkonvertierung

(2 Minuten)

Aufgabe 5: Implizite Typkonvertierung

(2 Minuten)

In welche Datentypen kann im allgemeinen Fall `short` a) *ohne* expliziten Typecast konvertiert werden? b) Welche Datentypen sind zusätzlich *mit* explizitem Typecast möglich?

Lösung 5: Implizite Typkonvertierung

Lösung 5: Implizite Typkonvertierung

a) Implizit: `int`, `long`, `float` und `double`.

Lösung 5: Implizite Typkonvertierung

- a) Implizit: `int`, `long`, `float` und `double`.
- b) Explizit zusätzlich: `byte` und `char`.

KONSTANTEN UND HINWEISE

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

- Schreiben wir eine Fließkommazahl wie 3.1415, so interpretiert sie Java erstmal als **double**.

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

- Schreiben wir eine Fließkommazahl wie 3.1415, so interpretiert sie Java erstmal als **double**. Damit sie als **float** interpretiert wird, müssen wir ein **f** anstellen.

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

- Schreiben wir eine Fließkommazahl wie 3.1415, so interpretiert sie Java erstmal als **double**. Damit sie als **float** interpretiert wird, müssen wir ein **f** anstellen. Also: 3.1415f.

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

- Schreiben wir eine Fließkommazahl wie 3.1415, so interpretiert sie Java erstmal als **double**. Damit sie als **float** interpretiert wird, müssen wir ein **f** anstellen. Also: 3.1415f.
- Zeichen (**char**) werden in Java im UTF-16 Format gespeichert.

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

- Schreiben wir eine Fließkommazahl wie 3.1415, so interpretiert sie Java erstmal als **double**. Damit sie als **float** interpretiert wird, müssen wir ein **f** anstellen. Also: 3.1415f.
- Zeichen (**char**) werden in Java im UTF-16 Format gespeichert. Für die unteren 7-Bit ist es identisch zur ASCII-Kodierung.

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

- Schreiben wir eine Fließkommazahl wie 3.1415, so interpretiert sie Java erstmal als **double**. Damit sie als **float** interpretiert wird, müssen wir ein **f** anstellen. Also: 3.1415f.
- Zeichen (**char**) werden in Java im UTF-16 Format gespeichert. Für die unteren 7-Bit ist es identisch zur ASCII-Kodierung.
- Der „Datentyp“ **void** gibt bei Methoden an,

KONSTANTEN UND HINWEISE

- Konstanten werden in Java mit dem Schlüsselbegriff **final** gekennzeichnet. Sie können nur genau einmal zugewiesen werden:

```
final int SUPER_ZAHL = 42;
```

- Schreiben wir eine Fließkommazahl wie `3.1415`, so interpretiert sie Java erstmal als `double`. Damit sie als `float` interpretiert wird, müssen wir ein `f` anstellen. Also: `3.1415f`.
- Zeichen (`char`) werden in Java im UTF-16 Format gespeichert. Für die unteren 7-Bit ist es identisch zur ASCII-Kodierung.
- Der „Datentyp“ `void` gibt bei Methoden an, dass diese nichts zurückgeben!

PRÄZEDENZREGELN

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt.

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$[a++, a--]$$

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$[a++, a--] \rightarrow [!a, -a, ++a, --a]$$

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$[a++, a--] \rightarrow [!a, -a, ++a, --a] \rightarrow [*, /, \%]$$

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$[a++, a--] \rightarrow [!a, -a, ++a, --a] \rightarrow [*, /, \%] \rightarrow [a + b, a - b]$$

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$\begin{aligned} [a++, a--] &\rightarrow [!a, -a, ++a, --a] \rightarrow [*, /, \%] \rightarrow [a + b, a - b] \\ &\rightarrow [==, >=, <, \dots] \end{aligned}$$

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$\begin{aligned} [a++, a--] &\rightarrow [!a, -a, ++a, --a] \rightarrow [*, /, \%] \rightarrow [a + b, a - b] \\ &\rightarrow [==, >=, <, \dots] \rightarrow [\&\&] \end{aligned}$$

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$\begin{aligned} [a++, a--] &\rightarrow [!a, -a, ++a, --a] \rightarrow [*, /, \%] \rightarrow [a + b, a - b] \\ &\rightarrow [==, >=, <, \dots] \rightarrow [\&\&] \rightarrow [||] \end{aligned}$$

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt. Diese wird durch die Präzedenzregeln bestimmt:

$$\begin{aligned} [a++, a--] &\rightarrow [!a, -a, ++a, --a] \rightarrow [*, /, \%] \rightarrow [a + b, a - b] \\ &\rightarrow [==, >=, <, \dots] \rightarrow [\&\&] \rightarrow [||] \end{aligned}$$

Aufgabe 6: Was liefert dieser Code?

(3 Minuten)

```
int x = 40, m = 3;
System.out.println(-x++ - --m);
System.out.println(x > 41 || m > 1 && 2 != 3);
```

PRÄZEDENZREGELN – LÖSUNG

Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```


Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```

Aufgrund der Postfix-Notation `x++` wird `x` *nach* dem Ausdruck um 1 erhöht!

Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```

Aufgrund der Postfix-Notation `x++` wird `x` *nach* dem Ausdruck um 1 erhöht! Wegen des Präfix-Dekrements von `m` wird `m` um eins verringert.

Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```

Aufgrund der Postfix-Notation `x++` wird `x` *nach* dem Ausdruck um 1 erhöht! Wegen des Präfix-Dekrements von `m` wird `m` um eins verringert. Nun werden `-x` (= -40) mit `m` (= 2) subtrahiert (= -42).

Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```

Aufgrund der Postfix-Notation `x++` wird `x` *nach* dem Ausdruck um 1 erhöht! Wegen des Präfix-Dekrements von `m` wird `m` um eins verringert. Nun werden `-x` ($= -40$) mit `m` ($= 2$) subtrahiert ($= -42$). `x` ist nun `41`.

```
System.out.println(x > 41 || m > 1 && 2 != 3); // → true
```

Da `||` am schwächsten bindet,

Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```

Aufgrund der Postfix-Notation `x++` wird `x` *nach* dem Ausdruck um 1 erhöht! Wegen des Präfix-Dekrements von `m` wird `m` um eins verringert. Nun werden `-x` ($= -40$) mit `m` ($= 2$) subtrahiert ($= -42$). `x` ist nun `41`.

```
System.out.println(x > 41 || m > 1 && 2 != 3); // → true
```

Da `||` am schwächsten bindet, werden zuerst `x > 41` ($41 > 41 \rightarrow \text{false}$) und `m > 1 && 2 != 3` betrachtet.

Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```

Aufgrund der Postfix-Notation `x++` wird `x` *nach* dem Ausdruck um 1 erhöht! Wegen des Präfix-Dekrements von `m` wird `m` um eins verringert. Nun werden `-x` ($= -40$) mit `m` ($= 2$) subtrahiert ($= -42$). `x` ist nun `41`.

```
System.out.println(x > 41 || m > 1 && 2 != 3); // → true
```

Da `||` am schwächsten bindet, werden zuerst `x > 41` ($41 > 41 \rightarrow \text{false}$) und `m > 1 && 2 != 3` betrachtet. Nun wird also `m > 1` ($2 > 1 \rightarrow \text{true}$) und `2 != 3` (true) ausgewertet.

Lösung 6: Was liefert dieser Code?

```
System.out.println(-x++ - --m); // → -42
```

Aufgrund der Postfix-Notation `x++` wird `x` *nach* dem Ausdruck um 1 erhöht! Wegen des Präfix-Dekrements von `m` wird `m` um eins verringert. Nun werden `-x` ($= -40$) mit `m` ($= 2$) subtrahiert ($= -42$). `x` ist nun `41`.

```
System.out.println(x > 41 || m > 1 && 2 != 3); // → true
```

Da `||` am schwächsten bindet, werden zuerst `x > 41` ($41 > 41 \rightarrow \text{false}$) und `m > 1 && 2 != 3` betrachtet. Nun wird also `m > 1` ($2 > 1 \rightarrow \text{true}$) und `2 != 3` (true) ausgewertet. Damit evaluiert `true && true` zu `true` und somit auch `false || true` zu `true`.

PRÄZEDENZREGELN, II

Aufgabe 7: Was liefert dieser Code?, II

(2 Minuten)

Aufgabe 7: Was liefert dieser Code?, II

(2 Minuten)

```
int x = 5;  
System.out.println(x);  
System.out.println(x++ + ++x + x++ + ++x);  
System.out.println(x);
```

Aufgabe 7: Was liefert dieser Code?, II

(2 Minuten)

```
int x = 5;  
System.out.println(x);  
System.out.println(x++ + ++x + x++ + ++x);  
System.out.println(x);
```

Hinweis: Dies ist lediglich eine Bonusübung, um den Unterschied zwischen der Präfix- und der Postfixnotation klar zu machen.

PRÄZEDENZREGELN, II – LÖSUNG

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile.

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile. Aufgeschlüsselt wird hier addiert:
 $5 + 7 + 7 + 9 = 28$.

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile. Aufgeschlüsselt wird hier addiert:
 $5 + 7 + 7 + 9 = 28$. Zuerst ist es 5, da `x` erst nachträglich erhöht wird.

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile. Aufgeschlüsselt wird hier addiert: $5 + 7 + 7 + 9 = 28$. Zuerst ist es 5, da `x` erst nachträglich erhöht wird. Dann wird `x`, welches jetzt 6 ist, direkt um eins erhöht.

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile. Aufgeschlüsselt wird hier addiert: $5 + 7 + 7 + 9 = 28$. Zuerst ist es 5, da `x` erst nachträglich erhöht wird. Dann wird `x`, welches jetzt 6 ist, direkt um eins erhöht. `x` ist jetzt 7, das Spiel wiederholt sich für das Hintere `++x`.

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile. Aufgeschlüsselt wird hier addiert: $5 + 7 + 7 + 9 = 28$. Zuerst ist es 5, da `x` erst nachträglich erhöht wird. Dann wird `x`, welches jetzt 6 ist, direkt um eins erhöht. `x` ist jetzt 7, das Spiel wiederholt sich für das Hintere `++x`.

```
System.out.println(x); // → 9
```

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile. Aufgeschlüsselt wird hier addiert: $5 + 7 + 7 + 9 = 28$. Zuerst ist es 5, da `x` erst nachträglich erhöht wird. Dann wird `x`, welches jetzt 6 ist, direkt um eins erhöht. `x` ist jetzt 7, das Spiel wiederholt sich für das Hintere `++x`.

```
System.out.println(x); // → 9
```

Durch die 4 Inkremente ist `x` nun 9.

Lösung 7: Was liefert dieser Code?, II

```
System.out.println(x); // → 5
```

Zu Beginn hält `x` den Wert 5, so weit nichts besonderes.

```
System.out.println(x++ + ++x + x++ + ++x); // → 28
```

Dies ist wohl die schwierigste Zeile. Aufgeschlüsselt wird hier addiert: $5 + 7 + 7 + 9 = 28$. Zuerst ist es 5, da `x` erst nachträglich erhöht wird. Dann wird `x`, welches jetzt 6 ist, direkt um eins erhöht. `x` ist jetzt 7, das Spiel wiederholt sich für das Hintere `++x`.

```
System.out.println(x); // → 9
```

Durch die 4 Inkremente ist `x` nun 9. (*Das ist kein guter Code!*)

PRÄZEDENZREGELN – KOMMENTAR

- Java wendet die mathematischen Rechenregeln wie bekannt an.

PRÄZEDENZREGELN – KOMMENTAR

- Java wendet die mathematischen Rechenregeln wie bekannt an.
- Auch das Klammern funktioniert wie gewohnt.

PRÄZEDENZREGELN – KOMMENTAR

- Java wendet die mathematischen Rechenregeln wie bekannt an.
- Auch das Klammern funktioniert wie gewohnt.
- Boolesche Operatoren funktionieren wie in der Aussagenlogik.

STANDARDWERTE

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - *Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.*

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - *Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.*
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - *Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.*
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - *Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.*
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)

- Java weist bestimmten Variablen initiale Werte zu!
 - *Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.*
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)
 - Komplexe Datentypen werden `null`.

- Java weist bestimmten Variablen initiale Werte zu!
 - *Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.*
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)
 - Komplexe Datentypen werden `null`.
 - Kurzgesagt: Sie werden „Null“.

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)
 - Komplexe Datentypen werden `null`.
 - Kurzgesagt: Sie werden „Null“.

```
public class Example {  
    private char k;  
    static Example ex;  
    void foo() {  
        int i;  
        float[] fs = new float[3];  
    }  
}
```

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)
 - Komplexe Datentypen werden `null`.
 - Kurzgesagt: Sie werden „Null“.

```
public class Example {  
    private char k;  
    static Example ex;  
    void foo() {  
        int i;  
        float[] fs = new float[3];  
    }  
}
```

- `k = '\u0000'`.

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)
 - Komplexe Datentypen werden `null`.
 - Kurzgesagt: Sie werden „Null“.

```
public class Example {  
    private char k;  
    static Example ex;  
    void foo() {  
        int i;  
        float[] fs = new float[3];  
    }  
}
```

- `k = '\u0000'`.
- `ex = null`.

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)
 - Komplexe Datentypen werden `null`.
 - Kurzgesagt: Sie werden „Null“.

```
public class Example {  
    private char k;  
    static Example ex;  
    void foo() {  
        int i;  
        float[] fs = new float[3];  
    }  
}
```

- `k = '\u0000'`.
- `ex = null`.
- `i` ist nur Deklariert, nicht Initialisiert.

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!
 - Nur: (nicht-finale) Klassen-, Instanzvariablen und Array-Komponenten.
 - `byte` wird `(byte)0`, `short` wird `(short)0`, `int` wird `0` und `long` wird `0L`.
 - `float` wird `0.0f` und `double` wird `0.0d`.
 - `char` wird zu `'\u0000'` („NUL“)
 - Komplexe Datentypen werden `null`.
 - Kurzgesagt: Sie werden „Null“.

```
public class Example {  
    private char k;  
    static Example ex;  
    void foo() {  
        int i;  
        float[] fs = new float[3];  
    }  
}
```

- `k = '\u0000'`.
- `ex = null`.
- `i` ist nur Deklariert, nicht Initialisiert.
- `fs = new float[] {0.0f, 0.0f, 0.0f}`.

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.
- Wichtig ist `String`, der von Java eine Sonderbehandlung erfährt.

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.
- Wichtig ist `String`, der von Java eine Sonderbehandlung erfährt.
- Bei der Deklaration und Initialisierung eines komplexen Datentyps:

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.
- Wichtig ist `String`, der von Java eine Sonderbehandlung erfährt.
- Bei der Deklaration und Initialisierung eines komplexen Datentyps:
`Random rnd = new Random();`

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.
- Wichtig ist `String`, der von Java eine Sonderbehandlung erfährt.
- Bei der Deklaration und Initialisierung eines komplexen Datentyps:
`Random rnd = new Random();`
speichert Java für `rnd` die Referenz,

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.
- Wichtig ist `String`, der von Java eine Sonderbehandlung erfährt.
- Bei der Deklaration und Initialisierung eines komplexen Datentyps:

```
Random rnd = new Random();
```

speichert Java für `rnd` die Referenz, an der sich die eigentlichen Daten des Objekts befinden. Stichwort: Heap und Stack.

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.
- Wichtig ist `String`, der von Java eine Sonderbehandlung erfährt.
- Bei der Deklaration und Initialisierung eines komplexen Datentyps:

```
Random rnd = new Random();
```

speichert Java für `rnd` die Referenz, an der sich die eigentlichen Daten des Objekts befinden. Stichwort: Heap und Stack.

- Deswegen sollten komplexe Datentypen mittels `.equals()` verglichen werden.

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.
- Java erlaubt es, mit Klassen komplexe Datentypen zu konstruieren.
- Wichtig ist `String`, der von Java eine Sonderbehandlung erfährt.
- Bei der Deklaration und Initialisierung eines komplexen Datentyps:

```
Random rnd = new Random();
```

speichert Java für `rnd` die Referenz, an der sich die eigentlichen Daten des Objekts befinden. Stichwort: Heap und Stack.

- Deswegen sollten komplexe Datentypen mittels `.equals()` verglichen werden. „`==`“ vergleicht die Speicheradressen und damit die *Identität*.

STRINGS

STRINGS

- Jedes Objekt `obj` (nicht `null`) kann in Java mittels `obj.toString()` in einen `String` konvertiert werden. Standard: `<Klassenname>@<HashCode>`.

STRINGS

- Jedes Objekt `obj` (nicht `null`) kann in Java mittels `obj.toString()` in einen `String` konvertiert werden. Standard: `<Klassenname>@<HashCode>`.
- Wir können auf einem `String`-Objekt diverse Operationen aufrufen:

STRINGS

- Jedes Objekt `obj` (nicht `null`) kann in Java mittels `obj.toString()` in einen `String` konvertiert werden. Standard: `<Klassenname>@<HashCode>`.
- Wir können auf einem String-Objekt diverse Operationen aufrufen:
`s1.equals(s2)`: Prüft, ob die Zeichenketten *gleich* sind.

STRINGS

- Jedes Objekt `obj` (nicht `null`) kann in Java mittels `obj.toString()` in einen `String` konvertiert werden. Standard: `<Klassenname>@<HashCode>`.
- Wir können auf einem `String`-Objekt diverse Operationen aufrufen:
 - `s1.equals(s2)`: Prüft, ob die Zeichenketten *gleich* sind.
 - `s1.length()`: Liefert die Länge einer Zeichenkette.

- Jedes Objekt `obj` (nicht `null`) kann in Java mittels `obj.toString()` in einen `String` konvertiert werden. Standard: `<Klassenname>@<HashCode>`.
- Wir können auf einem String-Objekt diverse Operationen aufrufen:
 - `s1.equals(s2)`: Prüft, ob die Zeichenketten *gleich* sind.
 - `s1.length()`: Liefert die Länge einer Zeichenkette.
 - `s1.charAt(k)`: Liefert das `k`-te Zeichen.

- Jedes Objekt `obj` (nicht `null`) kann in Java mittels `obj.toString()` in einen `String` konvertiert werden. Standard: `<Klassenname>@<HashCode>`.
- Wir können auf einem String-Objekt diverse Operationen aufrufen:
 - `s1.equals(s2)`: Prüft, ob die Zeichenketten *gleich* sind.
 - `s1.length()`: Liefert die Länge einer Zeichenkette.
 - `s1.charAt(k)`: Liefert das k -te Zeichen.
 - `s1.substring(a, b)`: Liefert einen Ausschnitt der Zeichenkette vom a -ten bis zum $b - 1$ -ten Zeichen.

- Jedes Objekt `obj` (nicht `null`) kann in Java mittels `obj.toString()` in einen `String` konvertiert werden. Standard: `<Klassenname>@<HashCode>`.
- Wir können auf einem String-Objekt diverse Operationen aufrufen:
 - `s1.equals(s2)`: Prüft, ob die Zeichenketten *gleich* sind.
 - `s1.length()`: Liefert die Länge einer Zeichenkette.
 - `s1.charAt(k)`: Liefert das `k`-te Zeichen.
 - `s1.substring(a, b)`: Liefert einen Ausschnitt der Zeichenkette vom `a`-ten bis zum `b - 1`-ten Zeichen.
 - `s1.toUpperCase()`: Liefert die Zeichenkette in Großbuchstaben.

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

```
<Zugriffsmodifikatoren> <Rückgabetyt> <Name>(<Parameterliste>) {  
    <Körper>  
}
```

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

```
⟨ Zugriffsmodifikatoren ⟩ ⟨ Rückgabetyt ⟩ ⟨ Name ⟩ (⟨ Parameterliste ⟩) {  
    ⟨ Körper ⟩  
}
```

- Beispiel:

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

```
⟨Zugriffsmodifikatoren⟩ ⟨Rückgabetyt⟩ ⟨Name⟩(⟨Parameterliste⟩) {  
    ⟨Körper⟩  
}
```

- Beispiel:

```
private static String generateWelcomeMessage(String name, int age,  
    boolean happy) {  
    return "Welcome,_" + name /* ... */ ;  
}
```

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

```
⟨Zugriffsmodifikatoren⟩ ⟨Rückgabetyt⟩ ⟨Name⟩(⟨Parameterliste⟩) {  
    ⟨Körper⟩  
}
```

- Beispiel:

```
private static String generateWelcomeMessage(String name, int age,  
    boolean happy) {  
    return "Welcome,_" + name /* ... */ ;  
}
```

- Hinweis:

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

```
⟨Zugriffsmodifikatoren⟩ ⟨Rückgabetyt⟩ ⟨Name⟩(⟨Parameterliste⟩) {  
    ⟨Körper⟩  
}
```

- Beispiel:

```
private static String generateWelcomeMessage(String name, int age,  
    boolean happy) {  
    return "Welcome,_" + name /* ... */ ;  
}
```

- Hinweis: Bevor wir zu Klassen an sich kommen,

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

```
⟨Zugriffsmodifikatoren⟩ ⟨Rückgabetyt⟩ ⟨Name⟩(⟨Parameterliste⟩) {  
    ⟨Körper⟩  
}
```

- Beispiel:

```
private static String generateWelcomeMessage(String name, int age,  
    boolean happy) {  
    return "Welcome,_" + name /* ... */ ;  
}
```

- Hinweis: Bevor wir zu Klassen an sich kommen, machen wir alle Funktionen **static**,

FUNKTIONEN

- Analog zur `main`-Funktion, lassen sich in Java Routinen implementieren.
- Die allgemeine Syntax lautet:

```
⟨Zugriffsmodifikatoren⟩ ⟨Rückgabebetyp⟩ ⟨Name⟩(⟨Parameterliste⟩) {  
    ⟨Körper⟩  
}
```

- Beispiel:

```
private static String generateWelcomeMessage(String name, int age,  
    boolean happy) {  
    return "Welcome,_" + name /* ... */ ;  
}
```

- Hinweis: Bevor wir zu Klassen an sich kommen, machen wir alle Funktionen **static**, um sie auch von der `main`-Funktion aus aufrufen zu können.

SIGNATUR & ÜBERLADUNG

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter.

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabebetyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabebetyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.
- Java verbietet zwei Funktionen mit gleicher Signatur im selben Gültigkeitsbereich.

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabotyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.
- Java verbietet zwei Funktionen mit gleicher Signatur im selben Gültigkeitsbereich.
- Gestattet sind Funktionen mit gleichem Namen, aber verschiedener Parameterliste.

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabotyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.
- Java verbietet zwei Funktionen mit gleicher Signatur im selben Gültigkeitsbereich.
- Gestattet sind Funktionen mit gleichem Namen, aber verschiedener Parameterliste. Dieses Prinzip wird *Überladen* (Overloading) genannt.

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabotyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.
- Java verbietet zwei Funktionen mit gleicher Signatur im selben Gültigkeitsbereich.
- Gestattet sind Funktionen mit gleichem Namen, aber verschiedener Parameterliste. Dieses Prinzip wird *Überladen* (Overloading) genannt.
- Existieren mehrere Überladungen mit gleicher Parameteranzahl

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabotyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.
- Java verbietet zwei Funktionen mit gleicher Signatur im selben Gültigkeitsbereich.
- Gestattet sind Funktionen mit gleichem Namen, aber verschiedener Parameterliste. Dieses Prinzip wird *Überladen* (Overloading) genannt.
- Existieren mehrere Überladungen mit gleicher Parameteranzahl (wie `m(int, double)`, `m(char, float)`, ...),

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabotyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.
- Java verbietet zwei Funktionen mit gleicher Signatur im selben Gültigkeitsbereich.
- Gestattet sind Funktionen mit gleichem Namen, aber verschiedener Parameterliste. Dieses Prinzip wird *Überladen* (Overloading) genannt.
- Existieren mehrere Überladungen mit gleicher Parameteranzahl (wie `m(int, double)`, `m(char, float)`, ...), entscheidet Java welche Methode aufgerufen werden soll.

SIGNATUR & ÜBERLADUNG

- Die Signatur einer Methode besteht aus dem Namen sowie den Datentypen der Parameter. Der Rückgabotyp ist *kein* Teil!
 - › Beispiel: `generateWelcomeMessage(String, int, boolean)`.
- Java verbietet zwei Funktionen mit gleicher Signatur im selben Gültigkeitsbereich.
- Gestattet sind Funktionen mit gleichem Namen, aber verschiedener Parameterliste. Dieses Prinzip wird *Überladen* (Overloading) genannt.
- Existieren mehrere Überladungen mit gleicher Parameteranzahl (wie `m(int, double)`, `m(char, float)`, ...), entscheidet Java welche Methode aufgerufen werden soll. Es nimmt die „Nächste“.

PRIVATE UND PUBLIC

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:
PRIVATE: Zugriff ist nur innerhalb der Klasse erlaubt.

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:
 - PRIVATE: Zugriff ist nur innerhalb der Klasse erlaubt.
 - PROTECTED: Ist sichtbar im gesamten Paket sowie in allen Unterklassen (Vererbung).

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:
 - PRIVATE: Zugriff ist nur innerhalb der Klasse erlaubt.
 - PROTECTED: Ist sichtbar im gesamten Paket sowie in allen Unterklassen (Vererbung).
 - PUBLIC: Ist überall sichtbar (auch über Pakete hinweg).

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:
 - PRIVATE: Zugriff ist nur innerhalb der Klasse erlaubt.
 - PROTECTED: Ist sichtbar im gesamten Paket sowie in allen Unterklassen (Vererbung).
 - PUBLIC: Ist überall sichtbar (auch über Pakete hinweg).
 - „DEFAULT“: Gibt man nichts an, so kann die Variable von Überall im Paket („selber Ordner“) erreicht werden.

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:
 - PRIVATE: Zugriff ist nur innerhalb der Klasse erlaubt.
 - PROTECTED: Ist sichtbar im gesamten Paket sowie in allen Unterklassen (Vererbung).
 - PUBLIC: Ist überall sichtbar (auch über Pakete hinweg).
 - „DEFAULT“: Gibt man nichts an, so kann die Variable von Überall im Paket („selber Ordner“) erreicht werden.
- Hinweis (Vorgriff):

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:
 - PRIVATE: Zugriff ist nur innerhalb der Klasse erlaubt.
 - PROTECTED: Ist sichtbar im gesamten Paket sowie in allen Unterklassen (Vererbung).
 - PUBLIC: Ist überall sichtbar (auch über Pakete hinweg).
 - „DEFAULT“: Gibt man nichts an, so kann die Variable von Überall im Paket („selber Ordner“) erreicht werden.
- Hinweis (Vorgriff): Objekte einer Klasse können auch auf private Elemente anderer Objekte der gleichen Klasse zugreifen.

GÜLTIGKEITSBEREICHE – SCOPES

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1.
```

```
}
```


GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1.  
    private int sozialversicherungsnummer; // 2.  
  
}
```

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1.  
    private int sozialversicherungsnummer; // 2.  
    protected int watschelIndex; // 3.  
  
}
```

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1.  
    private int sozialversicherungsnummer; // 2.  
    protected int watschelIndex; // 3.  
    public void watschel() { /* ... */ } // 4.  
  
}
```

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1.  
    private int sozialversicherungsnummer; // 2.  
    protected int watschelIndex; // 3.  
    public void watschel() { /* ... */ } // 4.  
    void piepsen() { /* ... */ } // 5.  
}
```

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1.  
    private int sozialversicherungsnummer; // 2.  
    protected int watschelIndex; // 3.  
    public void watschel() { /* ... */ } // 4.  
    void piepsen() { /* ... */ } // 5.  
}
```

- Die Klasse **Tiger** sei im selben, **Auto** sei in einem anderen Paket, **Felspingu** erbe von **Pinguin** aber sei in einem anderem Paket:

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt
    public void watschel() { /* ... */ } // 4. (erneut: global)
    void piepsen() { /* ... */ } // 5. Standard: Paket
}
```

- Die Klasse **Tiger** sei im selben, **Auto** sei in einem anderen Paket, **Felspingu** erbe von **Pinguin** aber sei in einem anderem Paket:

	1	2	3	4	5
Pinguin					
Tiger					

	1	2	3	4	5
Auto					
Felspingu					

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt
    public void watschel() { /* ... */ } // 4. (erneut: global)
    void piepsen() { /* ... */ } // 5. Standard: Paket
}
```

- Die Klasse **Tiger** sei im selben, **Auto** sei in einem anderen Paket, **Felspingu** erbe von **Pinguin** aber sei in einem anderem Paket:

	1	2	3	4	5
Pinguin	✓	✓	✓	✓	✓
Tiger					

	1	2	3	4	5
Auto					
Felspingu					

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt
    public void watschel() { /* ... */ } // 4. (erneut: global)
    void piepsen() { /* ... */ } // 5. Standard: Paket
}
```

- Die Klasse **Tiger** sei im selben, **Auto** sei in einem anderen Paket, **Felspingu** erbe von **Pinguin** aber sei in einem anderem Paket:

	1	2	3	4	5
Pinguin	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Tiger	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

	1	2	3	4	5
Auto					
Felspingu					

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt
    public void watschel() { /* ... */ } // 4. (erneut: global)
    void piepsen() { /* ... */ } // 5. Standard: Paket
}
```

- Die Klasse **Tiger** sei im selben, **Auto** sei in einem anderen Paket, **Felspingu** erbe von **Pinguin** aber sei in einem anderem Paket:

	1	2	3	4	5
Pinguin	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Tiger	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

	1	2	3	4	5
Auto	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Felspingu					

GÜLTIGKEITSBEREICHE – SCOPES

- Es gibt vier Geltungsbereiche (*engl. scopes*) für Variablen.

```
public class Pinguin { // 1. global
    private int sozialversicherungsnummer; // 2. Klasse
    protected int watschelIndex; // 3. geschützt
    public void watschel() { /* ... */ } // 4. (erneut: global)
    void piepsen() { /* ... */ } // 5. Standard: Paket
}
```

- Die Klasse **Tiger** sei im selben, **Auto** sei in einem anderen Paket, **Felspingu** erbe von **Pinguin** aber sei in einem anderem Paket:

	1	2	3	4	5
Pinguin	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Tiger	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

	1	2	3	4	5
Auto	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Felspingu	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

ÜBUNGSAUFGABE

Aufgabe 8: Sichtbarkeiten

(6 Minuten)

Welche der Aufrufe (1 bis 6) sind gültig oder ungültig und warum? Nehmen Sie an, dass die Dateien und Importe korrekt sind. B **extends** A heißt: B erbt von A.

Aufgabe 8: Sichtbarkeiten

(6 Minuten)

Welche der Aufrufe (1 bis 6) sind gültig oder ungültig und warum? Nehmen Sie an, dass die Dateien und Importe korrekt sind. B **extends** A heißt: B erbt von A.



Aufgabe 8: Sichtbarkeiten

(6 Minuten)

Welche der Aufrufe (1 bis 6) sind gültig oder ungültig und warum? Nehmen Sie an, dass die Dateien und Importe korrekt sind. B **extends** A heißt: B erbt von A.

```
public class Apple { void eat() { /* ... */ } }
```

```
public class Banana extends Apple {  
    public int lookAt(int x) { /* ... */ }  
    public void eat() { /* ... */ }  
    protected boolean eat(float amount) { /* ... */ }  
}
```

```
class Morrowind {  
    static int test() {  
        new Apple().eat(); // 1  
        Banana b = new Banana();  
        b.lookAt(14); // 2  
        b.eat(); // 3  
    }  
}
```

```
class Skyrim extends Banana {  
    static int test() {  
        new Apple().eat(); // 4  
        eat(3f); // 5  
        Banana b = new Banana();  
        b.lookAt(14); // 6  
    }  
}
```



Lösung 8: Sichtbarkeiten

1. *Ungültig*: Die Konstruktion eines „Apple-Objektes“ funktioniert, `Apple::eat()` ist in `Morrowind` aber nicht sichtbar (package private).

Lösung 8: Sichtbarkeiten

1. *Ungültig*: Die Konstruktion eines „Apple-Objektes“ funktioniert, `Apple::eat()` ist in `Morrowind` aber nicht sichtbar (package private).
2. *Gültig*: Von `Banana` kann ein Objekt erzeugt werden, `Banana::lookAt(int)` ist **public** sichtbar.

Lösung 8: Sichtbarkeiten

1. *Ungültig*: Die Konstruktion eines „Apple-Objektes“ funktioniert, `Apple::eat()` ist in `Morrowind` aber nicht sichtbar (package private).
2. *Gültig*: Von `Banana` kann ein Objekt erzeugt werden, `Banana::lookAt(int)` ist **public** sichtbar.
3. *Gültig*: Von `Banana` kann ein Objekt erzeugt werden, weiter erweitert `Banana::eat()` durch das Überschreiben die Sichtbarkeit von `Apple::eat()` auf **public**.

Lösung 8: Sichtbarkeiten

(Fortsetzung)

4. *Ungültig*: Die Konstruktion eines „Apple-Objektes“ funktioniert, `Apple::eat()` ist in `Skyrim` aber nicht sichtbar (package private).

Lösung 8: Sichtbarkeiten

(Fortsetzung)

4. *Ungültig*: Die Konstruktion eines „Apple-Objektes“ funktioniert, `Apple::eat()` ist in `Skyrim` aber nicht sichtbar (package private).
5. *Ungültig*: `Skyrim::test()` ist statisch, für `Banana::eat(float)` benötigt es aber ein Objekt.

Lösung 8: Sichtbarkeiten

(Fortsetzung)

4. *Ungültig*: Die Konstruktion eines „Apple-Objektes“ funktioniert, `Apple::eat()` ist in `Skyrim` aber nicht sichtbar (package private).
5. *Ungültig*: `Skyrim::test()` ist statisch, für `Banana::eat(float)` benötigt es aber ein Objekt.
6. *Gültig*: Von `Banana` kann ein Objekt erzeugt werden, `Banana::lookAt(int)` ist **public** sichtbar.

Definition 2: Subroutinen

Eine Subroutine / Funktion ist ein *Unterprogramm*,

Definition 2: Subroutinen

Eine Subroutine / Funktion ist ein *Unterprogramm*, das von anderen Programm(teilen) aufgerufen und so wiederverwendet werden kann.

Definition 2: Subroutinen

Eine Subroutine / Funktion ist ein *Unterprogramm*, das von anderen Programm(teilen) aufgerufen und so wiederverwendet werden kann.

Sie wird durch ihre *Signatur* eindeutig identifiziert,

Definition 2: Subroutinen

Eine Subroutine / Funktion ist ein *Unterprogramm*, das von anderen Programm(teilen) aufgerufen und so wiederverwendet werden kann.

Sie wird durch ihre *Signatur* eindeutig identifiziert, die in Java aus einem Bezeichner sowie einer Liste an Datentypen der Parameter besteht. Ein Unterprogramm ist in der Lage Daten an den aufrufenden Teil zurückzuliefern.

ÜBUNGSAUFGABE

Aufgabe 9: Fehler finden, I

(3 Minuten)

Finden und korrigieren Sie alle (syntaktischen) Fehler:

Aufgabe 9: Fehler finden, I

(3 Minuten)

Finden und korrigieren Sie alle (syntaktischen) Fehler:

```
public void main(String[] hihi) {  
    private final int X = 12;  
    System.out.println("Huhu " + (byte) X)  
}
```

Die korrigierte Version soll die Ausgabe „Huhu 12“ erzeugen.

Lösung 9: Fehler finden, I

```
public static void main(String[] hihi) {  
    final int X = 12;  
    System.out.println("Huhu_" + (byte) X);  
}
```

Lösung 9: Fehler finden, I

```
public static void main(String[] hihi) {  
    final int X = 12;  
    System.out.println("Huhu_" + (byte) X);  
}
```

1. Die `main`-Methode muss **static** sein! (Diskutierbar.)

Lösung 9: Fehler finden, I

```
public static void main(String[] hihi) {  
    final int X = 12;  
    System.out.println("Huhu_" + (byte) X);  
}
```

1. Die `main`-Methode muss **static** sein! (Diskutierbar.)
2. Zugriffsmodifikatoren wie **private** dürfen nicht in einer Funktion auftreten.

Lösung 9: Fehler finden, I

```
public static void main(String[] hihi) {  
    final int X = 12;  
    System.out.println("Huhu_" + (byte) X);  
}
```

1. Die `main`-Methode muss **static** sein! (Diskutierbar.)
2. Zugriffsmodifikatoren wie **private** dürfen nicht in einer Funktion auftreten.
3. Die Funktion `println` heißt `println`.

Lösung 9: Fehler finden, I

```
public static void main(String[] hihi) {  
    final int X = 12;  
    System.out.println("Huhu_" + (byte) X);  
}
```

1. Die `main`-Methode muss **static** sein! (Diskutierbar.)
2. Zugriffsmodifikatoren wie **private** dürfen nicht in einer Funktion auftreten.
3. Die Funktion `println` heißt `println`.
4. In der Ausgabezeile fehlt ein Semikolon!

Lösung 9: Fehler finden, I

```
public static void main(String[] hihi) {  
    final int X = 12;  
    System.out.println("Huhu_" + (byte) X);  
}
```

1. Die `main`-Methode muss **static** sein! (Diskutierbar.)
2. Zugriffsmodifikatoren wie **private** dürfen nicht in einer Funktion auftreten.
3. Die Funktion `println` heißt `println`.
4. In der Ausgabezeile fehlt ein Semikolon!

Hinweis: Das „hihi“ ist kein Fehler!

ÜBUNGSaufGABE

Aufgabe 10: Fehler finden, II

(3 Minuten)

Finden und korrigieren Sie alle (syntaktischen) Fehler:

Aufgabe 10: Fehler finden, II

(3 Minuten)

Finden und korrigieren Sie alle (syntaktischen) Fehler:

```
class A {  
    public int double(int x) { return (2*)x; }  
  
    public static void main() {  
        System.in.print(double(21));  
    }  
}
```

Aufgabe 10: Fehler finden, II

(3 Minuten)

Finden und korrigieren Sie alle (syntaktischen) Fehler:

```
class A {  
    public int double(int x) { return (2*)x; }  
  
    public static void main() {  
        System.in.print(double(21));  
    }  
}
```

Dabei soll Ihr korrigiertes Programm beim Start durch **java** die Ausgabe **42** (ohne neue Zeile) erzeugen.

Lösung 10: Fehler finden, II

```
public static int doDouble(int x) { return (2*x); }  
public static void main(String[] args) {  
    System.out.print(doDouble(21));  
}
```

Lösung 10: Fehler finden, II

```
public static int doDouble(int x) { return (2*x); }  
public static void main(String[] args) {  
    System.out.print(doDouble(21));  
}
```

1. Der Bezeichner `double` ist für eine Funktion nicht erlaubt.

Lösung 10: Fehler finden, II

```
public static int doDouble(int x) { return (2*x); }  
public static void main(String[] args) {  
    System.out.print(doDouble(21));  
}
```

1. Der Bezeichner `double` ist für eine Funktion nicht erlaubt.
2. Der Ausdruck in der Funktion ist ungültig geklammert.

Lösung 10: Fehler finden, II

```
public static int doDouble(int x) { return (2*x); }  
public static void main(String[] args) {  
    System.out.print(doDouble(21));  
}
```

1. Der Bezeichner `double` ist für eine Funktion nicht erlaubt.
2. Der Ausdruck in der Funktion ist ungültig geklammert.
3. Die Funktion muss `static` sein, um aufgerufen werden zu können.

Lösung 10: Fehler finden, II

```
public static int doDouble(int x) { return (2*x); }  
public static void main(String[] args) {  
    System.out.print(doDouble(21));  
}
```

1. Der Bezeichner `double` ist für eine Funktion nicht erlaubt.
2. Der Ausdruck in der Funktion ist ungültig geklammert.
3. Die Funktion muss `static` sein, um aufgerufen werden zu können.
4. Die `main`-Methode benötigt die Signatur `main(String[])`!

Lösung 10: Fehler finden, II

```
public static int doDouble(int x) { return (2*x); }  
public static void main(String[] args) {  
    System.out.print(doDouble(21));  
}
```

1. Der Bezeichner `double` ist für eine Funktion nicht erlaubt.
2. Der Ausdruck in der Funktion ist ungültig geklammert.
3. Die Funktion muss `static` sein, um aufgerufen werden zu können.
4. Die `main`-Methode benötigt die Signatur `main(String[])`!
5. `System.in` hat die Funktion `print(String)` nicht.

Lösung 10: Fehler finden, II

```
public static int doDouble(int x) { return (2*x); }  
public static void main(String[] args) {  
    System.out.print(doDouble(21));  
}
```

1. Der Bezeichner `double` ist für eine Funktion nicht erlaubt.
2. Der Ausdruck in der Funktion ist ungültig geklammert.
3. Die Funktion muss `static` sein, um aufgerufen werden zu können.
4. Die `main`-Methode benötigt die Signatur `main(String[])`!
5. `System.in` hat die Funktion `print(String)` nicht.

An der Klasse hat sich nichts geändert.

ÜBUNGSAUFGABE

Aufgabe 11: Signaturen

(2 Minuten)

Aufgabe 11: Signaturen

(2 Minuten)

Geben Sie die Signaturen folgender Funktionen an. Sind die Java-Funktionen ungültig, so geben Sie bitte eine kurze Begründung an:

Aufgabe 11: Signaturen

(2 Minuten)

Geben Sie die Signaturen folgender Funktionen an. Sind die Java-Funktionen ungültig, so geben Sie bitte eine kurze Begründung an:

```
String a(int a1, double a2) { /* ... */ }
int b(String[] b1, int b2, double[] b3) { /* ... */ }
double c(String... c1, char c2) { /* ... */ }
double d(int d1, double... d2) { /* ... */ }
```


Lösung 11: Signaturen

1. `String a(int a1, double a2)` hat die Signatur `a(int, double)`

Lösung 11: Signaturen

1. `String a(int a1, double a2)` hat die Signatur `a(int, double)`
2. `int b(String[] b1, int b2, double[] b3)` hat die Signatur `b(String[], int, double[])`

Lösung 11: Signaturen

1. `String a(int a1, double a2)` hat die Signatur `a(int, double)`
2. `int b(String[] b1, int b2, double[] b3)` hat die Signatur `b(String[], int, double[])`
3. `double c(String... c1, char c2)` ist nicht möglich,

Lösung 11: Signaturen

1. `String a(int a1, double a2)` hat die Signatur `a(int, double)`
2. `int b(String[] b1, int b2, double[] b3)` hat die Signatur `b(String[], int, double[])`
3. `double c(String... c1, char c2)` ist nicht möglich, da nach *varargs* keine verpflichtenden Argumente mehr kommen dürfen.

Lösung 11: Signaturen

1. `String a(int a1, double a2)` hat die Signatur `a(int, double)`
2. `int b(String[] b1, int b2, double[] b3)` hat die Signatur `b(String[], int, double[])`
3. `double c(String... c1, char c2)` ist nicht möglich, da nach *varargs* keine verpflichtenden Argumente mehr kommen dürfen.
4. `double d(int d1, double... d2)` hat die Signatur `d(int, double...)`

ÜBUNGSaufGABE

Aufgabe 12: Konkatenation

(2 Minuten)

Welche Ausgabe erzeugen die folgenden Zeilen (jeweils)?

Aufgabe 12: Konkatenation

(2 Minuten)

Welche Ausgabe erzeugen die folgenden Zeilen (jeweils)?

```
System.out.println("2_+_3_=_ " + 2 + 3);  
System.out.println("-'=' + 2 * 3 + '=' + "_2*3");  
System.out.println(2 - 12 + "_*_2_=_ " + -1_0 * 2 + ("_" + 'b' + 10));  
System.out.println("c_-_a_-_1_=_ " + ('c' - 1 - 'a'));
```


Lösung 12: Konkatenation

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

Lösung 12: Konkatination

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

```
println("2_+_3_=_ " + 2 + 3); // → '2 + 3 = 23'
```


Lösung 12: Konkatenation

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

```
println("2_+_3_=_") + 2 + 3; // → '2 + 3 = 23'
```

Da der linke Operand von `+` ein `String` ist,

Lösung 12: Konkatenation

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

```
println("2_+_3_=_ " + 2 + 3); // → '2 + 3 = 23'
```

Da der linke Operand von `+` ein `String` ist, wird erst die `2` als String konkateniert,

Lösung 12: Konkatenation

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

```
println("2_+_3_=_ " + 2 + 3); // → '2 + 3 = 23'
```

Da der linke Operand von `+` ein `String` ist, wird erst die `2` als String konkateniert, und dann die `3` ebenfalls als Zeichen.

Lösung 12: Konkatenation

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

```
println("2+_3=_ " + 2 + 3); // → '2 + 3 = 23'
```

Da der linke Operand von `+` ein `String` ist, wird erst die `2` als `String` konkateniert, und dann die `3` ebenfalls als Zeichen.

```
println("-'=' + 2 * 3 + '=' + "_2*3"); // → '6 2*3'
```

Lösung 12: Konkatenation

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

```
println("2+_3=_" + 2 + 3); // → '2 + 3 = 23'
```

Da der linke Operand von `+` ein `String` ist, wird erst die `2` als `String` konkateniert, und dann die `3` ebenfalls als Zeichen.

```
println('-=' + 2 * 3 + '=' + "_2*3"); // → '6 2*3'
```

Da die `chars` einfach zu ihrem Integerwert konvertiert werden,

Lösung 12: Konkatination

Hinweis: Ich schreibe `println` kurz für `System.out.println`.

```
println("2+_3=_ " + 2 + 3); // → '2 + 3 = 23'
```

Da der linke Operand von `+` ein `String` ist, wird erst die `2` als String konkatiniert, und dann die `3` ebenfalls als Zeichen.

```
println('-' + 2 * 3 + '=' + "2*3"); // → '6 2*3'
```

Da die `char`s einfach zu ihrem Integerwert konvertiert werden, findet erst beim letzten Plus eine Konkatination statt.

Lösung 12: Konkatenation

(Fortsetzung)

Lösung 12: Konkatenation

(Fortsetzung)

```
println(2 - 12 + "_*_2_=__" + -1_0 * 2 + ("_" + 'b' + 10));  
    // → '-10 * 2 = -20_b10'
```


Lösung 12: Konkatination

(Fortsetzung)

```
println(2 - 12 + "_*_2*_=" + -1_0 * 2 + ("_" + 'b' + 10));  
    // → '-10 * 2 = -20_b10'
```

In der Klammer wird durch erneute Konkatination `_b10` erzeugt. Anschließend greift die Präzedenzregel „Punkt vor Strich“,

Lösung 12: Konkatenation

(Fortsetzung)

```
println(2 - 12 + "_*_2*_=" + -1_0 * 2 + ("_" + 'b' + 10));  
    // → '-10 * 2 = -20_b10'
```

In der Klammer wird durch erneute Konkatenation `_b10` erzeugt. Anschließend greift die Präzedenzregel „Punkt vor Strich“, und liefert $-10 \cdot 2 = -20$.

Lösung 12: Konkatenation

(Fortsetzung)

```
println(2 - 12 + "_*_2*_=" + -1_0 * 2 + ("_" + 'b' + 10));  
    // → '-10 * 2 = -20_b10'
```

In der Klammer wird durch erneute Konkatenation `_b10` erzeugt. Anschließend greift die Präzedenzregel „Punkt vor Strich“, und liefert $-10 \cdot 2 = -20$. Dann liefert die erste arithmetische Operation durch $2 - 12 = -20$.

Lösung 12: Konkatenation

(Fortsetzung)

```
println(2 - 12 + "_*_2*_=" + -1_0 * 2 + ("_" + 'b' + 10));  
    // → '-10 * 2 = -20_b10'
```

In der Klammer wird durch erneute Konkatenation `_b10` erzeugt. Anschließend greift die Präzedenzregel „Punkt vor Strich“, und liefert $-10 \cdot 2 = -20$. Dann liefert die erste arithmetische Operation durch $2 - 12 = -20$. Anschließend findet die String-Konkatenation von links nach rechts statt.

Lösung 12: Konkatenation

(Fortsetzung)

```
println(2 - 12 + "_*_2*_=" + -1_0 * 2 + ("_" + 'b' + 10));  
    // → '-10 * 2 = -20_b10'
```

In der Klammer wird durch erneute Konkatenation `_b10` erzeugt. Anschließend greift die Präzedenzregel „Punkt vor Strich“, und liefert $-10 \cdot 2 = -20$. Dann liefert die erste arithmetische Operation durch $2 - 12 = -20$. Anschließend findet die String-Konkatenation von links nach rechts statt.

Hinweis: Der Unterstrich bei `-10` ist in Java (zwischen Zahlen) komplett valide

Lösung 12: Konkatenation

(Fortsetzung)

```
println(2 - 12 + "_*_2*__" + -1_0 * 2 + ("_" + 'b' + 10));  
    // → '-10 * 2 = -20_b10'
```

In der Klammer wird durch erneute Konkatenation `_b10` erzeugt. Anschließend greift die Präzedenzregel „Punkt vor Strich“, und liefert $-10 \cdot 2 = -20$. Dann liefert die erste arithmetische Operation durch $2 - 12 = -20$. Anschließend findet die String-Konkatenation von links nach rechts statt.

Hinweis: Der Unterstrich bei `-10` ist in Java (zwischen Zahlen) komplett valide und wird in der Regel zum Trennen von 1000er Blöcken verwendet.

Lösung 12: Konkatination

(Fortsetzung)

Lösung 12: Konkatination

(Fortsetzung)

```
println("c - a - 1 = " + ('c' - 1 - 'a'));  
// → 'c - a - 1 = 1'
```


Lösung 12: Konkatination

(Fortsetzung)

```
println("c_a_1=" + ('c' - 1 - 'a'));  
    // → 'c - a - 1 = 1'
```

Java berechnet zuerst die Operationen in der Klammer und wandelt die Zeichen in ihre ASCII-Werte um.

Lösung 12: Konkatination

(Fortsetzung)

```
println("c_a_1" + ('c' - 1 - 'a'));  
    // → 'c - a - 1 = 1'
```

Java berechnet zuerst die Operationen in der Klammer und wandelt die Zeichen in ihre ASCII-Werte um. Diese muss man hierfür nicht kennen,

Lösung 12: Konkatination

(Fortsetzung)

```
println("c - a - 1 = ");  
    // → 'c - a - 1 = 1'
```

Java berechnet zuerst die Operationen in der Klammer und wandelt die Zeichen in ihre ASCII-Werte um. Diese muss man hierfür nicht kennen, da es genügt zu wissen, dass zwischen `a` und `c` ein Abstand von 2 vorliegt, der um eins verringert wird.

Lösung 12: Konkatination

(Fortsetzung)

```
println("c_a_1=" + ('c' - 1 - 'a'));  
    // → 'c - a - 1 = 1'
```

Java berechnet zuerst die Operationen in der Klammer und wandelt die Zeichen in ihre ASCII-Werte um. Diese muss man hierfür nicht kennen, da es genügt zu wissen, dass zwischen `a` und `c` ein Abstand von 2 vorliegt, der um eins verringert wird. Wir berechnen also in der Klammer den Wert 1.

Lösung 12: Konkatination

(Fortsetzung)

```
println("c_a_1=" + ('c' - 1 - 'a'));  
    // → 'c - a - 1 = 1'
```

Java berechnet zuerst die Operationen in der Klammer und wandelt die Zeichen in ihre ASCII-Werte um. Diese muss man hierfür nicht kennen, da es genügt zu wissen, dass zwischen `a` und `c` ein Abstand von 2 vorliegt, der um eins verringert wird. Wir berechnen also in der Klammer den Wert 1. Dieser wird nun normal zum String konkatiniert, was die Ausgabe erzeugt.

ÜBUNGSaufGABE

Aufgabe 13: Javas' Typkonvertierung

(5 Minuten)

Aufgabe 13: Javas' Typkonvertierung

(5 Minuten)

Bitte geben Sie für jeden Ausdruck den Typ des resultierenden Wertes oder Objekts an und erklären Sie kurz

Aufgabe 13: Javas' Typkonvertierung

(5 Minuten)

Bitte geben Sie für jeden Ausdruck den Typ des resultierenden Wertes oder Objekts an und erklären Sie kurz (Tipp: Alle Ausdrücke sind korrekt):

Aufgabe 13: Javas' Typkonvertierung

(5 Minuten)

Bitte geben Sie für jeden Ausdruck den Typ des resultierenden Wertes oder Objekts an und erklären Sie kurz (Tipp: Alle Ausdrücke sind korrekt):

1. `2 + 3`

2. `7 - 2.0`

3. `'x' * (byte) 4`

4. `5L / (char) 2`

5. `(short)3 + (byte)-3.1415`

6. `(float)0b1001 + (long)0x42`

7. `new Scanner(System.in)`

8. `"x" + 3`

Lösung 13: Javas' Typkonvertierung

Lösung 13: Javas' Typkonvertierung

1. „2 + 3“: `int` (5)

Lösung 13: Javas' Typkonvertierung

1. „2 + 3“: `int` (5)

Die Addition zweier Integer liefert wieder einen Integer.

Lösung 13: Javas' Typkonvertierung

1. „`2 + 3`“: `int` (5)

Die Addition zweier Integer liefert wieder einen Integer.

2. „`7 - 2.0`“: `double` (5.0)

Lösung 13: Javas' Typkonvertierung

1. „`2 + 3`“: `int` (5)

Die Addition zweier Integer liefert wieder einen Integer.

2. „`7 - 2.0`“: `double` (5.0)

Die Integer und Double können nicht subtrahiert werden. Deswegen konvertiert Java `7` implizit zu einem `double` und die Differenz liefert wieder ein Double.

Lösung 13: Javas' Typkonvertierung

1. „`2 + 3`“: `int` (5)

Die Addition zweier Integer liefert wieder einen Integer.

2. „`7 - 2.0`“: `double` (5.0)

Die Integer und Double können nicht subtrahiert werden. Deswegen konvertiert Java `7` implizit zu einem `double` und die Differenz liefert wieder ein Double.

3. „`'x' * (byte) 4`“: `int` (480)

Lösung 13: Javas' Typkonvertierung

1. „`2 + 3`“: `int` (5)

Die Addition zweier Integer liefert wieder einen Integer.

2. „`7 - 2.0`“: `double` (5.0)

Die Integer und Double können nicht subtrahiert werden. Deswegen konvertiert Java `7` implizit zu einem `double` und die Differenz liefert wieder ein Double.

3. „`'x' * (byte) 4`“: `int` (480)

Ein Character kann nicht mit einem Byte multipliziert werden. Deswegen konvertiert Java `'x'` zu einem Integer (zugehöriger UTF16-Wert) und anschließend das Byte ebenfalls zu einem Integer.

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

4. „5L/(char) 2“: long (2)

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

4. „5L/(char) 2“: long (2)

5L besitzt den Typ `long`, damit wird auch der Character mit UTF16-Wert 2 in ein `long` konvertiert. Das Ergebnis der Division aus `long` und `long` ergibt wieder `long`.

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

4. „5L/(char) 2“: long (2)

5L besitzt den Typ long, damit wird auch der Character mit UTF16-Wert 2 in ein long konvertiert. Das Ergebnis der Division aus long und long ergibt wieder long.

5. „(short)3 + (byte)-3.1415“: int (0)

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

4. „`5L/(char) 2`“: `long` (2)

`5L` besitzt den Typ `long`, damit wird auch der Character mit UTF16-Wert 2 in ein `long` konvertiert. Das Ergebnis der Division aus `long` und `long` ergibt wieder `long`.

5. „`(short)3 + (byte)-3.1415`“: `int` (0)

Java konvertiert die Summe aus einem Short und einem Byte implizit zu einem Integer.

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

4. „5L/(char) 2“: long (2)

5L besitzt den Typ long, damit wird auch der Character mit UTF16-Wert 2 in ein long konvertiert. Das Ergebnis der Division aus long und long ergibt wieder long.

5. „(short)3 + (byte)-3.1415“: int (0)

Java konvertiert die Summe aus einem Short und einem Byte implizit zu einem Integer.

6. „(float)0b1001 + (long)0x42“: float (75.0)

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

4. „5L/(char) 2“: long (2)

5L besitzt den Typ `long`, damit wird auch der Character mit UTF16-Wert 2 in ein `long` konvertiert. Das Ergebnis der Division aus `long` und `long` ergibt wieder `long`.

5. „(short)3 + (byte)-3.1415“: int (0)

Java konvertiert die Summe aus einem Short und einem Byte implizit zu einem Integer.

6. „(float)0b1001 + (long)0x42“: float (75.0)

Java konvertiert die Summe aus einem Float und einem Long zu einem Float (da Float die Werte von Long inkludiert).

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

7. „**new** `Scanner(System.in)`“: `Scanner` (oder voll: `java.util.Scanner`)

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

7. „**new** `Scanner(System.in)`“: `Scanner` (oder voll: `java.util.Scanner`)
Mit **new** wird ein neues Objekt der angegebenen Klasse erstellt.

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

7. „**new** `Scanner(System.in)`“: `Scanner` (oder voll: `java.util.Scanner`)
Mit **new** wird ein neues Objekt der angegebenen Klasse erstellt.
8. „`"x" + 3`“: `String` (x3)

Lösung 13: Javas' Typkonvertierung

(Fortsetzung)

7. „**new** **Scanner**(**System.in**)“: **Scanner** (oder voll: `java.util.Scanner`)
Mit **new** wird ein neues Objekt der angegebenen Klasse erstellt.
8. „**"x"** + **3**“: **String** (x3)
Analog zur Aufgabe zur String-Konkatenation wird hier **3** in eine Zeichenkette konvertiert.

Kontrollstrukturen und Arrays



FALLUNTERSCHIEDUNG MIT IF

FALLUNTERSCHIEDUNG MIT IF

- Java erlaubt Fallunterscheidungen mit **if**:

FALLUNTERSCHIEDUNG MIT IF

- Java erlaubt Fallunterscheidungen mit **if**:

```
if(⟨Bedingung 1⟩) {  
    // Bedingung 1 ist wahr  
} else if (⟨Bedingung 2⟩) {  
    // Bedingung 1 ist falsch, Bedingung 2 ist wahr  
} else {  
    // Beide Bedingungen sind falsch  
}
```

FALLUNTERSCHIEDUNG MIT IF

- Java erlaubt Fallunterscheidungen mit **if**:

```
if(⟨Bedingung 1⟩) {  
    // Bedingung 1 ist wahr  
} else if (⟨Bedingung 2⟩) {  
    // Bedingung 1 ist falsch, Bedingung 2 ist wahr  
} else {  
    // Beide Bedingungen sind falsch  
}
```

- Folgt der **if**-Instruktion nur eine Anweisung,

FALLUNTERSCHIEDUNG MIT IF

- Java erlaubt Fallunterscheidungen mit **if**:

```
if(<Bedingung 1>) {  
    // Bedingung 1 ist wahr  
} else if (<Bedingung 2>) {  
    // Bedingung 1 ist falsch, Bedingung 2 ist wahr  
} else {  
    // Beide Bedingungen sind falsch  
}
```

- Folgt der **if**-Instruktion nur eine Anweisung, so können die geschwungenen Klammern weggelassen werden.

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:
〈Bedingung〉 ? 〈Bedingung-wahr〉 : 〈Bedingung-falsch〉

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

$\langle \text{Bedingung} \rangle ? \langle \text{Bedingung-wahr} \rangle : \langle \text{Bedingung-falsch} \rangle$

Trifft die Bedingung zu, wird der *wahr*-Teil, sonst der *falsch*-Teil ausgeführt.

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

$\langle \text{Bedingung} \rangle ? \langle \text{Bedingung-wahr} \rangle : \langle \text{Bedingung-falsch} \rangle$

Trifft die Bedingung zu, wird der *wahr*-Teil, sonst der *falsch*-Teil ausgeführt.

- Ein Beispiel:

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

`<Bedingung> ? <Bedingung-wahr> : <Bedingung-falsch>`

Trifft die Bedingung zu, wird der *wahr*-Teil, sonst der *falsch*-Teil ausgeführt.

- Ein Beispiel:

```
int n = 14;  
return (n >= 18) ? "Volljährig" : "Nicht_volljährig";
```

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

`<Bedingung> ? <Bedingung-wahr> : <Bedingung-falsch>`

Trifft die Bedingung zu, wird der *wahr*-Teil, sonst der *falsch*-Teil ausgeführt.

- Ein Beispiel:

```
int n = 14;
```

```
return (n >= 18) ? "Volljährig" : "Nicht_volljährig";
```

> Liefert: "Nicht_volljährig".

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

`<Bedingung> ? <Bedingung-wahr> : <Bedingung-falsch>`

Trifft die Bedingung zu, wird der *wahr*-Teil, sonst der *falsch*-Teil ausgeführt.

- Ein Beispiel:

```
int n = 14;
```

```
return (n >= 18) ? "Volljährig" : "Nicht_volljährig";
```

> Liefert: "Nicht_volljährig".

- Diese Anweisung kann (beliebig tief) verschachtelt werden.

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch( <Ausdruck >){
```

```
}
```


FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
  
}
```

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
    case ⟨Fall 3⟩:  
        // Code für Fall 3  
        break;  
  
}
```

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
    case ⟨Fall 3⟩:  
        // Code für Fall 3  
        break;  
    default: // Code, wenn keiner der Fälle greift.  
}
```

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
    case ⟨Fall 3⟩:  
        // Code für Fall 3  
        break;  
    default: // Code, wenn keiner der Fälle greift.  
}
```

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
    case ⟨Fall 3⟩:  
        // Code für Fall 3  
        break;  
    default: // Code, wenn keiner der Fälle greift.  
}
```

- Dies funktioniert für:

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
    case ⟨Fall 3⟩:  
        // Code für Fall 3  
        break;  
    default: // Code, wenn keiner der Fälle greift.  
}
```

- Dies funktioniert für: Zahlen (also auch **char**, was konvertiert werden kann),

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
    case ⟨Fall 3⟩:  
        // Code für Fall 3  
        break;  
    default: // Code, wenn keiner der Fälle greift.  
}
```

- Dies funktioniert für: Zahlen (also auch **char**, was konvertiert werden kann), **enum**-Konstanten

FALLUNTERSCHIEDUNG MIT SWITCH-CASE

- Java erlaubt Fallunterscheidungen mit **switch-case**:

```
switch(⟨Ausdruck⟩){  
    case ⟨Fall 1⟩: // Code für Fall 1  
    case ⟨Fall 2⟩:  
        // Code für Fall 1 & 2  
        break; // Verlässt Anweisung  
    case ⟨Fall 3⟩:  
        // Code für Fall 3  
        break;  
    default: // Code, wenn keiner der Fälle greift.  
}
```

- Dies funktioniert für: Zahlen (also auch **char**, was konvertiert werden kann), **enum**-Konstanten und (seit Java Version 7) auch für **Strings**.

WHILE

WHILE

- Schleifen erlauben es in Java bestimmte Programmanweisungen mehrfach auszuführen:

WHILE

- Schleifen erlauben es in Java bestimmte Programmanweisungen mehrfach auszuführen:

```
while(⟨Bedingung⟩) {  
    ⟨Anweisung(en)⟩  
}
```

WHILE

- Schleifen erlauben es in Java bestimmte Programmanweisungen mehrfach auszuführen:

```
while(⟨Bedingung⟩) {  
    ⟨Anweisung(en)⟩  
}
```

Die Schleife wird so lange ausgeführt, wie die **Bedingung** *wahr* ist.

WHILE UND DO-WHILE

WHILE UND DO-WHILE

- Im Gegensatz zur **while**-Schleife führt die **do-while**-Schleife die Prüfung am Ende des Durchlaufs durch.

WHILE UND DO-WHILE

- Im Gegensatz zur **while**-Schleife führt die **do-while**-Schleife die Prüfung am Ende des Durchlaufs durch. Sie wird also mindestens einmal durchgeführt,

WHILE UND DO-WHILE

- Im Gegensatz zur **while**-Schleife führt die **do-while**-Schleife die Prüfung am Ende des Durchlaufs durch. Sie wird also mindestens einmal durchgeführt, auch wenn die Bedingung von Beginn an *falsch* ist:

```
do {  
    <Anweisung(en)>  
} while (<Bedingung>); // ← Semikolon!
```

WHILE UND DO-WHILE

- Im Gegensatz zur **while**-Schleife führt die **do-while**-Schleife die Prüfung am Ende des Durchlaufs durch. Sie wird also mindestens einmal durchgeführt, auch wenn die Bedingung von Beginn an *falsch* ist:

```
do {  
    <Anweisung(en)>  
} while (<Bedingung>); // <- Semikolon!
```

- Wichtig ist das Semikolon am Ende der **do-while**-Anweisung. Es wird gern vergessen.

ÜBUNGSaufgabe

Aufgabe 14: While vs. Do-While, I

(2 Minuten)

Aufgabe 14: While vs. Do-While, I

(2 Minuten)

Erzeugen die Schleifen für $i = 0$ die selbe Ausgabe? Was ändert sich, wenn $i = 42$ anstelle von $i = 0$?

Aufgabe 14: While vs. Do-While, I

(2 Minuten)

Erzeugen die Schleifen für $i = 0$ die selbe Ausgabe? Was ändert sich, wenn $i = 42$ anstelle von $i = 0$?

```
int i = {?}; /* a) 0, b) 42 */  
do {  
    System.out.print(i++);  
} while(i < 5);
```

```
int i = {?}; /* a) 0, b) 42 */  
while(i < 5) {  
    System.out.print(i++);  
}
```


Lösung 14: While vs. Do-While, I

- Für $i = 0$ produzieren beide Schleifen „01234“ und brechen dann für $5 < 5$ ab.

Lösung 14: While vs. Do-While, I

- Für $i = 0$ produzieren beide Schleifen „01234“ und brechen dann für $5 < 5$ ab.
- Für $i = 42$ unterscheiden sich die Ausgaben. Hier gibt do-while noch „42“ aus, während die while Schleife gar nicht erst betreten wird.

ÜBUNGSAUFGABE

Aufgabe 15: Schleifen umwandeln

(5 Minuten)

Aufgabe 15: Schleifen umwandeln

(5 Minuten)

Schreiben Sie die folgende Funktion so um, dass sie einmal a) nur **for**- und einmal b) nur **do-while**-Schleifen anstelle der **while**-Schleife benutzt, aber immer noch die selbe Ergebnisse produziert.

Aufgabe 15: Schleifen umwandeln

(5 Minuten)

Schreiben Sie die folgende Funktion so um, dass sie einmal a) nur **for**- und einmal b) nur **do-while**-Schleifen anstelle der **while**-Schleife benutzt, aber immer noch die selbe Ergebnisse produziert.

```
static int calc(int k, int[] ms) {  
    int i = 0;  
    while(i >= 0 && i < ms.length && ms[i] < k) {  
        i += ms[i];  
    }  
    return i;  
}
```


Lösung 15: Schleifen umwandeln

Lösung 15: Schleifen umwandeln

- Für die **for**-Schleife:

Lösung 15: Schleifen umwandeln

- Für die **for**-Schleife:

```
static int calc(int k, int[] ms) {  
    int i = 0;  
  
}
```

Lösung 15: Schleifen umwandeln

- Für die **for**-Schleife:

```
static int calc(int k, int[] ms) {  
    int i = 0;  
    for (; i >= 0 && i < ms.length && ms[i] < k; i += ms[i]) ;  
  
}
```

Lösung 15: Schleifen umwandeln

- Für die **for**-Schleife:

```
static int calc(int k, int[] ms) {  
    int i = 0;  
    for (; i >= 0 && i < ms.length && ms[i] < k; i += ms[i]) ;  
    return i;  
}
```

Lösung 15: Schleifen umwandeln

- Für die **for**-Schleife:

```
static int calc(int k, int[] ms) {  
    int i = 0;  
    for (; i >= 0 && i < ms.length && ms[i] < k; i += ms[i]) ;  
    return i;  
}
```

Alternativ kann man `i += ms[i]` auch im Body der **for**-Schleife ausführen.

Lösung 15: Schleifen umwandeln

(Fortsetzung)

Lösung 15: Schleifen umwandeln

(Fortsetzung)

- Bei **do-while** müssen wir uns um die erste Iteration kümmern:

Lösung 15: Schleifen umwandeln

(Fortsetzung)

- Bei **do-while** müssen wir uns um die erste Iteration kümmern:

```
static int calcDoWhile(int k, int[] ms) {
```

```
}
```


Lösung 15: Schleifen umwandeln

(Fortsetzung)

- Bei **do-while** müssen wir uns um die erste Iteration kümmern:

```
static int calcDoWhile(int k, int[] ms) {  
    int i = 0;  
    if (i >= ms.length || ms[i] >= k) // i >= 0 immer erfüllt  
        return i; // Überhaupt kein Durchlauf?  
  
}
```

Lösung 15: Schleifen umwandeln

(Fortsetzung)

- Bei **do-while** müssen wir uns um die erste Iteration kümmern:

```
static int calcDoWhile(int k, int[] ms) {  
    int i = 0;  
    if (i >= ms.length || ms[i] >= k) // i >= 0 immer erfüllt  
        return i; // Überhaupt kein Durchlauf?  
    do {  
  
    } while (i >= 0 && i < ms.length && ms[i] < k);  
  
}
```

Lösung 15: Schleifen umwandeln

(Fortsetzung)

- Bei **do-while** müssen wir uns um die erste Iteration kümmern:

```
static int calcDoWhile(int k, int[] ms) {  
    int i = 0;  
    if (i >= ms.length || ms[i] >= k) // i >= 0 immer erfüllt  
        return i; // Überhaupt kein Durchlauf?  
    do {  
        i += ms[i];  
    } while (i >= 0 && i < ms.length && ms[i] < k);  
  
}
```

Lösung 15: Schleifen umwandeln

(Fortsetzung)

- Bei **do-while** müssen wir uns um die erste Iteration kümmern:

```
static int calcDoWhile(int k, int[] ms) {  
    int i = 0;  
    if (i >= ms.length || ms[i] >= k) // i >= 0 immer erfüllt  
        return i; // Überhaupt kein Durchlauf?  
    do {  
        i += ms[i];  
    } while (i >= 0 && i < ms.length && ms[i] < k);  
    return i;  
}
```


ZÄHLSCHLEIFEN MIT FOR

ZÄHLSCHLEIFEN MIT FOR

- Die **for**-Schleife in Java besteht aus drei optionalen Komponenten,

ZÄHLSCHLEIFEN MIT FOR

- Die **for**-Schleife in Java besteht aus drei optionalen Komponenten, die alle leer sein können (⟨Start⟩, ⟨Schleifenbedingung⟩ und ⟨Nach⟩):

```
for(⟨Start⟩; ⟨Schleifenbedingung⟩; ⟨Nach⟩) {  
    ⟨Anweisung(en)⟩  
}
```

ZÄHLSCHLEIFEN MIT FOR

- Die **for**-Schleife in Java besteht aus drei optionalen Komponenten, die alle leer sein können (`<Start >`, `<Schleifenbedingung >` und `<Nach >`):

```
for(<Start >; <Schleifenbedingung >; <Nach >) {  
    <Anweisung(en) >  
}
```

- Betrachten wir ein Beispiel:

ZÄHLSCHLEIFEN MIT FOR

- Die **for**-Schleife in Java besteht aus drei optionalen Komponenten, die alle leer sein können (`<Start>`, `<Schleifenbedingung>` und `<Nach>`):

```
for(<Start>; <Schleifenbedingung>; <Nach>) {  
    <Anweisung(en)>  
}
```

- Betrachten wir ein Beispiel:

```
for(int i = 0, k = 1; i < 20; i++, k *= 2)  
    System.out.println(i + ":\u0304" + k);
```

ZÄHLSCHLEIFEN MIT FOR

- Die **for**-Schleife in Java besteht aus drei optionalen Komponenten, die alle leer sein können (`<Start>`, `<Schleifenbedingung>` und `<Nach>`):

```
for(<Start>; <Schleifenbedingung>; <Nach>) {  
    <Anweisung(en)>  
}
```

- Betrachten wir ein Beispiel:

```
for(int i = 0, k = 1; i < 20; i++, k *= 2)  
    System.out.println(i + ":\u25b6" + k);
```

Diese Schleife liefert im i -ten Durchlauf $k = 2^i$,

ZÄHLSCHLEIFEN MIT FOR

- Die **for**-Schleife in Java besteht aus drei optionalen Komponenten, die alle leer sein können (`⟨Start⟩`, `⟨Schleifenbedingung⟩` und `⟨Nach⟩`):

```
for(⟨Start⟩; ⟨Schleifenbedingung⟩; ⟨Nach⟩) {  
    ⟨Anweisung(en)⟩  
}
```

- Betrachten wir ein Beispiel:

```
for(int i = 0, k = 1; i < 20; i++, k *= 2)  
    System.out.println(i + " : " + k);
```

Diese Schleife liefert im i -ten Durchlauf $k = 2^i$, bis zu $i = 19$.

ZÄHLSCHLEIFEN MIT FOR

- Die **for**-Schleife in Java besteht aus drei optionalen Komponenten, die alle leer sein können (`<Start>`, `<Schleifenbedingung>` und `<Nach>`):

```
for(<Start>; <Schleifenbedingung>; <Nach>) {  
    <Anweisung(en)>  
}
```

- Betrachten wir ein Beispiel:

```
for(int i = 0, k = 1; i < 20; i++, k *= 2)  
    System.out.println(i + ":\u25b6" + k);
```

Diese Schleife liefert im i -ten Durchlauf $k = 2^i$, bis zu $i = 19$.

- Hinweis: `for(;;) { /* ... */ }` ist eine Endlosschleife.

BREAK UND CONTINUE

BREAK UND CONTINUE

- Das Schlüsselwort **break** bricht die „innerste“ Schleife ab.

BREAK UND CONTINUE

- Das Schlüsselwort **break** bricht die „innerste“ Schleife ab.
- Mit **continue** wird nur der aktuelle Durchlauf abgebrochen und,

BREAK UND CONTINUE

- Das Schlüsselwort **break** bricht die „innerste“ Schleife ab.
- Mit **continue** wird nur der aktuelle Durchlauf abgebrochen und, zum Beispiel in einer **for**-Schleife,

BREAK UND CONTINUE

- Das Schlüsselwort **break** bricht die „innerste“ Schleife ab.
- Mit **continue** wird nur der aktuelle Durchlauf abgebrochen und, zum Beispiel in einer **for**-Schleife, das Inkrement durchgeführt:

BREAK UND CONTINUE

- Das Schlüsselwort **break** bricht die „innerste“ Schleife ab.
- Mit **continue** wird nur der aktuelle Durchlauf abgebrochen und, zum Beispiel in einer **for**-Schleife, das Inkrement durchgeführt:

```
for(int i = 0, k = 1; i < 20; i++, k *= 2) {  
    if(i % 2 == 0) continue;  
    System.out.println(i + " : " + k);  
}
```

BREAK UND CONTINUE

- Das Schlüsselwort **break** bricht die „innerste“ Schleife ab.
- Mit **continue** wird nur der aktuelle Durchlauf abgebrochen und, zum Beispiel in einer **for**-Schleife, das Inkrement durchgeführt:

```
for(int i = 0, k = 1; i < 20; i++, k *= 2) {  
    if(i % 2 == 0) continue;  
    System.out.println(i + ": " + k);  
}
```

Alle geraden *i*-Werte werden übersprungen, die Ausgabe erfolgt nicht.

ITERIEREN MIT FOREACH

ITERIEREN MIT FOREACH

- Mit Java Version 5 gibt es eine weitere Variante der **for**-Schleife:

ITERIEREN MIT FOREACH

- Mit Java Version 5 gibt es eine weitere Variante der **for**-Schleife:

```
for(⟨Variable-Deklaration⟩ : ⟨Iterierbare Variable⟩) {  
    // Etwas mit der Variable machen  
}
```

ITERIEREN MIT FOREACH

- Mit Java Version 5 gibt es eine weitere Variante der **for**-Schleife:

```
for(⟨Variable-Deklaration⟩ : ⟨Iterierbare Variable⟩) {  
    // Etwas mit der Variable machen  
}
```

Hinweis: Ohne Wissen über Interfaces ist es schwer zu „erfassen“, welche Datentypen erlaubt sind.

ITERIEREN MIT FOREACH

- Mit Java Version 5 gibt es eine weitere Variante der **for**-Schleife:

```
for(⟨Variable-Deklaration⟩ : ⟨Iterierbare Variable⟩) {  
    // Etwas mit der Variable machen  
}
```

Hinweis: Ohne Wissen über Interfaces ist es schwer zu „erfassen“, welche Datentypen erlaubt sind. Darunter: Arrays, Listen, ...

ITERIEREN MIT FOREACH

- Mit Java Version 5 gibt es eine weitere Variante der **for**-Schleife:

```
for(⟨Variable-Deklaration⟩ : ⟨Iterierbare Variable⟩) {  
    // Etwas mit der Variable machen  
}
```

Hinweis: Ohne Wissen über Interfaces ist es schwer zu „erfassen“, welche Datentypen erlaubt sind. Darunter: Arrays, Listen, ...

- Ein Beispiel:

ITERIEREN MIT FOREACH

- Mit Java Version 5 gibt es eine weitere Variante der **for**-Schleife:

```
for(⟨Variable-Deklaration⟩ : ⟨Iterierbare Variable⟩) {  
    // Etwas mit der Variable machen  
}
```

Hinweis: Ohne Wissen über Interfaces ist es schwer zu „erfassen“, welche Datentypen erlaubt sind. Darunter: Arrays, Listen, ...

- Ein Beispiel:

```
int[] dinge = new int[] {42, 21, 12, -4};  
for(int ding : dinge){  
    System.out.println(ding);  
}
```

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur,

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur, die aus mehreren Elementen des gleichen Typs aufgebaut ist.

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur, die aus mehreren Elementen des gleichen Typs aufgebaut ist.
- Der Index eines Arrays beginnt bei 0.

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur, die aus mehreren Elementen des gleichen Typs aufgebaut ist.
- Der Index eines Arrays beginnt bei 0.
- Die Länge eines Arrays ist *fest*:

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur, die aus mehreren Elementen des gleichen Typs aufgebaut ist.
- Der Index eines Arrays beginnt bei 0.
- Die Länge eines Arrays ist *fest*:

```
double[] a = new double[42]; // 42 Elemente, alle: 0.0d
int[] b = {2, 4, 6, 8, 10}; // 5 Elemente
char[] c = new char[] {'a', 'z', '9'}; // 3 Elemente
```

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur, die aus mehreren Elementen des gleichen Typs aufgebaut ist.
- Der Index eines Arrays beginnt bei 0.
- Die Länge eines Arrays ist *fest*:

```
double[] a = new double[42]; // 42 Elemente, alle: 0.0d  
int[] b = {2, 4, 6, 8, 10}; // 5 Elemente  
char[] c = new char[] {'a', 'z', '9'}; // 3 Elemente
```

- Die Länge erhalten wir durch `<array>.length`.

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur, die aus mehreren Elementen des gleichen Typs aufgebaut ist.
- Der Index eines Arrays beginnt bei 0.
- Die Länge eines Arrays ist *fest*:

```
double[] a = new double[42]; // 42 Elemente, alle: 0.0d
int[] b = {2, 4, 6, 8, 10}; // 5 Elemente
char[] c = new char[] {'a', 'z', '9'}; // 3 Elemente
```

- Die Länge erhalten wir durch `<array>.length`. Dies ist (im Gegensatz zu `<string>.length()`) *keine* Methode!

ARRAYS, TECHNISCHER HINTERGRUND

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert.

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert. Das bedeutet, ein Array aus 12 `int`-Elementen nimmt einen Speicherblock von $12 \cdot 32$ bit ein.

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert. Das bedeutet, ein Array aus 12 `int`-Elementen nimmt einen Speicherblock von $12 \cdot 32$ bit ein.
- Greift man auf einen nicht erlaubten Index zu,

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert. Das bedeutet, ein Array aus 12 `int`-Elementen nimmt einen Speicherblock von $12 \cdot 32$ bit ein.
- Greift man auf einen nicht erlaubten Index zu, so wird eine `ArrayIndexOutOfBoundsException` geworfen.

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert. Das bedeutet, ein Array aus 12 `int`-Elementen nimmt einen Speicherblock von $12 \cdot 32$ bit ein.
- Greift man auf einen nicht erlaubten Index zu, so wird eine `ArrayIndexOutOfBoundsException` geworfen.
- Da Arrays eine komplexe Datenstruktur sind, wird bei der Erstellung in der Variable selbst nur die Speicheradressen abgelegt.

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert. Das bedeutet, ein Array aus 12 `int`-Elementen nimmt einen Speicherblock von $12 \cdot 32$ bit ein.
- Greift man auf einen nicht erlaubten Index zu, so wird eine `ArrayIndexOutOfBoundsException` geworfen.
- Da Arrays eine komplexe Datenstruktur sind, wird bei der Erstellung in der Variable selbst nur die Speicheradressen abgelegt.

```
int[] a = {2, 4, 6, 8, 10}, b = a;
```

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert. Das bedeutet, ein Array aus 12 `int`-Elementen nimmt einen Speicherblock von $12 \cdot 32$ bit ein.
- Greift man auf einen nicht erlaubten Index zu, so wird eine `ArrayIndexOutOfBoundsException` geworfen.
- Da Arrays eine komplexe Datenstruktur sind, wird bei der Erstellung in der Variable selbst nur die Speicheradressen abgelegt.

```
int[] a = {2, 4, 6, 8, 10}, b = a;
```

Hier verweisen beide Variablen auf denselben Datensatz:

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert. Das bedeutet, ein Array aus 12 `int`-Elementen nimmt einen Speicherblock von $12 \cdot 32$ bit ein.
- Greift man auf einen nicht erlaubten Index zu, so wird eine `ArrayIndexOutOfBoundsException` geworfen.
- Da Arrays eine komplexe Datenstruktur sind, wird bei der Erstellung in der Variable selbst nur die Speicheradressen abgelegt.

```
int[] a = {2, 4, 6, 8, 10}, b = a;
```

Hier verweisen beide Variablen auf denselben Datensatz:

```
a[4] = 42; b[2] = 7;
```

```
for(int i = 0; i < a.length; i++)
```

```
    System.out.print(a[i] + "_"); // → 2 4 7 8 42
```

ÜBER ARRAYS ITERIEREN

ÜBER ARRAYS ITERIEREN

- Wir können,

ÜBER ARRAYS ITERIEREN

- Wir können, mit einer for-Schleife über ein Array iterieren:

ÜBER ARRAYS ITERIEREN

- Wir können, mit einer for-Schleife über ein Array iterieren:

```
for(int i = 0; i < a.length; i++) {  
    System.out.print(a[i] + "_"); // → 2 4 7 8 42  
}
```

ÜBER ARRAYS ITERIEREN

- Wir können, mit einer for-Schleife über ein Array iterieren:

```
for(int i = 0; i < a.length; i++) {  
    System.out.print(a[i] + "_"); // → 2 4 7 8 42  
}
```

- Oder mit „for-each“:

ÜBER ARRAYS ITERIEREN

- Wir können, mit einer for-Schleife über ein Array iterieren:

```
for(int i = 0; i < a.length; i++) {  
    System.out.print(a[i] + "_"); // → 2 4 7 8 42  
}
```

- Oder mit „for-each“:

```
for(int i : a) {  
    System.out.print(i + "_"); // → 2 4 7 8 42  
}
```

ÜBUNGSAUFGABE

Aufgabe 16: Arrays halbieren

(3 Minuten)

Gegeben ein Array `int[] arr` *variabler* Länge. Schreiben Sie Java-Code, der ein neues Array `int[] newArr` erstellt und so befüllt, dass es nur noch die Elemente mit geradem Index aus `arr` enthält.

Beispiele:

Aufgabe 16: Arrays halbieren

(3 Minuten)

Gegeben ein Array `int[] arr` *variabler* Länge. Schreiben Sie Java-Code, der ein neues Array `int[] newArr` erstellt und so befüllt, dass es nur noch die Elemente mit geradem Index aus `arr` enthält.

Beispiele:

`{7, 23, 15, 8, 10, 2}` \Rightarrow `{7, 15, 10}`

`{1, 2, 3}` \Rightarrow `{1, 3}`

`{31}` \Rightarrow `{31}`

Lösung 16: Arrays halbieren

Lösung 16: Arrays halbieren

Zunächst müssen wir die Größe des neuen Arrays berechnen:

Lösung 16: Arrays halbieren

Zunächst müssen wir die Größe des neuen Arrays berechnen:

```
int[] newArr = new int[(arr.length + 1)/2];
```

Lösung 16: Arrays halbieren

Zunächst müssen wir die Größe des neuen Arrays berechnen:

```
int[] newArr = new int[(arr.length + 1)/2];
```

Sauberer (aber nicht gefordert) wäre:

Lösung 16: Arrays halbieren

Zunächst müssen wir die Größe des neuen Arrays berechnen:

```
int[] newArr = new int[(arr.length + 1)/2];
```

Sauberer (aber nicht gefordert) wäre:

```
int[] newArr = new int[(int) Math.ceil(arr.length/2.0)];
```

Lösung 16: Arrays halbieren

Zunächst müssen wir die Größe des neuen Arrays berechnen:

```
int[] newArr = new int[(arr.length + 1)/2];
```

Sauberer (aber nicht gefordert) wäre:

```
int[] newArr = new int[(int) Math.ceil(arr.length/2.0)];
```

Unabhängig davon, der Übertrag der Elemente:

Lösung 16: Arrays halbieren

Zunächst müssen wir die Größe des neuen Arrays berechnen:

```
int[] newArr = new int[(arr.length + 1)/2];
```

Sauberer (aber nicht gefordert) wäre:

```
int[] newArr = new int[(int) Math.ceil(arr.length/2.0)];
```

Unabhängig davon, der Übertrag der Elemente:

```
for(int i = 0; i < arr.length; i += 2) {  
    newArr[i/2] = arr[i];  
}
```


MEHRDIMENSIONALE ARRAYS

MEHRDIMENSIONALE ARRAYS

- Wir sind in der Lage, mehrdimensionale Arrays zu erstellen:

MEHRDIMENSIONALE ARRAYS

- Wir sind in der Lage, mehrdimensionale Arrays zu erstellen:

```
double[][][] a = new double[2][4][6];  
int[][] b = {{1,2}, {2,3}, {1,2,4,5}};
```

MEHRDIMENSIONALE ARRAYS

- Wir sind in der Lage, mehrdimensionale Arrays zu erstellen:

```
double[][][] a = new double[2][4][6];  
int[][] b = {{1,2}, {2,3}, {1,2,4,5}};
```

- Wir erstellen also ein Array von Arrays von Arrays von ...

MEHRDIMENSIONALE ARRAYS

- Wir sind in der Lage, mehrdimensionale Arrays zu erstellen:

```
double[][][] a = new double[2][4][6];  
int[][] b = {{1,2}, {2,3}, {1,2,4,5}};
```

- Wir erstellen also ein Array von Arrays von Arrays von ...
- Der Zugriff auf ein spezifisches Element erfolgt über die Angaben mehrerer Indizes:

```
int x = b[2][3]; // x → 5
```

VERSCHACHTELTE FOR-SCHLEIFEN

VERSCHACHTELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

VERSCHACHELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

```
public static void printMatrix(int[][] m){
```

```
}
```


VERSCHACHELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

```
public static void printMatrix(int[][] m){  
    for(int row = 0; row < m.length; row++) {  
  
    }  
}
```

VERSCHACHELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

```
public static void printMatrix(int[][] m){
    for(int row = 0; row < m.length; row++) {
        for(int col = 0; col < m[row].length; col++) {

        }

    }
}
```

VERSCHACHELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

```
public static void printMatrix(int[][] m){
    for(int row = 0; row < m.length; row++) {
        for(int col = 0; col < m[row].length; col++) {
            System.out.print(m[row][col] + " ");
        }
    }
}
```

VERSCHACHELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

```
public static void printMatrix(int[][] m){
    for(int row = 0; row < m.length; row++) {
        for(int col = 0; col < m[row].length; col++) {
            System.out.print(m[row][col] + " ");
        }
        System.out.println();
    }
}
```

VERSCHACHELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

```
public static void printMatrix(int[][] m){
    for(int row = 0; row < m.length; row++) {
        for(int col = 0; col < m[row].length; col++) {
            System.out.print(m[row][col] + " ");
        }
        System.out.println();
    }
}
```

Aufgabe 17: Iteration mit for-each

(4 Minuten)

Schreiben Sie diese Java-Funktion so um, dass sie ausschließlich den for-each Mechanismus zur Iteration verwendet.

VERSCHACHTELTE FOR-SCHLEIFEN – LÖSUNG

Lösung 17: Iteration mit for-each

Lösung 17: Iteration mit for-each

```
public static void printMatrix(int[][] m){  
    for(int[] row : m) {  
  
  
  
  
  
  
    }  
}
```

Lösung 17: Iteration mit for-each

```
public static void printMatrix(int[][] m){  
    for(int[] row : m) {  
        for(int col : row) {  
  
            }  
  
        }  
  
    }  
}
```

Lösung 17: Iteration mit for-each

```
public static void printMatrix(int[][] m){
    for(int[] row : m) {
        for(int col : row) {
            System.out.print(col + "_");
        }
    }
}
```

Lösung 17: Iteration mit for-each

```
public static void printMatrix(int[][] m){
    for(int[] row : m) {
        for(int col : row) {
            System.out.print(col + "_");
        }
        System.out.println();
    }
}
```

ÜBUNGSaufGABE

Aufgabe 18: Fehler finden, III

(4 Minuten)

Finden und korrigieren Sie alle Kompilier- und Laufzeitfehler:

Aufgabe 18: Fehler finden, III

(4 Minuten)

Finden und korrigieren Sie alle Kompilier- und Laufzeitfehler:

```
int[][][] x = {{}, {{1,2},{2},{5}}, {{3}}};
foreach (int[][] 'ex' : x) {
    int[] eex = ex[0];
    for(int i = 0; i < ex.length; i++)
        System.out.print(eex[i] + " ");
}
```

Wie lautet die Ausgabe Ihres korrigierten Codes?

Lösung 18: Fehler finden, III

```
for (int[][] ex : x) {  
    if(ex.length == 0) continue;  
    int[] eex = ex[0];  
    for(int i = 0; i < eex.length; i++)  
        System.out.print(eex[i] + " ");  
}
```

Lösung 18: Fehler finden, III

```
for (int[][] ex : x) {  
    if(ex.length == 0) continue;  
    int[] eex = ex[0];  
    for(int i = 0; i < eex.length; i++)  
        System.out.print(eex[i] + " ");  
}
```

1. `foreach` ist kein gültiger Schlüsselbegriff, weiter darf ein Variablenbezeichner keine Anführungszeichen enthalten.

Lösung 18: Fehler finden, III

```
for (int[][] ex : x) {  
    if(ex.length == 0) continue;  
    int[] eex = ex[0];  
    for(int i = 0; i < eex.length; i++)  
        System.out.print(eex[i] + " ");  
}
```

1. `foreach` ist kein gültiger Schlüsselbegriff, weiter darf ein Variablenbezeichner keine Anführungszeichen enthalten.
2. Die folgende Zuweisung scheitert, wenn `ex` kein nulltes Element besitzt.

Lösung 18: Fehler finden, II

(Fortsetzung)

```
for (int[][] ex : x) {  
    if(ex.length == 0) continue;  
    int[] eex = ex[0];  
    for(int i = 0; i < eex.length; i++)  
        System.out.print(eex[i] + " ");  
}
```

Lösung 18: Fehler finden, III

(Fortsetzung)

```
for (int[][] ex : x) {  
    if(ex.length == 0) continue;  
    int[] eex = ex[0];  
    for(int i = 0; i < eex.length; i++)  
        System.out.print(eex[i] + " ");  
}
```

3. Die innere **for**-Schleife prüft weiterhin auf die Länge eines anderen Arrays als das es zugreift.

Lösung 18: Fehler finden, III

(Fortsetzung)

```
for (int[][] ex : x) {  
    if(ex.length == 0) continue;  
    int[] eex = ex[0];  
    for(int i = 0; i < eex.length; i++)  
        System.out.print(eex[i] + "_");  
}
```

3. Die innere **for**-Schleife prüft weiterhin auf die Länge eines anderen Arrays als das es zugreift.

Die Ausgabe lautet: "1_2_3_". (Anführungszeichen nur zur Übersicht)

ÜBUNGSaufGABE

Aufgabe 19: Begriffserklärungen

(4 Minuten)

Erklären Sie bitte jeweils in ein bis zwei Sätzen:

Aufgabe 19: Begriffserklärungen

(4 Minuten)

Erklären Sie bitte jeweils in ein bis zwei Sätzen:

- i) Was ist ein Deadlock (und wann tritt er auf)?

Aufgabe 19: Begriffserklärungen

(4 Minuten)

Erklären Sie bitte jeweils in ein bis zwei Sätzen:

- i) Was ist ein Deadlock (und wann tritt er auf)?
- ii) Was veranschaulicht ein Histogramm?

Aufgabe 19: Begriffserklärungen

(4 Minuten)

Erklären Sie bitte jeweils in ein bis zwei Sätzen:

- i) Was ist ein Deadlock (und wann tritt er auf)?
- ii) Was veranschaulicht ein Histogramm?
- iii) Was versteht man unter einem deterministischen Algorithmus?

Lösung 19: Begriffserklärungen

- i) Ein Deadlock (Verklemmung) ist ein gemeinsamer Zustand parallel laufender Programme,

Lösung 19: Begriffserklärungen

- i) Ein Deadlock (Verklemmung) ist ein gemeinsamer Zustand parallel laufender Programme, bei der diese sich gegenseitig blockieren.

Lösung 19: Begriffserklärungen

- i) Ein Deadlock (Verklemmung) ist ein gemeinsamer Zustand parallel laufender Programme, bei der diese sich gegenseitig blockieren.
Beispiel:

Lösung 19: Begriffserklärungen

- i) Ein Deadlock (Verklemmung) ist ein gemeinsamer Zustand parallel laufender Programme, bei der diese sich gegenseitig blockieren.
Beispiel: Prozess A benötigt die Ressource R_1 um R_2 fertigzustellen,

Lösung 19: Begriffserklärungen

- i) Ein Deadlock (Verklemmung) ist ein gemeinsamer Zustand parallel laufender Programme, bei der diese sich gegenseitig blockieren.
Beispiel: Prozess A benötigt die Ressource R_1 um R_2 fertigzustellen, Prozess B hingegen hält R_1 und benötigt R_2 um sie fertigzustellen.

Lösung 19: Begriffserklärungen

- i) Ein Deadlock (Verklemmung) ist ein gemeinsamer Zustand parallel laufender Programme, bei der diese sich gegenseitig blockieren.
Beispiel: Prozess A benötigt die Ressource R_1 um R_2 fertigzustellen, Prozess B hingegen hält R_1 und benötigt R_2 um sie fertigzustellen.
- ii) Ein Histogramm bezeichnet die grafische Veranschaulichung von Häufigkeiten.

Lösung 19: Begriffserklärungen

- i) Ein Deadlock (Verklemmung) ist ein gemeinsamer Zustand parallel laufender Programme, bei der diese sich gegenseitig blockieren.
Beispiel: Prozess A benötigt die Ressource R_1 um R_2 fertigzustellen, Prozess B hingegen hält R_1 und benötigt R_2 um sie fertigzustellen.
- ii) Ein Histogramm bezeichnet die grafische Veranschaulichung von Häufigkeiten.
- iii) Ein Algorithmus ist deterministisch, sofern sein nächster Schritt zu jeder Zeit eindeutig ist.

ÜBUNGSAUFGABE

Aufgabe 20: Programmieren eines Häufigkeitszähler (5 Minuten)

Schreiben Sie eine Java-Routine `countChars(String)`, welche ein Array `int[26]` zurückliefert,

Aufgabe 20: Programmieren eines Häufigkeitszähler (5 Minuten)

Schreiben Sie eine Java-Routine `countChars(String)`, welche ein Array `int[26]` zurückliefert, das von $0 \hat{=} a$

Aufgabe 20: Programmieren eines Häufigkeitszähler (5 Minuten)

Schreiben Sie eine Java-Routine `countChars(String)`, welche ein Array `int[26]` zurückliefert, das von $0 \hat{=} a$ bis $25 \hat{=} z$ die Häufigkeit der Buchstaben angibt.

Aufgabe 20: Programmieren eines Häufigkeitszähler (5 Minuten)

Schreiben Sie eine Java-Routine `countChars(String)`, welche ein Array `int[26]` zurückliefert, das von $0 \hat{=} a$ bis $25 \hat{=} z$ die Häufigkeit der Buchstaben angibt. Groß- und Kleinschreibung, sowie andere Zeichen sollen dabei ignoriert werden.

Lösung 20: Programmieren eines Häufigkeitszähler – Lösung

Lösung 20: Programmieren eines Häufigkeitszähler – Lösung

```
public static int[] countChars(String text){  
    int[] counts = new int[26];  
  
}
```

Lösung 20: Programmieren eines Häufigkeitszähler – Lösung

```
public static int[] countChars(String text){  
    int[] counts = new int[26];  
    for(char c : text.toLowerCase().toCharArray()){  
  
    }  
  
}
```

Lösung 20: Programmieren eines Häufigkeitszähler – Lösung

```
public static int[] countChars(String text){
    int[] counts = new int[26];
    for(char c : text.toLowerCase().toCharArray()){
        if('a' <= c && c <= 'z')
            counts[c - 'a'] += 1;
    }
}
```

Lösung 20: Programmieren eines Häufigkeitszähler – Lösung

```
public static int[] countChars(String text){
    int[] counts = new int[26];
    for(char c : text.toLowerCase().toCharArray()){
        if('a' <= c && c <= 'z')
            counts[c - 'a'] += 1;
    }
    return counts;
}
```

ÜBUNGSAUFGABE

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst).

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst). Für jedes Feld `field[i][j]` gibt der Wert an, ob es passierbar (`true`) oder blockiert (`false`) ist.

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst). Für jedes Feld `field[i][j]` gibt der Wert an, ob es passierbar (`true`) oder blockiert (`false`) ist. Die Variablen `static int turtleX, turtleY;`

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst). Für jedes Feld `field[i][j]` gibt der Wert an, ob es passierbar (`true`) oder blockiert (`false`) ist. Die Variablen `static int turtleX, turtleY`; repräsentieren eine Schildkröte an Position `field[turtleY][turtleX]`.

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst). Für jedes Feld `field[i][j]` gibt der Wert an, ob es passierbar (`true`) oder blockiert (`false`) ist. Die Variablen `static int turtleX, turtleY`; repräsentieren eine Schildkröte an Position `field[turtleY][turtleX]`.

Schreiben Sie eine Funktion `moveTurtle(int x, int y)`, die versucht die Schildkröte relativ um `x` und `y` zu bewegen.

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst). Für jedes Feld `field[i][j]` gibt der Wert an, ob es passierbar (`true`) oder blockiert (`false`) ist. Die Variablen `static int turtleX, turtleY`; repräsentieren eine Schildkröte an Position `field[turtleY][turtleX]`.

Schreiben Sie eine Funktion `moveTurtle(int x, int y)`, die versucht die Schildkröte relativ um `x` und `y` zu bewegen.

Eine Bewegung gelingt nur, wenn das Zielfeld frei ist.

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst). Für jedes Feld `field[i][j]` gibt der Wert an, ob es passierbar (`true`) oder blockiert (`false`) ist. Die Variablen `static int turtleX, turtleY`; repräsentieren eine Schildkröte an Position `field[turtleY][turtleX]`.

Schreiben Sie eine Funktion `moveTurtle(int x, int y)`, die versucht die Schildkröte relativ um `x` und `y` zu bewegen.

Eine Bewegung gelingt nur, wenn das Zielfeld frei ist. In diesem Fall gilt es, `turtleX` und `turtleY` zu modifizieren und `true` zurück zu liefern.

Aufgabe 21: Programmieren: Turtlewalk

(6 Minuten)

Das Array `static boolean[][] field` repräsentiere ein rechteckiges Gitter (welches mindestens 1 Feld umfasst). Für jedes Feld `field[i][j]` gibt der Wert an, ob es passierbar (`true`) oder blockiert (`false`) ist. Die Variablen `static int turtleX, turtleY`; repräsentieren eine Schildkröte an Position `field[turtleY][turtleX]`.

Schreiben Sie eine Funktion `moveTurtle(int x, int y)`, die versucht die Schildkröte relativ um `x` und `y` zu bewegen.

Eine Bewegung gelingt nur, wenn das Zielfeld frei ist. In diesem Fall gilt es, `turtleX` und `turtleY` zu modifizieren und `true` zurück zu liefern.

Andernfalls sollen die Koordinaten unverändert bleiben, sowie ein `false` zurückgegeben werden.

Lösung 21: Programmieren: Turtlewalk

Lösung 21: Programmieren: Turtlewalk

```
public static boolean moveTurtle(int tx, int ty){  
    int nx = turtleX + tx, ny = turtleY + ty;  
  
}
```


Lösung 21: Programmieren: Turtlewalk

```
public static boolean moveTurtle(int tx, int ty){  
    int nx = turtleX + tx, ny = turtleY + ty;  
    if(nx >= field[0].length || nx < 0 // x-Richtung  
        || ny >= field.length || ny < 0 // y-Richtung  
        || !field[ny][nx]) // blockiert  
  
}
```

Lösung 21: Programmieren: Turtlewalk

```
public static boolean moveTurtle(int tx, int ty){
    int nx = turtleX + tx, ny = turtleY + ty;
    if(nx >= field[0].length || nx < 0 // x-Richtung
        || ny >= field.length || ny < 0 // y-Richtung
        || !field[ny][nx]) // blockiert
        return false;
}
```

Lösung 21: Programmieren: Turtlewalk

```
public static boolean moveTurtle(int tx, int ty){
    int nx = turtleX + tx, ny = turtleY + ty;
    if(nx >= field[0].length || nx < 0 // x-Richtung
        || ny >= field.length || ny < 0 // y-Richtung
        || !field[ny][nx]) // blockiert
        return false;
    turtleX = nx;
    turtleY = ny;
}
```


Lösung 21: Programmieren: Turtlewalk

```
public static boolean moveTurtle(int tx, int ty){
    int nx = turtleX + tx, ny = turtleY + ty;
    if(nx >= field[0].length || nx < 0 // x-Richtung
        || ny >= field.length || ny < 0 // y-Richtung
        || !field[ny][nx]) // blockiert
        return false;
    turtleX = nx;
    turtleY = ny;
    return true;
}
```

ÜBUNGSaufGABE

Aufgabe 22: Programmieren: Turtlewalk, II

(6 Minuten)

Im Kontext der vorherigen Aufgabe existiere die Methode `moveTurtle(int tx, int ty)`.

Aufgabe 22: Programmieren: Turtlewalk, II

(6 Minuten)

Im Kontext der vorherigen Aufgabe existiere die Methode `moveTurtle(int tx, int ty)`. Weiter seien nun diese Konstanten gegeben:

Aufgabe 22: Programmieren: Turtlewalk, II

(6 Minuten)

Im Kontext der vorherigen Aufgabe existiere die Methode `moveTurtle(int tx, int ty)`. Weiter seien nun diese Konstanten gegeben:

```
static final int UP = 0, LEFT = 1, DOWN = 2, RIGHT = 3;
```

Aufgabe 22: Programmieren: Turtlewalk, II

(6 Minuten)

Im Kontext der vorherigen Aufgabe existiere die Methode `moveTurtle(int tx, int ty)`. Weiter seien nun diese Konstanten gegeben:

```
static final int UP = 0, LEFT = 1, DOWN = 2, RIGHT = 3;
```

Schreiben Sie eine Methode `moveTurtle(int)`, welche die relative Bewegung um *so viele Felder wie möglich* vornimmt.

Aufgabe 22: Programmieren: Turtlewalk, II

(6 Minuten)

Im Kontext der vorherigen Aufgabe existiere die Methode `moveTurtle(int tx, int ty)`. Weiter seien nun diese Konstanten gegeben:

```
static final int UP = 0, LEFT = 1, DOWN = 2, RIGHT = 3;
```

Schreiben Sie eine Methode `moveTurtle(int)`, welche die relative Bewegung um *so viele Felder wie möglich* vornimmt. Die Methode soll die Anzahl der passierten Felder zurückgeben.

Aufgabe 22: Programmieren: Turtlewalk, II

(6 Minuten)

Im Kontext der vorherigen Aufgabe existiere die Methode `moveTurtle(int tx, int ty)`. Weiter seien nun diese Konstanten gegeben:

```
static final int UP = 0, LEFT = 1, DOWN = 2, RIGHT = 3;
```

Schreiben Sie eine Methode `moveTurtle(int)`, welche die relative Bewegung um *so viele Felder wie möglich* vornimmt. Die Methode soll die Anzahl der passierten Felder zurückgeben.

Dabei sei „Oben“ als eine Erhöhung der y-Koordinate definiert.

Lösung 22: Programmieren: Turtlewalk, II

Lösung 22: Programmieren: Turtlewalk, II

```
public static int moveTurtle(int dir){  
    int tx = 0, ty = 0;  
    switch (dir) {  
  
    }  
  
}
```

Lösung 22: Programmieren: Turtlewalk, II

```
public static int moveTurtle(int dir){  
    int tx = 0, ty = 0;  
    switch (dir) {  
        case UP: ty = 1; break; case DOWN: ty = -1; break;  
        case RIGHT: tx = 1; break; case LEFT: tx = -1; break;  
        default: return -1; // Unterscheiden von kein Feld  
    }  
  
}
```

Lösung 22: Programmieren: Turtlewalk, II

```
public static int moveTurtle(int dir){
    int tx = 0, ty = 0;
    switch (dir) {
        case UP: ty = 1; break; case DOWN: ty = -1; break;
        case RIGHT: tx = 1; break; case LEFT: tx = -1; break;
        default: return -1; // Unterscheiden von kein Feld
    }
    int total = 0;

}
```

Lösung 22: Programmieren: Turtlewalk, II

```
public static int moveTurtle(int dir){
    int tx = 0, ty = 0;
    switch (dir) {
        case UP: ty = 1; break; case DOWN: ty = -1; break;
        case RIGHT: tx = 1; break; case LEFT: tx = -1; break;
        default: return -1; // Unterscheiden von kein Feld
    }
    int total = 0;
    while(moveTurtle(tx, ty)) { total++; }
}
```


Lösung 22: Programmieren: Turtlewalk, II

```
public static int moveTurtle(int dir){
    int tx = 0, ty = 0;
    switch (dir) {
        case UP: ty = 1; break; case DOWN: ty = -1; break;
        case RIGHT: tx = 1; break; case LEFT: tx = -1; break;
        default: return -1; // Unterscheiden von kein Feld
    }
    int total = 0;
    while(moveTurtle(tx, ty)) { total++; }
    return total;
}
```

OO-Konzepte



PARADIGMA: OBJEKTORIENTIERUNG

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „Person“ besteht aus zwei Komponenten:

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „Person“ besteht aus zwei Komponenten:
DATEN: diese, wie der Name, das Alter oder die Größe der Person,

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „**Person**“ besteht aus zwei Komponenten:
DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „**Person**“ besteht aus zwei Komponenten:
 - DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.
 - FUNKTIONEN: diese definieren, was eine **Person** kann.

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „**Person**“ besteht aus zwei Komponenten:
 - DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.
 - FUNKTIONEN: diese definieren, was eine **Person** kann. Also `gehen()`, `tanzen()`, `reden()`,

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „**Person**“ besteht aus zwei Komponenten:
 - DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.
 - FUNKTIONEN: diese definieren, was eine **Person** kann. Also `gehen()`, `tanzen()`, `reden()`, Dies sind die **Methoden** der Klasse, sie definieren das *Verhalten*.

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „**Person**“ besteht aus zwei Komponenten:
 - DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.
 - FUNKTIONEN: diese definieren, was eine **Person** kann. Also `gehen()`, `tanzen()`, `reden()`, Dies sind die **Methoden** der Klasse, sie definieren das *Verhalten*.
- Eine Funktion nennen wir *Methode*, wenn sie an eine Objekt gebunden ist. (Zumindest in Java)

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.
- Eine Klasse wie „**Person**“ besteht aus zwei Komponenten:
 - DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.
 - FUNKTIONEN: diese definieren, was eine **Person** kann. Also `gehen()`, `tanzen()`, `reden()`, Dies sind die **Methoden** der Klasse, sie definieren das *Verhalten*.
- Eine Funktion nennen wir *Methode*, wenn sie an eine Objekt gebunden ist. (Zumindest in Java)
- Klassen erlauben es, komplexe Probleme zu abstrahieren und Funktionalität zu kapseln.

PARADIGMA: OBJEKTORIENTIERUNG

Definition 3: Objekt

Definition 3: Objekt

Ein Objekt bezeichnet eine spezifische Ausprägung (eine sogenannte *Instanz*) einer Klasse.

Definition 3: Objekt

Ein Objekt bezeichnet eine spezifische Ausprägung (eine sogenannte *Instanz*) einer Klasse.

OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.

Definition 3: Objekt

Ein Objekt bezeichnet eine spezifische Ausprägung (eine sogenannte *Instanz*) einer Klasse.

OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.

OBJEKTVERHALTEN: bezeichnet die Reaktion des Objekts auf das Aufrufen von Methoden.

Definition 3: Objekt

Ein Objekt bezeichnet eine spezifische Ausprägung (eine sogenannte *Instanz*) einer Klasse.

OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.

OBJEKTVERHALTEN: bezeichnet die Reaktion des Objekts auf das Aufrufen von Methoden.

OBJEKTIDENTITÄT: ermöglicht die eindeutige Identifikation.
In Java: die Speicheradresse.

Definition 3: Objekt

Ein Objekt bezeichnet eine spezifische Ausprägung (eine sogenannte *Instanz*) einer Klasse.

OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.

OBJEKTVERHALTEN: bezeichnet die Reaktion des Objekts auf das Aufrufen von Methoden.

OBJEKTIDENTITÄT: ermöglicht die eindeutige Identifikation.
In Java: die Speicheradresse.

- Legt man die Objektorientierung streng aus,

Definition 3: Objekt

Ein Objekt bezeichnet eine spezifische Ausprägung (eine sogenannte *Instanz*) einer Klasse.

OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.

OBJEKTVERHALTEN: bezeichnet die Reaktion des Objekts auf das Aufrufen von Methoden.

OBJEKTIDENTITÄT: ermöglicht die eindeutige Identifikation.
In Java: die Speicheradresse.

- Legt man die Objektorientierung streng aus, so darf ein Objekt keinen direkten Zugriff auf seinen Zustand gestatten. (\Rightarrow Getter & Setter).

- Das Verhalten von Klassen wird oft auch als ein Botschaft-beziehungsweise Nachrichtensystem betrachtet.

- Das Verhalten von Klassen wird oft auch als ein Botschaft-beziehungsweise Nachrichtensystem betrachtet.
- Hier bezeichnet das Senden einer Botschaft mit Inhalt das Aufrufen der Methode mit entsprechendem Inhalt.

PROGRAMMKONTEXT

- Das Verhalten von Klassen wird oft auch als ein Botschaft-beziehungsweise Nachrichtensystem betrachtet.
- Hier bezeichnet das Senden einer Botschaft mit Inhalt das Aufrufen der Methode mit entsprechendem Inhalt.
- Aus Sicht der Objektorientierung ist ein *Programm*

- Das Verhalten von Klassen wird oft auch als ein Botschaft-beziehungsweise Nachrichtensystem betrachtet.
- Hier bezeichnet das Senden einer Botschaft mit Inhalt das Aufrufen der Methode mit entsprechendem Inhalt.
- Aus Sicht der Objektorientierung ist ein *Programm* nicht mehr als das (wechselseitige) Aufrufen eben dieser Methoden.

EINE AUFGABE ZWISCHENDURCH

Aufgabe 23: Vorteile der OOP

(2 Minuten)

Nennen Sie drei Vorteile der objektorientierten Programmierung, neben der Möglichkeit die Realität zu abstrahieren.

Lösung 23: Vorteile der OOP

Lösung 23: Vorteile der OOP

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

Lösung 23: Vorteile der OOP

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

MODULARITÄT: Eine Klasse kann unabhängig von anderen geschrieben und gehalten werden.

Lösung 23: Vorteile der OOP

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

MODULARITÄT: Eine Klasse kann unabhängig von anderen geschrieben und gehalten werden.

INFORMATIONSVERRUCKUNG: Die Implementierungsdetails des Objekts werden verborgen.

Lösung 23: Vorteile der OOP

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

MODULARITÄT: Eine Klasse kann unabhängig von anderen geschrieben und gehalten werden.

INFORMATIONSVERRUCKUNG: Die Implementierungsdetails des Objekts werden verborgen.

WIEDERVERWENDBARKEIT: Eine Klasse kann so geschrieben werden, dass sie auch in anderen Projekten wiederverwendet werden kann

Lösung 23: Vorteile der OOP

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

MODULARITÄT: Eine Klasse kann unabhängig von anderen geschrieben und gehalten werden.

INFORMATIONSVERRUCKUNG: Die Implementierungsdetails des Objekts werden verborgen.

WIEDERVERWENDBARKEIT: Eine Klasse kann so geschrieben werden, dass sie auch in anderen Projekten wiederverwendet werden kann (ein Datentyp, wie ein Ringbuffer zum Beispiel).

Lösung 23: Vorteile der OOP

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

MODULARITÄT: Eine Klasse kann unabhängig von anderen geschrieben und gehalten werden.

INFORMATIONSVERRUCKUNG: Die Implementierungsdetails des Objekts werden verborgen.

WIEDERVERWENDBARKEIT: Eine Klasse kann so geschrieben werden, dass sie auch in anderen Projekten wiederverwendet werden kann (ein Datentyp, wie ein Ringbuffer zum Beispiel).

KOMPOSITION/POLYMORPHIE: Klassen implementieren Schnittstellen, die sie von der Implementation abstrahieren,

Lösung 23: Vorteile der OOP

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

MODULARITÄT: Eine Klasse kann unabhängig von anderen geschrieben und gehalten werden.

INFORMATIONSVERRUCKUNG: Die Implementierungsdetails des Objekts werden verborgen.

WIEDERVERWENDBARKEIT: Eine Klasse kann so geschrieben werden, dass sie auch in anderen Projekten wiederverwendet werden kann (ein Datentyp, wie ein Ringbuffer zum Beispiel).

KOMPOSITION/POLYMORPHIE: Klassen implementieren Schnittstellen, die sie von der Implementation abstrahieren, so dass diese sich problemlos austauschen lassen.

VOM KLASSENKONZEPT ZUM JAVACODE

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.
- Ein solcher Punkt (x,y) benötigt zwei Attribute:

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.
- Ein solcher Punkt (x,y) benötigt zwei Attribute:
x-KOORDINATE: die x-Komponente (Fließkommazahl).

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse **Punkt2D** zu kreieren.
- Ein solcher Punkt (x,y) benötigt zwei Attribute:
X-KOORDINATE: die x-Komponente (Fließkommazahl).
Y-KOORDINATE: die y-Komponente (Fließkommazahl).

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.
- Ein solcher Punkt (x,y) benötigt zwei Attribute:
X-KOORDINATE: die x-Komponente (Fließkommazahl).
Y-KOORDINATE: die y-Komponente (Fließkommazahl).
- Wir möchten ein paar Dinge mit dem Punkt anstellen können:

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.
- Ein solcher Punkt (x,y) benötigt zwei Attribute:
X-KOORDINATE: die x-Komponente (Fließkommazahl).
Y-KOORDINATE: die y-Komponente (Fließkommazahl).
- Wir möchten ein paar Dinge mit dem Punkt anstellen können:
 - Den Punkt (relativ) verschieben.

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.
- Ein solcher Punkt (x,y) benötigt zwei Attribute:
X-KOORDINATE: die x-Komponente (Fließkommazahl).
Y-KOORDINATE: die y-Komponente (Fließkommazahl).
- Wir möchten ein paar Dinge mit dem Punkt anstellen können:
 - Den Punkt (relativ) verschieben.
 - Den Abstand zu einem anderen Punkt berechnen.

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.
- Ein solcher Punkt (x,y) benötigt zwei Attribute:
X-KOORDINATE: die x-Komponente (Fließkommazahl).
Y-KOORDINATE: die y-Komponente (Fließkommazahl).
- Wir möchten ein paar Dinge mit dem Punkt anstellen können:
 - Den Punkt (relativ) verschieben.
 - Den Abstand zu einem anderen Punkt berechnen.
 - Den Punkt auf einen anderen Punkt setzen.

Point2D
- x : double - y : double
+ distance(Point2D): double + shift(double, double): void + copy(Point2D): void

IMPLEMENTATION IN JAVA

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.
- Diesem folgt der Name der Klasse,

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.
- Diesem folgt der Name der Klasse, der dem Dateinamen entsprechen *muss*.

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.
- Diesem folgt der Name der Klasse, der dem Dateinamen entsprechen *muss*. Es wird auf groß-Kleinschreibung geachtet.

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.
- Diesem folgt der Name der Klasse, der dem Dateinamen entsprechen *muss*. Es wird auf groß-Kleinschreibung geachtet.
- Innerhalb einer Klasse referenziert **this** auf das jeweilige Objekt.

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.
- Diesem folgt der Name der Klasse, der dem Dateinamen entsprechen *muss*. Es wird auf groß-Kleinschreibung geachtet.
- Innerhalb einer Klasse referenziert **this** auf das jeweilige Objekt.
- Bei der Implementation müssen wir den Konstruktor beachten:

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.
- Diesem folgt der Name der Klasse, der dem Dateinamen entsprechen *muss*. Es wird auf groß-Kleinschreibung geachtet.
- Innerhalb einer Klasse referenziert **this** auf das jeweilige Objekt.
- Bei der Implementation müssen wir den Konstruktor beachten:

Definition 4: Konstruktor

Ein Konstruktor verhält sich nur bedingt wie eine Methode ohne Rückgabewert. Er ist *keine Methode*,

IMPLEMENTATION IN JAVA

- Die Implementation einer Klasse erfolgt mit der Schlüsselwort **class**.
- Diesem folgt der Name der Klasse, der dem Dateinamen entsprechen *muss*. Es wird auf groß-Kleinschreibung geachtet.
- Innerhalb einer Klasse referenziert **this** auf das jeweilige Objekt.
- Bei der Implementation müssen wir den Konstruktor beachten:

Definition 4: Konstruktor

Ein Konstruktor verhält sich nur bedingt wie eine Methode ohne Rückgabewert. Er ist *keine Methode*, er gehört zum Klassenkonstrukt und kann nur mit **new** aufgerufen werden. Allerdings kann man Konstruktoren überladen.

Ein Konstruktor trägt stets denselben Namen wie die Klasse selbst und kann den Aufruf an andere Überladungen des Konstruktors mittels **this** durchreichen.

IMPLEMENTATION: KONSTRUKTOR

IMPLEMENTATION: KONSTRUKTOR

```
public class Point2D {
```

```
}
```


IMPLEMENTATION: KONSTRUKTOR

```
public class Point2D {  
  
    private double x, y;  
  
    // Leerer Konstruktor  
    public Point2D() { this(0.0, 0.0); }  
  
}
```

IMPLEMENTATION: KONSTRUKTOR

```
public class Point2D {  
  
    private double x, y;  
  
    // Leerer Konstruktor  
    public Point2D() { this(0.0, 0.0); }  
  
    // Initialisiere mit Punkt  
    public Point2D(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

IMPLEMENTATION: METHODEN

IMPLEMENTATION: METHODEN

- Die Implementation der Methoden läuft wie bekannt!

IMPLEMENTATION: METHODEN

- Die Implementation der Methoden läuft wie bekannt!
- Exemplarisch das relative Verschieben:

IMPLEMENTATION: METHODEN

- Die Implementation der Methoden läuft wie bekannt!
- Exemplarisch das relative Verschieben:

```
public class Point2D {  
    //...  
  
}
```

IMPLEMENTATION: METHODEN

- Die Implementation der Methoden läuft wie bekannt!
- Exemplarisch das relative Verschieben:

```
public class Point2D {  
    //...  
  
    public void shift(double sx, double sy) {  
  
    }  
}
```


IMPLEMENTATION: METHODEN

- Die Implementation der Methoden läuft wie bekannt!
- Exemplarisch das relative Verschieben:

```
public class Point2D {  
    //...  
  
    public void shift(double sx, double sy) {  
        this.x += sx;  
        this.y += sy;  
    }  
}
```

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden,

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:
 - equals: die Methode `equals(Object)` prüft das Objekt mit dem Übergebenen auf „Gleichheit“.

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:
 - `equals`: die Methode `equals(Object)` prüft das Objekt mit dem Übergebenen auf „Gleichheit“. So können die hierfür relevanten Attribute genau festgelegt werden.

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:
 - `equals`: die Methode `equals(Object)` prüft das Objekt mit dem Übergebenen auf „Gleichheit“. So können die hierfür relevanten Attribute genau festgelegt werden.
 - `toString`: wird aufgerufen, um eine Repräsentation als Zeichenkette zu erhalten.

EINE AUFGABE ZWISCHENDURCH

Aufgabe 24: Eine .equals()-Methode

(6 Minuten)

Schreiben Sie eine `equals(Object)`-Methode für `Point2D`. Hinweis: Denken Sie an **instanceof** beziehungsweise `getClass()`.

```
public class Point2D {  
    private double x, y;  
    // ...  
}
```


Lösung 24: Eine .equals()-Methode

Lösung 24: Eine .equals()-Methode

```
public boolean equals(Object obj) {  
    if (this == obj) return true; // Sind identisch  
  
}
```


Lösung 24: Eine .equals()-Methode

```
public boolean equals(Object obj) {  
    if (this == obj) return true; // Sind identisch  
    if (obj == null) return false; // Wir sind nicht null  
  
}
```

Lösung 24: Eine .equals()-Methode

```
public boolean equals(Object obj) {  
    if (this == obj) return true; // Sind identisch  
    if (obj == null) return false; // Wir sind nicht null  
    // Ist es von der selben Klasse?  
  
}
```

Lösung 24: Eine .equals()-Methode

```
public boolean equals(Object obj) {  
    if (this == obj) return true; // Sind identisch  
    if (obj == null) return false; // Wir sind nicht null  
    // Ist es von der selben Klasse?  
    // Oder: 'if (obj instanceof Point2D)' (erspart null-check)  
    if (this.getClass() != obj.getClass())  
        return false;  
  
}
```

Lösung 24: Eine .equals()-Methode

```
public boolean equals(Object obj) {  
    if (this == obj) return true; // Sind identisch  
    if (obj == null) return false; // Wir sind nicht null  
    // Ist es von der selben Klasse?  
    // Oder: 'if (obj instanceof Point2D)' (erspart null-check)  
    if (this.getClass() != obj.getClass())  
        return false;  
    Point2D p1 = (Point2D) obj; // Es ist ein Point2D  
}
```

Lösung 24: Eine .equals()-Methode

```
public boolean equals(Object obj) {  
    if (this == obj) return true; // Sind identisch  
    if (obj == null) return false; // Wir sind nicht null  
    // Ist es von der selben Klasse?  
    // Oder: 'if (obj instanceof Point2D)' (erspart null-check)  
    if (this.getClass() != obj.getClass())  
        return false;  
    Point2D p1 = (Point2D) obj; // Es ist ein Point2D  
    return this.x == p1.x && this.y == p1.y;  
}
```


STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier:  static.pdf.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: [📎 static.pdf](#).
- Nichtstatische Methoden, sind an ein Objekt gebunden.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier:  static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: [📎 static.pdf](#).
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double), ...`).

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: [📎 static.pdf](#).
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double), ...`).
- Diese Methoden deklarieren wir mit **static**,

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: [📎 static.pdf](#).
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double), ...`).
- Diese Methoden deklarieren wir mit **static**, sie sind nun auch ohne ein Objekt aufrufbar.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: [📎 static.pdf](#).
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double), ...`).
- Diese Methoden deklarieren wir mit **static**, sie sind nun auch ohne ein Objekt aufrufbar. So kann man zum Beispiel auch ein statisches `Point2D::distance(Point2D, Point2D)` für zwei Punkte bauen.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: [📎 static.pdf](#).
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double), ...`).
- Diese Methoden deklarieren wir mit **static**, sie sind nun auch ohne ein Objekt aufrufbar. So kann man zum Beispiel auch ein statisches `Point2D::distance(Point2D, Point2D)` für zwei Punkte bauen.
- Auch Variablen, die für alle Objekte einer Klasse identisch sind,

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: [📎 static.pdf](#).
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double), ...`).
- Diese Methoden deklarieren wir mit **static**, sie sind nun auch ohne ein Objekt aufrufbar. So kann man zum Beispiel auch ein statisches `Point2D::distance(Point2D, Point2D)` für zwei Punkte bauen.
- Auch Variablen, die für alle Objekte einer Klasse identisch sind, können wir mit **static** deklarieren.

DER LEBENSZYKLUS EINES OBJEKTS

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe...)

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe...) können wir weitere Referenzen darauf erstellen.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe...) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe...) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.
- Existiert für ein Objekt keine Referenz mehr,

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe...) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.
- Existiert für ein Objekt keine Referenz mehr, gibt es (für Java) keine Möglichkeit mehr darauf zuzugreifen.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe...) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.
- Existiert für ein Objekt keine Referenz mehr, gibt es (für Java) keine Möglichkeit mehr darauf zuzugreifen.
- In diesem Fall wird es (irgendwann) vom Garbage-Collector aufgeräumt.

Aufgabe 25: Garbage Collector

(2 Minuten)

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe...) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.
- Existiert für ein Objekt keine Referenz mehr, gibt es (für Java) keine Möglichkeit mehr darauf zuzugreifen.
- In diesem Fall wird es (irgendwann) vom Garbage-Collector aufgeräumt.

Aufgabe 25: Garbage Collector

(2 Minuten)

Kann der Java Garbage Collector direkt beeinflusst werden? Was sind die Möglichkeiten?

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen und mittels `System.gc()` den Prozess erbitten. Nach dem Abschluss dieser Methode hat Java „sein Bestes getan“.

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen und mittels `System.gc()` den Prozess erbitten. Nach dem Abschluss dieser Methode hat Java „sein Bestes getan“.

Information: Der GC räumt nicht nur Speicher auf, sondern räumt ihn auch neu an.

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen und mittels `System.gc()` den Prozess erbitten. Nach dem Abschluss dieser Methode hat Java „sein Bestes getan“.

Information: Der GC räumt nicht nur Speicher auf, sondern räumt ihn auch neu an. Es gibt verschiedene Varianten für einen GC (Referencecount,

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen und mittels `System.gc()` den Prozess erbitten. Nach dem Abschluss dieser Methode hat Java „sein Bestes getan“.

Information: Der GC räumt nicht nur Speicher auf, sondern räumt ihn auch neu an. Es gibt verschiedene Varianten für einen GC (Referencecount, Mark & Sweep,

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen und mittels `System.gc()` den Prozess erbitten. Nach dem Abschluss dieser Methode hat Java „sein Bestes getan“.

Information: Der GC räumt nicht nur Speicher auf, sondern räumt ihn auch neu an. Es gibt verschiedene Varianten für einen GC (Referencecount, Mark & Sweep, Stop & Copy).

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen und mittels `System.gc()` den Prozess erbitten. Nach dem Abschluss dieser Methode hat Java „sein Bestes getan“.

Information: Der GC räumt nicht nur Speicher auf, sondern räumt ihn auch neu an. Es gibt verschiedene Varianten für einen GC (Referencecount, Mark & Sweep, Stop & Copy). Wird ein Objekt aufgeräumt,

Lösung 25: Garbage Collector

Wir können den Collector nicht direkt beeinflussen! Allerdings können wir nicht mehr benötigte Objekte auf **null** setzen und mittels `System.gc()` den Prozess erbitten. Nach dem Abschluss dieser Methode hat Java „sein Bestes getan“.

Information: Der GC räumt nicht nur Speicher auf, sondern räumt ihn auch neu an. Es gibt verschiedene Varianten für einen GC (Referencecount, Mark & Sweep, Stop & Copy). Wird ein Objekt aufgeräumt, ruft die JVM direkt davor die Methode `finalize()` auf.

ENUMERATIONEN

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden,

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann.

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*
- Die einfachste Möglichkeit eine Enumeration zu definieren,

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*
- Die einfachste Möglichkeit eine Enumeration zu definieren, funktioniert über das **enum**-Schlüsselwort:

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*
- Die einfachste Möglichkeit eine Enumeration zu definieren, funktioniert über das **enum**-Schlüsselwort:

```
enum <Name der Enumeration> {  
    <Komma separierte Liste an Werten>  
}
```

ENUMERATIONEN DEFINIEREN

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {  
    HOCH, RUNTER, LINKS, RECHTS  
}
```

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {  
    HOCH, RUNTER, LINKS, RECHTS  
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {  
    HOCH, RUNTER, LINKS, RECHTS  
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.
- *Hinweis:* Anders als in Sprachen wie C++,

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {  
    HOCH, RUNTER, LINKS, RECHTS  
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.
- *Hinweis:* Anders als in Sprachen wie C++, wird den Konstanten kein (Integer)-Wert zugeordnet.

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {  
    HOCH, RUNTER, LINKS, RECHTS  
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.
- *Hinweis:* Anders als in Sprachen wie C++, wird den Konstanten kein (Integer)-Wert zugeordnet. (Dafür gibt es `ordinal()`.)

ENUMERATIONEN VERWENDEN

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){
```

```
}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){  
    switch(ziel) {  
  
  
  
  
  
  
    }  
  
}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){  
    switch(ziel) {  
        case HOCH: return "Es_geht_nach_oben!";  
  
    }  
  
}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){  
    switch(ziel) {  
        case HOCH: return "Es geht nach oben!";  
        case RUNTER: return "Es geht nach unten!";  
  
    }  
  
}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){  
    switch(ziel) {  
        case HOCH: return "Es geht nach oben!";  
        case RUNTER: return "Es geht nach unten!";  
        case LINKS: return "Es geht nach links!";  
  
    }  
  
}
```


ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es geht nach oben!";
        case RUNTER: return "Es geht nach unten!";
        case LINKS: return "Es geht nach links!";
        case RECHTS: return "Es geht nach rechts!";
    }
}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es_geht_nach_oben!";
        case RUNTER: return "Es_geht_nach_unten!";
        case LINKS: return "Es_geht_nach_links!";
        case RECHTS: return "Es_geht_nach_rechts!";
    }
    return "Fehler!_Richtung_unbekannt";
}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es_geht_nach_oben!";
        case RUNTER: return "Es_geht_nach_unten!";
        case LINKS: return "Es_geht_nach_links!";
        case RECHTS: return "Es_geht_nach_rechts!";
    }
    return "Fehler!_Richtung_unbekannt";
}
```

- Beispielhafte Verwendung:

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und `switch-case`:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es geht nach oben!";
        case RUNTER: return "Es geht nach unten!";
        case LINKS: return "Es geht nach links!";
        case RECHTS: return "Es geht nach rechts!";
    }
    return "Fehler! Richtung unbekannt";
}
```

- Beispielhafte Verwendung:

```
System.out.println(wohinGehtEs(Richtung.RECHTS));
```

WAS ENUMERATIONEN LIEFERN

WAS ENUMERATIONEN LIEFERN

- Mittels `<Enum Name>.values()` erhalten wir ein Array aller Werte.

WAS ENUMERATIONEN LIEFERN

- Mittels `<Enum Name>.values()` erhalten wir ein Array aller Werte.
- `<Enum Name>.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen,

WAS ENUMERATIONEN LIEFERN

- Mittels `<Enum Name>.values()` erhalten wir ein Array aller Werte.
- `<Enum Name>.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.

WAS ENUMERATIONEN LIEFERN

- Mittels `<Enum Name>.values()` erhalten wir ein Array aller Werte.
- `<Enum Name>.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.
- Enumerationen besitzen ein wie zu erwarten funktionierendes `equals(Object)`.

WAS ENUMERATIONEN LIEFERN

- Mittels `<Enum Name>.values()` erhalten wir ein Array aller Werte.
- `<Enum Name>.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.
- Enumerationen besitzen ein wie zu erwarten funktionierendes `equals(Object)`.
- Analog funktioniert `toString()` wie zu erwarten.

WAS ENUMERATIONEN LIEFERN

- Mittels `<Enum Name>.values()` erhalten wir ein Array aller Werte.
- `<Enum Name>.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.
- Enumerationen besitzen ein wie zu erwarten funktionierendes `equals(Object)`.
- Analog funktioniert `toString()` wie zu erwarten.
- Da es sich auch um Klassen handelt, können wir den Konstanten auch Datentypen zuordnen!

ALTERNATIVE RICHTUNGS-ENUM

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {
```

```
}
```

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {
```

```
    private String text;
```

```
}
```

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {  
  
    private String text;  
    public String getText() { return this.text; }  
  
}
```

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {  
  
    private String text;  
    public String getText() { return this.text; }  
  
    Richtung(String text) {  
        this.text = text;  
    }  
}
```


ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {  
    HOCH("Es geht nach oben!"),  
    RUNTER("Es geht nach unten!"),  
    LINKS("Es geht nach links!"),  
    RECHTS("Es geht nach rechts!");  
  
    private String text;  
    public String getText() { return this.text; }  
  
    Richtung(String text) {  
        this.text = text;  
    }  
}
```

ABSCHLUSS ZU ENUMERATIONEN

ABSCHLUSS ZU ENUMERATIONEN

- Die Funktion `wohinGehtEs` lässt sich nun kompakter fassen:

ABSCHLUSS ZU ENUMERATIONEN

- Die Funktion `wohinGehtEs` lässt sich nun kompakter fassen:

```
public static String wohinGehtEs(Richtung ziel){  
    return ziel.getText();  
}
```

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen),

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen), wird *keine* Kopie des Objekts erstellt,

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen), wird *keine* Kopie des Objekts erstellt, sondern eine Kopie der Referenz auf das Objekt übergeben.

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen), wird *keine* Kopie des Objekts erstellt, sondern eine Kopie der Referenz auf das Objekt übergeben.
- Dieser (Parameter-)Übergabemechanismus ähnelt *call-by-reference* aus anderen Programmiersprachen

EINE LEICHTE AUFGABE ALS EINSTIEG

Aufgabe 26: Was liefert dieser Code?

(1 Minute)

Aufgabe 26: Was liefert dieser Code?

(1 Minute)

```
public static void multBy2(int[] x){  
    x[0] += x[1] * 2;  
}
```

Aufgabe 26: Was liefert dieser Code?

(1 Minute)

```
public static void multBy2(int[] x){
    x[0] += x[1] * 2;
}

public static void main(String[] args) {

}
```

Aufgabe 26: Was liefert dieser Code?

(1 Minute)

```
public static void multBy2(int[] x){
    x[0] += x[1] * 2;
}

public static void main(String[] args) {
    int[] arr = { 22, 10 };
    System.out.println(arr[0]);
    multBy2(arr);
    System.out.println(arr[0]);
}
```

EINE LEICHTE AUFGABE ALS EINSTIEG – LÖSUNG

Lösung 26: Was liefert dieser Code?

Der Code liefert 22 und 42. Ersterer Wert, da wir das Array ja so zuweisen,

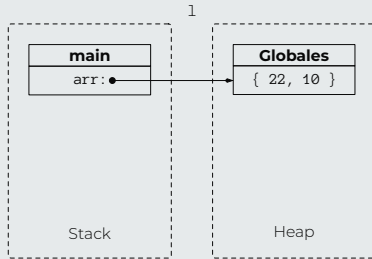
Lösung 26: Was liefert dieser Code?

Der Code liefert 22 und 42. Ersterer Wert, da wir das Array ja so zuweisen, den zweiten, weil durch die Übergabe des Arrays mittels *call-by-reference* auch das Ursprungsarray verändert wird.

EINE LEICHTE AUFGABE ALS EINSTIEG – LÖSUNG

Lösung 26: Was liefert dieser Code?

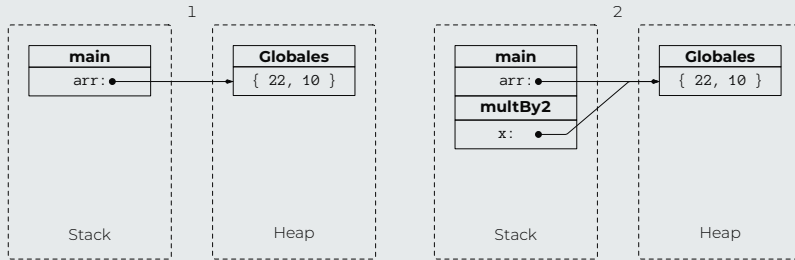
Der Code liefert 22 und 42. Ersterer Wert, da wir das Array ja so zuweisen, den zweiten, weil durch die Übergabe des Arrays mittels *call-by-reference* auch das Ursprungsarray verändert wird.



EINE LEICHTE AUFGABE ALS EINSTIEG – LÖSUNG

Lösung 26: Was liefert dieser Code?

Der Code liefert 22 und 42. Ersterer Wert, da wir das Array ja so zuweisen, den zweiten, weil durch die Übergabe des Arrays mittels *call-by-reference* auch das Ursprungsarray verändert wird.

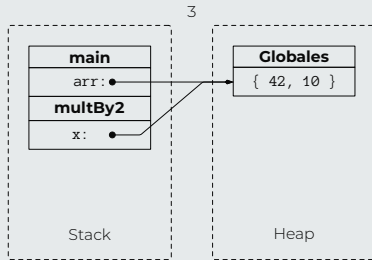


Lösung 26: Was liefert dieser Code?

(Fortsetzung)

Lösung 26: Was liefert dieser Code?

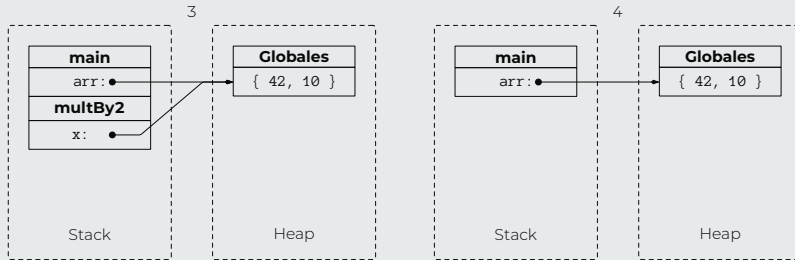
(Fortsetzung)



EINE LEICHTE AUFGABE ALS EINSTIEG – LÖSUNG

Lösung 26: Was liefert dieser Code?

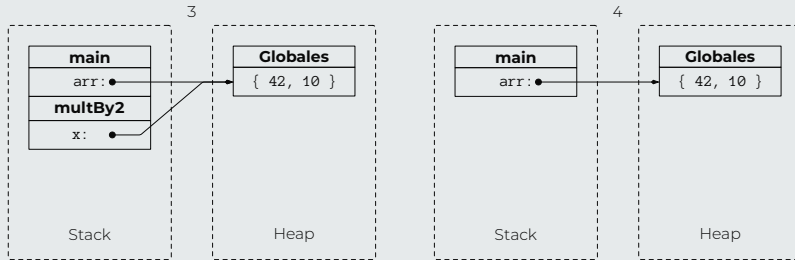
(Fortsetzung)



EINE LEICHTE AUFGABE ALS EINSTIEG – LÖSUNG

Lösung 26: Was liefert dieser Code?

(Fortsetzung)



Hinweis: eine derartige Grafik wird nicht gefordert werden, sie kann aber durchaus beim Verständnis der Thematik hilfreich sein.

DER NETTE BRUDER: CALL-BY-VALUE

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst,

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*,

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*, man übergibt also den eigentlichen Wert der Variable („was auf dem Stack liegt“).

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*, man übergibt also den eigentlichen Wert der Variable („was auf dem Stack liegt“).
- Wenn man komplexe Datentypen kopieren möchte,

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*, man übergibt also den eigentlichen Wert der Variable („was auf dem Stack liegt“).
- Wenn man komplexe Datentypen kopieren möchte, erstellt man sie in der Regel neu und greift dabei auf die *call-by-value*-Charakteristik der primitiven Datentypen zurück.

ÜBUNGSAUFGABE

Aufgabe 27: Fehler finden, IV

(3 Minuten)

Finden und korrigieren Sie alle (syntaktischen) Fehler:

Aufgabe 27: Fehler finden, IV

(3 Minuten)

Finden und korrigieren Sie alle (syntaktischen) Fehler:

```
class Laenge {
    final double METER KILOMETER = 1_000;
    static float meter2kilometer(final int meter){
        return meter/METER KILOMETER;
    }
    static int kilometer2meter(double kilometer){
        return kilometer * METER KILOMETER;
    }
}
```


Lösung 27: Fehler finden, IV

```
class Laenge {
    static final double METER_KILOMETER = 1_000;
    static float meter2kilometer(final int meter){
        return (float) (meter/METER_KILOMETER);
    }
    static int kilometer2meter(double kilometer){
        return (int) (kilometer * METER_KILOMETER);
    }
}
```

Lösung 27: Fehler finden, IV

(Fortsetzung)

Lösung 27: Fehler finden, IV

(Fortsetzung)

1. Die Konstante `METER KILOMETER` darf kein Leerfeld enthalten.

Lösung 27: Fehler finden, IV

(Fortsetzung)

1. Die Konstante `METER KILOMETER` darf kein Leerfeld enthalten.
2. Die Konstante muss **static** sein.

Lösung 27: Fehler finden, IV

(Fortsetzung)

1. Die Konstante `METER KILOMETER` darf kein Leerfeld enthalten.
2. Die Konstante muss **static** sein.
3. In `meter2kilometer` und `kilometer2meter` muss eine explizite Konvertierung erfolgen.

Lösung 27: Fehler finden, IV

(Fortsetzung)

1. Die Konstante `METER KILOMETER` darf kein Leerfeld enthalten.
2. Die Konstante muss **static** sein.
3. In `meter2kilometer` und `kilometer2meter` muss eine explizite Konvertierung erfolgen.

*Hinweis: das **final** in den Parametern und der Unterstrich in `1000` sind keine Fehler!*

ÜBUNGSaufGABE

Aufgabe 28: Kreisklasse

(4 Minuten)

Schreiben Sie eine Klasse `Circle`, die einen Kreis repräsentiert. Ein Kreis besitzt eine x und eine y Koordinate im Fließkommabereich, sowie einen Radius.

Aufgabe 28: Kreisklasse

(4 Minuten)

Schreiben Sie eine Klasse `Circle`, die einen Kreis repräsentiert. Ein Kreis besitzt eine x und eine y Koordinate im Fließkommabereich, sowie einen Radius.

Es soll möglich sein, den Umfang sowie die Fläche des Kreises berechnen zu lassen.

Aufgabe 28: Kreisklasse

(4 Minuten)

Schreiben Sie eine Klasse `Circle`, die einen Kreis repräsentiert. Ein Kreis besitzt eine x und eine y Koordinate im Fließkommabereich, sowie einen Radius.

Es soll möglich sein, den Umfang sowie die Fläche des Kreises berechnen zu lassen. Auf Datenkapselung muss hierbei keine Rücksicht genommen werden,

Aufgabe 28: Kreisklasse

(4 Minuten)

Schreiben Sie eine Klasse `Circle`, die einen Kreis repräsentiert. Ein Kreis besitzt eine x und eine y Koordinate im Fließkommabereich, sowie einen Radius.

Es soll möglich sein, den Umfang sowie die Fläche des Kreises berechnen zu lassen. Auf Datenkapselung muss hierbei keine Rücksicht genommen werden, allerdings muss es möglich sein den Kreis unter der Angabe aller Attribute zu konstruieren.

Aufgabe 28: Kreisklasse

(4 Minuten)

Schreiben Sie eine Klasse `Circle`, die einen Kreis repräsentiert. Ein Kreis besitzt eine x und eine y Koordinate im Fließkommabereich, sowie einen Radius.

Es soll möglich sein, den Umfang sowie die Fläche des Kreises berechnen zu lassen. Auf Datenkapselung muss hierbei keine Rücksicht genommen werden, allerdings muss es möglich sein den Kreis unter der Angabe aller Attribute zu konstruieren. ($U = 2 \cdot r \cdot \pi$, $A = r^2 \cdot \pi$)

Lösung 28: Kreisklasse

Lösung 28: Kreisklasse

```
class Circle {
```

```
}
```

Lösung 28: Kreisklasse

```
class Circle {  
    public double x, y, r;  
  
}
```

Lösung 28: Kreisklasse

```
class Circle {  
    public double x, y, r;  
    public Circle(double _x, double _y, double _r) {  
        x = _x; y = _y; r = _r;  
    }  
  
}
```

Lösung 28: Kreisklasse

```
class Circle {  
    public double x, y, r;  
    public Circle(double _x, double _y, double _r) {  
        x = _x; y = _y; r = _r;  
    }  
    public double getCircumference(){  
        return 2 * r * Math.PI;  
    }  
  
}
```

Lösung 28: Kreisklasse

```
class Circle {  
    public double x, y, r;  
    public Circle(double _x, double _y, double _r) {  
        x = _x; y = _y; r = _r;  
    }  
    public double getCircumference(){  
        return 2 * r * Math.PI;  
    }  
  
    public double getArea() { return r * r * Math.PI; }  
}
```

ÜBUNGSAUFGABE

Aufgabe 29: Schienennetz

(4 Minuten)

Aufgabe 29: Schienennetz

(4 Minuten)

Ein Feld in einem gitterartigen Schienennetz kann entweder

Aufgabe 29: Schienennetz

(4 Minuten)

Ein Feld in einem gitterartigen Schienennetz kann entweder frei ($\hat{=}$ keine Schiene) oder von einer Schiene, von einem Waggon oder einer Lokomotive „belegt“ sein.

Aufgabe 29: Schienennetz

(4 Minuten)

Ein Feld in einem gitterartigen Schienennetz kann entweder frei ($\hat{=}$ keine Schiene) oder von einer Schiene, von einem Waggon oder einer Lokomotive „belegt“ sein.

Schreiben Sie eine Enumeration, die einen Feldzustand darstellt.

Aufgabe 29: Schienennetz

(4 Minuten)

Ein Feld in einem gitterartigen Schienennetz kann entweder frei ($\hat{=}$ keine Schiene) oder von einer Schiene, von einem Waggon oder einer Lokomotive „belegt“ sein.

Schreiben Sie eine Enumeration, die einen Feldzustand darstellt. Jede Konstante soll dabei gleich mitspeichern, ob das Feld von einer Lok befahrbar ist (

Aufgabe 29: Schienennetz

(4 Minuten)

Ein Feld in einem gitterartigen Schienennetz kann entweder frei ($\hat{=}$ keine Schiene) oder von einer Schiene, von einem Waggon oder einer Lokomotive „belegt“ sein.

Schreiben Sie eine Enumeration, die einen Feldzustand darstellt. Jede Konstante soll dabei gleich mitspeichern, ob das Feld von einer Lok befahrbar ist (dies nur für „Schienefelder“ der Fall).

Lösung 29: Schienennetz

Lösung 29: Schienennetz

```
enum Feld {
```

```
}
```

Lösung 29: Schienennetz

```
enum Feld {  
    FREI(false),  
    SCHIENE(true),  
    WAGGON(false),  
    LOKOMOTIVE(false);  
  
}
```


Lösung 29: Schienennetz

```
enum Feld {  
    FREI(false),  
    SCHIENE(true),  
    WAGGON(false),  
    LOKOMOTIVE(false);  
  
    boolean befahrbar;  
  
}
```

Lösung 29: Schienennetz

```
enum Feld {  
    FREI(false),  
    SCHIENE(true),  
    WAGGON(false),  
    LOKOMOTIVE(false);  
  
    boolean befahrbar;  
    Feld(boolean befahrbar) {  
        this.befahrbar = befahrbar;  
    }  
}
```

ÜBUNGSaufGABE

Aufgabe 30: Kamera

(5 Minuten)

Aufgabe 30: Kamera

(5 Minuten)

Schreiben Sie eine **Camera** Klasse,

Aufgabe 30: Kamera

(5 Minuten)

Schreiben Sie eine **Camera** Klasse, mit drei Fließkommazahlen für die aktuelle Position (x,y,z) .

Aufgabe 30: Kamera

(5 Minuten)

Schreiben Sie eine **Camera** Klasse, mit drei Fließkommazahlen für die aktuelle Position (x,y,z) . Von der Klasse soll es von außen nicht möglich sein, mehr als eine Instanz zu erzeugen.

Aufgabe 30: Kamera

(5 Minuten)

Schreiben Sie eine **Camera** Klasse, mit drei Fließkommazahlen für die aktuelle Position (x,y,z) . Von der Klasse soll es von außen nicht möglich sein, mehr als eine Instanz zu erzeugen.

Weitere „Anfragen“ sollen dasselbe Objekt zurückliefern.

Lösung 30: Kamera (Singleton)

Lösung 30: Kamera (Singleton)

```
public class Camera {  
    double x, y, z;  
  
    private Camera() {  
        x = y = z = 0.0;  
    }  
  
}
```

Lösung 30: Kamera (Singleton)

```
public class Camera {  
    double x, y, z;  
    private static Camera instance;  
    private Camera() {  
        x = y = z = 0.0;  
    }  
  
}
```

Lösung 30: Kamera (Singleton)

```
public class Camera {  
    double x, y, z;  
    private static Camera instance;  
    private Camera() {  
        x = y = z = 0.0;  
    }  
    public static Camera getInstance() {  
  
    }  
}
```

Lösung 30: Kamera (Singleton)

```
public class Camera {  
    double x, y, z;  
    private static Camera instance;  
    private Camera() {  
        x = y = z = 0.0;  
    }  
    public static Camera getInstance() {  
        if(instance == null) instance = new Camera();  
    }  
}
```


Lösung 30: Kamera (Singleton)

```
public class Camera {  
    double x, y, z;  
    private static Camera instance;  
    private Camera() {  
        x = y = z = 0.0;  
    }  
    public static Camera getInstance() {  
        if(instance == null) instance = new Camera();  
        return instance;  
    }  
}
```

Programmiertheorie



REKURSIVE UND ITERATIVE PROGRAMMIERUNG

Definition 5: Rekursive Methode

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
- *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
 - *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.
- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
 - *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.
- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:
ABBRUCHBEDINGUNG: Das lösbares Teilproblem. Ende der Rekursion. Hier wird die Methode nicht weiter referenziert.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
 - *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.
- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:
ABBRUCHBEDINGUNG: Das lösbar Teilproblem. Ende der Rekursion. Hier wird die Methode nicht weiter referenziert.
REKURSIVER ZWEIG: Enthält die rekursiv auszuführende Prozedur.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
 - *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.
-
- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:
ABBRUCHBEDINGUNG: Das lösbare Teilproblem. Ende der Rekursion. Hier wird die Methode nicht weiter referenziert.
REKURSIVER ZWEIG: Enthält die rekursiv auszuführende Prozedur.
-
- Rekursion und Iteration sind *gleichmächtig*. Iteration ist in der Regel „performanter“, Rekursion ist oft „eleganter“ (rekursive Datenstrukturen, ...).

ARTEN VON REKURSION: HEAD UND TAIL

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion.

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){
```

```
}
```

```
public void tailDecrement(int i){
```

```
}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
  
}
```

```
public void tailDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
  
}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
    // Rekursion  
    else headDecrement(i - 1);  
  
}
```

```
public void tailDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
  
}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
    // Rekursion  
    else headDecrement(i - 1);  
  
}
```

```
public void tailDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
    // Verarbeitung  
    else System.out.println(i);  
  
}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
    // Rekursion  
    else headDecrement(i - 1);  
    // Verarbeitung  
    System.out.println(i);  
}
```

```
public void tailDecrement(int i){  
    // Abbruchbedingung  
    if(i == 0) return;  
    // Verarbeitung  
    else System.out.println(i);  
    // Rekursion  
    tailDecrement(i - 1);  
}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
 - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
 - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
// → 1 2 3 4 5 6 ...
public void headDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Rekursion
    else headDecrement(i - 1);
    // Verarbeitung
    System.out.println(i);
}
```

```
// → 10 9 8 7 6 ...
public void tailDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Verarbeitung
    else System.out.println(i);
    // Rekursion
    tailDecrement(i - 1);
}
```

ARTEN VON REKURSION: BAUMARTIGE REKURSION

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg,

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.
- Beliebte Beispiele: die Fibonaccifolge und der Binomialkoeffizient.

ARTEN VON REKURSION: BAUMARTIGE REKURSION

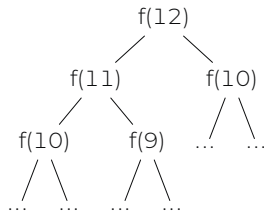
- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.
- Beliebte Beispiele: die Fibonaccifolge und der Binomialkoeffizient.

```
int fib(int n) {  
    if(n <= 1) return 1;  
    return fib(n - 1) + fib(n - 2);  
}
```

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.
- Beliebte Beispiele: die Fibonaccifolge und der Binomialkoeffizient.

```
int fib(int n) {  
    if(n <= 1) return 1;  
    return fib(n - 1) + fib(n - 2);  
}
```



VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:
GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf.

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:
GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf. Beispielsweise die Ackermannfunktion (nach Péter):

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf. Beispielsweise die Ackermannfunktion (nach Péter):

```
int a(int n, int m) {  
    if(n == 0) return m + 1;  
    else if (m == 0) return a(n - 1, 1);  
    else return a(n - 1, a(n, m - 1));  
}
```

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf. Beispielsweise die Ackermannfunktion (nach Péter):

```
int a(int n, int m) {  
    if(n == 0) return m + 1;  
    else if (m == 0) return a(n - 1, 1);  
    else return a(n - 1, a(n, m - 1));  
}
```

VERSCHRÄNKT: Hier rufen sich mehrere Funktionen rekursiv gegenseitig auf. Diese Variante lässt sich schwer bis gar nicht (direkt) in eine Schleife übersetzen.

ETWAS REKURSION

Aufgabe 31: Formel zu Rekursion

(4 Minuten)

Aufgabe 31: Formel zu Rekursion

(4 Minuten)

Implementieren Sie eine rekursive Java-Methode, welche die folgende Funktion berechnet ($a, b, c \in \mathbb{N}$).

$$f(a, b, c) = \begin{cases} \max(a, b) & \text{wenn } c = 0, \\ f(a + b, b - 1, c) & \text{wenn } b \geq 0, \\ f(a - 1, b - 1, |c - 1|) & \text{sonst.} \end{cases}$$

Aufgabe 31: Formel zu Rekursion

(4 Minuten)

Implementieren Sie eine rekursive Java-Methode, welche die folgende Funktion berechnet ($a, b, c \in \mathbb{N}$).

$$f(a, b, c) = \begin{cases} \max(a, b) & \text{wenn } c = 0, \\ f(a + b, b - 1, c) & \text{wenn } b \geq 0, \\ f(a - 1, b - 1, |c - 1|) & \text{sonst.} \end{cases}$$

Approximieren Sie \mathbb{N} durch den Java Datentyp `long`. Sie dürfen *keine* bestehenden Java Funktionen (wie beispielsweise `Math::abs`) benutzen.

Welchen Wert liefert die Berechnung von $f(12, 3, -2)$?

Lösung 31: Formel zu Rekursion

Lösung 31: Formel zu Rekursion

```
public long f(long a, long b, long c) {  
    if(c == 0)  
        return a > b ? a : b;  
  
}
```

Lösung 31: Formel zu Rekursion

```
public long f(long a, long b, long c) {  
    if(c == 0)  
        return a > b ? a : b;  
    if(b >= 0)  
        return f(a + b, b - 1, c);  
}
```

Lösung 31: Formel zu Rekursion

```
public long f(long a, long b, long c) {  
    if(c == 0)  
        return a > b ? a : b;  
    if(b >= 0)  
        return f(a + b, b - 1, c);  
    return f(a - 1, b - 1, c > 0 ? (c - 1) : -(c - 1));  
}
```

Lösung 31: Formel zu Rekursion

```
public long f(long a, long b, long c) {  
    if(c == 0)  
        return a > b ? a : b;  
    if(b >= 0)  
        return f(a + b, b - 1, c);  
    return f(a - 1, b - 1, c > 0 ? (c - 1) : -(c - 1));  
}
```

Weiter gilt $f(12, 3, -2) = 14$.

Lösung 31: Formel zu Rekursion

```
public long f(long a, long b, long c) {  
    if(c == 0)  
        return a > b ? a : b;  
    if(b >= 0)  
        return f(a + b, b - 1, c);  
    return f(a - 1, b - 1, c > 0 ? (c - 1) : -(c - 1));  
}
```

Weiter gilt $f(12, 3, -2) = 14$. Warum?

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Für $f(12, 3, -2)$ gilt mit $c = -2 \neq 0$ und $b = 3 \geq 0$ die Berechnung:
 $f(a + b, b - 1, c)$. Damit erhalten wir:

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Für $f(12, 3, -2)$ gilt mit $c = -2 \neq 0$ und $b = 3 \geq 0$ die Berechnung:
 $f(a + b, b - 1, c)$. Damit erhalten wir:

$$f(12 + 3, 3 - 1, -2) = f(15, 2, -2)$$

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Für $f(12, 3, -2)$ gilt mit $c = -2 \neq 0$ und $b = 3 \geq 0$ die Berechnung:
 $f(a + b, b - 1, c)$. Damit erhalten wir:

$$f(12 + 3, 3 - 1, -2) = f(15, 2, -2)$$

Damit ist wieder $c = -2 \neq 0$ und $b = 2 \geq 0$. Daraus ergibt sich:

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Für $f(12, 3, -2)$ gilt mit $c = -2 \neq 0$ und $b = 3 \geq 0$ die Berechnung:
 $f(a + b, b - 1, c)$. Damit erhalten wir:

$$f(12 + 3, 3 - 1, -2) = f(15, 2, -2)$$

Damit ist wieder $c = -2 \neq 0$ und $b = 2 \geq 0$. Daraus ergibt sich:

$$f(15 + 2, 2 - 1, -2) = f(17, 1, -2)$$

$$f(17 + 1, 1 - 1, -2) = f(18, 0, -2)$$

$$f(18 + 0, 0 - 1, -2) = f(18, -1, -2)$$

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Für $f(12, 3, -2)$ gilt mit $c = -2 \neq 0$ und $b = 3 \geq 0$ die Berechnung:
 $f(a + b, b - 1, c)$. Damit erhalten wir:

$$f(12 + 3, 3 - 1, -2) = f(15, 2, -2)$$

Damit ist wieder $c = -2 \neq 0$ und $b = 2 \geq 0$. Daraus ergibt sich:

$$f(15 + 2, 2 - 1, -2) = f(17, 1, -2)$$

$$f(17 + 1, 1 - 1, -2) = f(18, 0, -2)$$

$$f(18 + 0, 0 - 1, -2) = f(18, -1, -2)$$

Mit $b = -1 \not\geq 0$ betreten wir nun den „sonst“-Teil.

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Nun bei $f(18, -1, -2)$ berechnen wir $f(a - 1, b - 1, |c - 1|)$:

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Nun bei $f(18, -1, -2)$ berechnen wir $f(a - 1, b - 1, |c - 1|)$:

$$f(18 - 1, -1 - 1, |-2 - 1|) = f(17, -2, 3)$$

$$f(17 - 1, -2 - 1, |3 - 1|) = f(16, -3, 2)$$

$$f(16 - 1, -3 - 1, |2 - 1|) = f(15, -4, 1)$$

$$f(15 - 1, -4 - 1, |1 - 1|) = f(14, -5, 0)$$

Lösung 31: Formel zu Rekursion

(Fortsetzung)

Nun bei $f(18, -1, -2)$ berechnen wir $f(a - 1, b - 1, |c - 1|)$:

$$f(18 - 1, -1 - 1, |-2 - 1|) = f(17, -2, 3)$$

$$f(17 - 1, -2 - 1, |3 - 1|) = f(16, -3, 2)$$

$$f(16 - 1, -3 - 1, |2 - 1|) = f(15, -4, 1)$$

$$f(15 - 1, -4 - 1, |1 - 1|) = f(14, -5, 0)$$

Mit $c = 0$ terminiert die Funktion für $\max(14, -5) = 14$.

DAS PARADIGMA: DIVIDE AND CONQUER

DAS PARADIGMA: DIVIDE AND CONQUER

- Manche komplizierte Probleme,

DAS PARADIGMA: DIVIDE AND CONQUER

- Manche komplizierte Probleme, lassen sich durch Rekursion in immer kleinere Probleme aufspalten, die dann beherrschbarer sind.

DAS PARADIGMA: DIVIDE AND CONQUER

- Manche komplizierte Probleme, lassen sich durch Rekursion in immer kleinere Probleme aufspalten, die dann beherrschbarer sind.
- Dies wird uns bei den Sortieralgorithmen Merge- und Quicksort wieder begegnen.

DAS PARADIGMA: BACKTRACKING

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.
- Das Problem wird von einer Teillösung aus bis zur Gesamtlösung erweitert.

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.
- Das Problem wird von einer Teillösung aus bis zur Gesamtlösung erweitert. Eine „Sackgasse“ veranlasst einen neuen Versuch (trial and error).

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.
- Das Problem wird von einer Teillösung aus bis zur Gesamtlösung erweitert. Eine „Sackgasse“ veranlasst einen neuen Versuch (trial and error).
- Im Sackgassen-Fall macht man die Entscheidungen solange rückgängig, bis man eine andere Erweiterung wählen kann und erweitert die Teillösung dann durch diese.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:*

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth*: Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth*: Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth*: Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku*.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth*: Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku*. Füge in erstes freies Feld eine der dort möglichen Zahlen ein.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth*: Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku*. Füge in erstes freies Feld eine der dort möglichen Zahlen ein. Verfahre so, bis alle Felder gefüllt sind oder für ein Feld keine Zahl mehr möglich ist.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth*: Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku*. Füge in erstes freies Feld eine der dort möglichen Zahlen ein. Verfahre so, bis alle Felder gefüllt sind oder für ein Feld keine Zahl mehr möglich ist. In diesem Fall: springe zum letzten Punkt zurück, an dem noch andere Zahlen möglich sind. Probiere weiter.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth*: Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku*. Füge in erstes freies Feld eine der dort möglichen Zahlen ein. Verfahre so, bis alle Felder gefüllt sind oder für ein Feld keine Zahl mehr möglich ist. In diesem Fall: springe zum letzten Punkt zurück, an dem noch andere Zahlen möglich sind. Probiere weiter.
- Rucksackproblem, 8-Damen Problem, ...

ZUR VERTIEFUNG

DIE LIEBE ZUR REKURSION

Der Java-Stack, Funktionsaufrufe und Rekursion.
Ein wiederkehrendes Dilemma.

Florian Sihler

27. Juni 2021
SP, Universität Ulm

DIE LIEBE ZUR REKURSION

Der Java-Stack, Funktionsaufrufe und Rekursion.
Ein wiederkehrendes Dilemma.

Florian Sihler

27. Juni 2021
SP, Universität Ulm



Mehr zum Thema „Rekursion“ per Klick...

KOMPLEXITÄTSBETRACHTUNG

KOMPLEXITÄTSBETRACHTUNG

- Da die Ausführungszeit eines Programms von vielen Parametern

KOMPLEXITÄTSBETRACHTUNG

- Da die Ausführungszeit eines Programms von vielen Parametern (Taktrate des Prozessors, andere laufende Programme, ...) abhängig ist,

KOMPLEXITÄTSBETRACHTUNG

- Da die Ausführungszeit eines Programms von vielen Parametern (Taktrate des Prozessors, andere laufende Programme, ...) abhängig ist, betrachten wir oft nur dessen Skalierung.

- Da die Ausführungszeit eines Programms von vielen Parametern (Taktrate des Prozessors, andere laufende Programme, ...) abhängig ist, betrachten wir oft nur dessen Skalierung.

Definition 6: Effizienz

- Da die Ausführungszeit eines Programms von vielen Parametern (Taktrate des Prozessors, andere laufende Programme, ...) abhängig ist, betrachten wir oft nur dessen Skalierung.

Definition 6: Effizienz

Die Effizienz eines Programms wird durch dessen *Speicher-*, sowie *Laufzeitaufwand* bestimmt.

- Da die Ausführungszeit eines Programms von vielen Parametern (Taktrate des Prozessors, andere laufende Programme, ...) abhängig ist, betrachten wir oft nur dessen Skalierung.

Definition 6: Effizienz

Die Effizienz eines Programms wird durch dessen *Speicher-*, sowie *Laufzeitaufwand* bestimmt.

- Letztere werden wir ausführlicher betrachten. Genauer: In welcher Komplexitätsklasse liegt der Algorithmus?

KOMPLEXITÄTSBETRACHTUNG

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:
WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:
 - WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.
 - BEST-CASE: Die Laufzeitkomplexität im günstigsten Fall für den (spezifischen) Algorithmus.

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:
 - WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.
 - BEST-CASE: Die Laufzeitkomplexität im günstigsten Fall für den (spezifischen) Algorithmus.
 - AVERAGE-CASE: Die Laufzeitkomplexität im durchschnittlichen Fall für den (spezifischen) Algorithmus.

- Für die Laufzeitkomplexität unterscheidet man:
 - WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.
 - BEST-CASE: Die Laufzeitkomplexität im günstigsten Fall für den (spezifischen) Algorithmus.
 - AVERAGE-CASE: Die Laufzeitkomplexität im durchschnittlichen Fall für den (spezifischen) Algorithmus. Dies bezeichnet in der Regel gleichverteilt zufällige Eingaben.

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:
 - WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.
 - BEST-CASE: Die Laufzeitkomplexität im günstigsten Fall für den (spezifischen) Algorithmus.
 - AVERAGE-CASE: Die Laufzeitkomplexität im durchschnittlichen Fall für den (spezifischen) Algorithmus. Dies bezeichnet in der Regel gleichverteilt zufällige Eingaben.
- Dabei werden wir den *average-case* vernachlässigen.

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen: $2 + n$

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen: $2 + n$
- Vergleiche:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen: $2 + n$
- Vergleiche: $n + 1 + \frac{n(n+1)}{2}$

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen: $2 + n$
- Vergleiche: $n + 1 + \frac{n(n+1)}{2}$
- Inkrementierungen:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen: $2 + n$
- Vergleiche: $n + 1 + \frac{n(n+1)}{2}$
- Inkrementierungen: $n + \frac{n(n-1)}{2}$

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen: $2 + n$
- Vergleiche: $n + 1 + \frac{n(n+1)}{2}$
- Inkrementierungen: $n + \frac{n(n-1)}{2}$
- Dekrementierungen:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {  
    int count = 2;  
    for(int i = 1; i <= n; i++) {  
        for(int j = n; j > i; j--)  
            count++;  
    }  
    return count;  
}
```

- Zuweisungen: $2 + n$
- Vergleiche: $n + 1 + \frac{n(n+1)}{2}$
- Inkrementierungen: $n + \frac{n(n-1)}{2}$
- Dekrementierungen: $\frac{n(n-1)}{2}$

ERFASSEN DER KOMPLEXITÄT

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

Definition 7: \mathcal{O} -Notation

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

Definition 7: \mathcal{O} -Notation

Es gilt $T(n) \in \mathcal{O}(f(n))$, wenn $f(n)$ eine obere Schranke von $T(n)$ ist, also:

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

Definition 7: \mathcal{O} -Notation

Es gilt $T(n) \in \mathcal{O}(f(n))$, wenn $f(n)$ eine obere Schranke von $T(n)$ ist, also:

$$T(n) \in \mathcal{O}(f(n)) \iff \exists n_0 \in \mathbb{N} \ c \in \mathbb{R}^+ \ \forall n \geq n_0 : T(n) \leq c \cdot f(n).$$

ERFASSEN DER KOMPLEXITÄT

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel:

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen,

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$
Bsp:	42
Bez:	konst.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Bsp:	42	$4 \log(3n)$
Bez:	konst.	logarithm.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Bsp:	42	$4 \log(3n)$	$4n - 3$
Bez:	konst.	logarithm.	linear

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$
Bez:	konst.	logarithm.	linear	linear log.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$	$n^3 - 42n^2$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$	$n^3 - 42n^2$	$14 \cdot 2^n$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch	exponentiell

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, ...)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der \mathcal{O} Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$	$\mathcal{O}(n!)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$	$n^3 - 42n^2$	$14 \cdot 2^n$	$n! \cdot 10^{-42}$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch	exponentiell	faktoriell

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten,

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten, die es erlaubt ein Problem / Programm / Projekt aus verschiedenen Blickwinkeln zu betrachten.

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten, die es erlaubt ein Problem / Programm / Projekt aus verschiedenen Blickwinkeln zu betrachten.
- Im Kontext der Vorlesung gilt es drei Typen kurz zu skizzieren:

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten, die es erlaubt ein Problem / Programm / Projekt aus verschiedenen Blickwinkeln zu betrachten.
- Im Kontext der Vorlesung gilt es drei Typen kurz zu skizzieren:
KLASSENDIAGRAMME: modellieren die Beziehungen und Eigenschaften der beteiligten Klassen.

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten, die es erlaubt ein Problem / Programm / Projekt aus verschiedenen Blickwinkeln zu betrachten.
- Im Kontext der Vorlesung gilt es drei Typen kurz zu skizzieren:
 - KLASSENDIAGRAMME: modellieren die Beziehungen und Eigenschaften der beteiligten Klassen.
 - OBJEKTDIAGRAMME: modellieren die Beziehungen und Ausprägungen (spezifischer) Objekte.

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten, die es erlaubt ein Problem / Programm / Projekt aus verschiedenen Blickwinkeln zu betrachten.
- Im Kontext der Vorlesung gilt es drei Typen kurz zu skizzieren:
 - KLASSENDIAGRAMME: modellieren die Beziehungen und Eigenschaften der beteiligten Klassen.
 - OBJEKTDIAGRAMME: modellieren die Beziehungen und Ausprägungen (spezifischer) Objekte.
 - SEQUENZDIAGRAMME: modellieren den Nachrichtenaustausch in einem Programm.

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten, die es erlaubt ein Problem / Programm / Projekt aus verschiedenen Blickwinkeln zu betrachten.
- Im Kontext der Vorlesung gilt es drei Typen kurz zu skizzieren:
 - KLASSENDIAGRAMME: modellieren die Beziehungen und Eigenschaften der beteiligten Klassen.
 - OBJEKTDIAGRAMME: modellieren die Beziehungen und Ausprägungen (spezifischer) Objekte.
 - SEQUENZDIAGRAMME: modellieren den Nachrichtenaustausch in einem Programm. Sie sind ereignisbasiert.

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten, die es erlaubt ein Problem / Programm / Projekt aus verschiedenen Blickwinkeln zu betrachten.
- Im Kontext der Vorlesung gilt es drei Typen kurz zu skizzieren:
 - KLASSENDIAGRAMME: modellieren die Beziehungen und Eigenschaften der beteiligten Klassen.
 - OBJEKTDIAGRAMME: modellieren die Beziehungen und Ausprägungen (spezifischer) Objekte.
 - SEQUENZDIAGRAMME: modellieren den Nachrichtenaustausch in einem Programm. Sie sind ereignisbasiert.
- UML wird hier (wie in der Vorlesung auch) nur oberflächlich betrachtet.

UML – EIN ÜBERBLICK

UML – EIN ÜBERBLICK



UML

UML – EIN ÜBERBLICK

UML

Strukturdiagramme

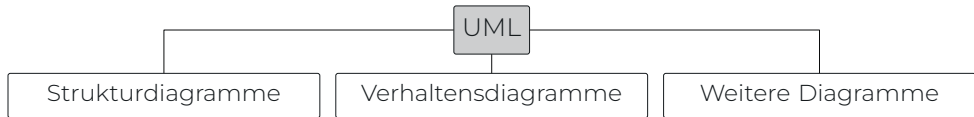
UML – EIN ÜBERBLICK

UML

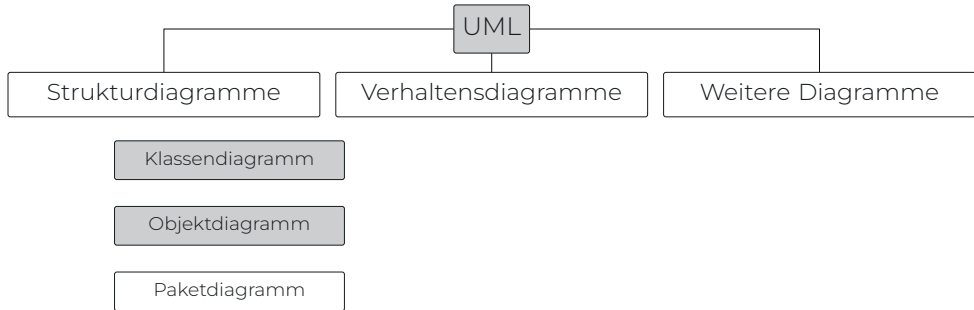
Strukturdiagramme

Verhaltensdiagramme

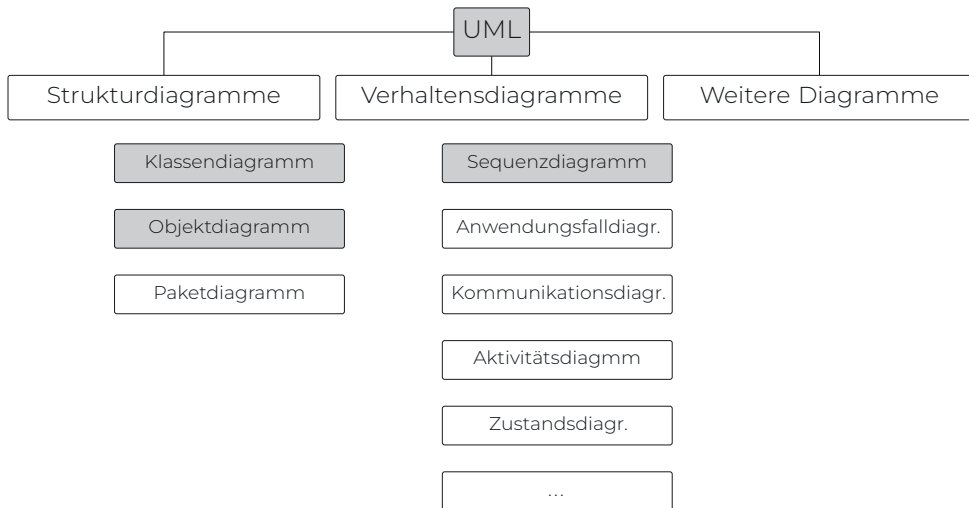
UML – EIN ÜBERBLICK



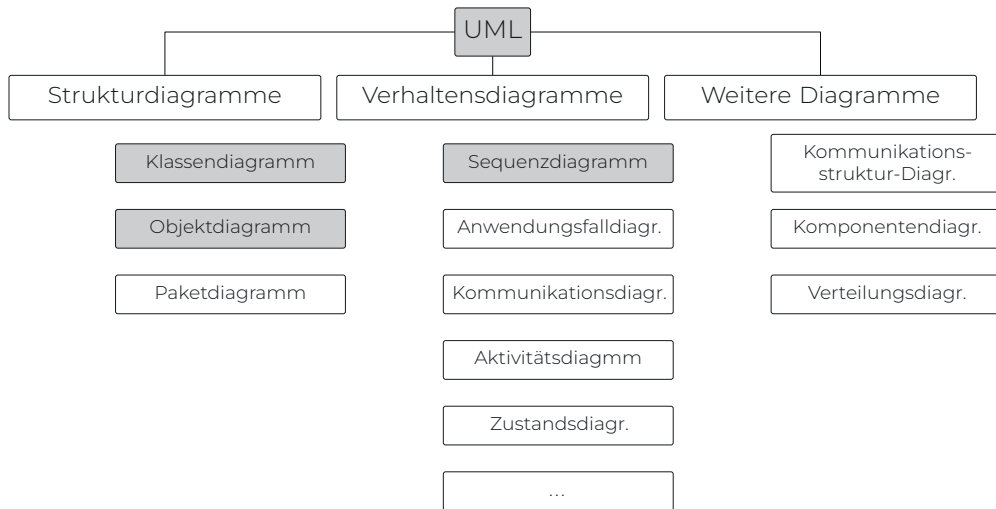
UML – EIN ÜBERBLICK



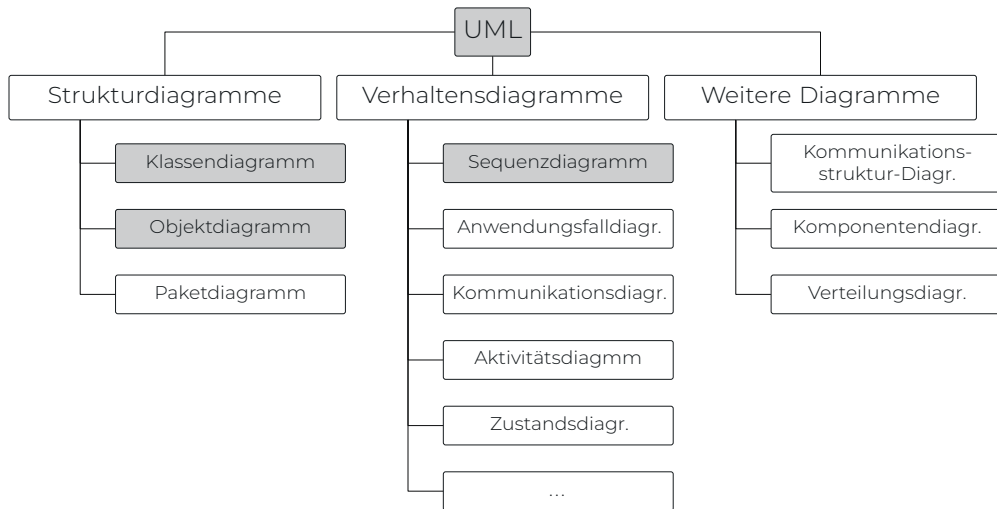
UML – EIN ÜBERBLICK



UML – EIN ÜBERBLICK

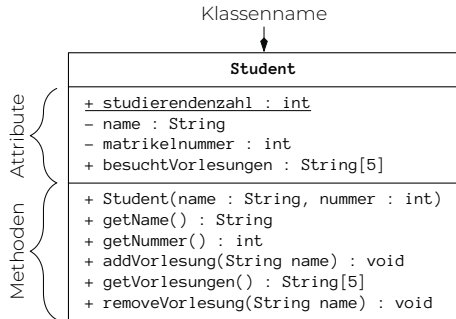


UML – EIN ÜBERBLICK



UML – KLASSENDIAGRAMME

UML – KLASSENDIAGRAMME



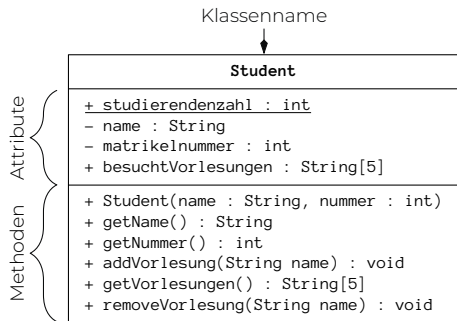
(-) steht für ein **private** Attribut

(+) für ein **public** Attribut

(#) für ein **protected** Attribut

static-Komponenten werden unterstrichen.

UML – KLASSENDIAGRAMME



(-) steht für ein **private** Attribut

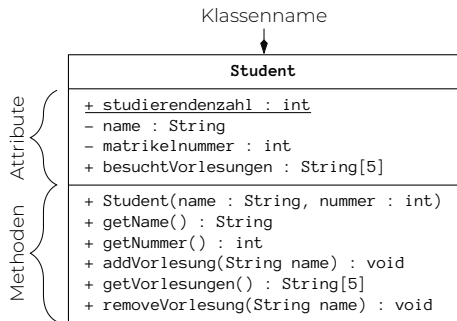
(+) für ein **public** Attribut

(#) für ein **protected** Attribut

static-Komponenten werden unterstrichen.

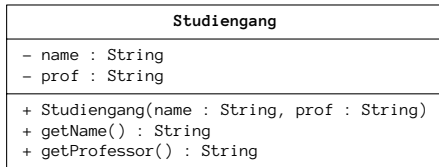
← bis zu 5 Vorlesungen

UML – KLASSENDIAGRAMME

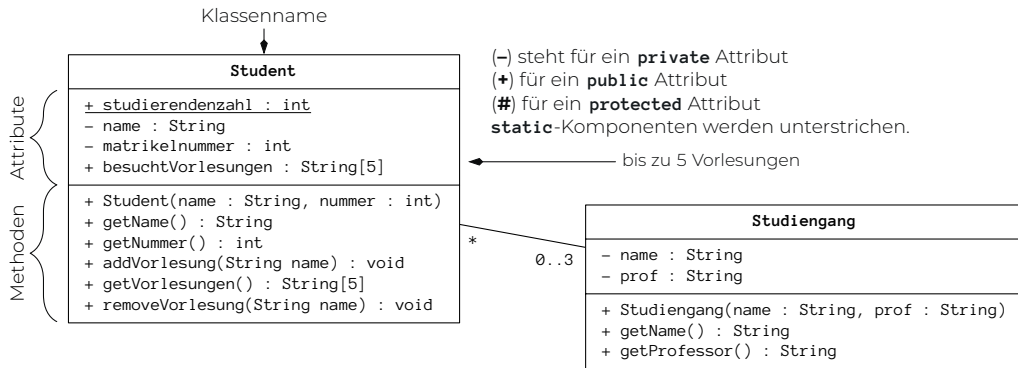


(-) steht für ein **private** Attribut
(+) für ein **public** Attribut
(#) für ein **protected** Attribut
static-Komponenten werden unterstrichen.

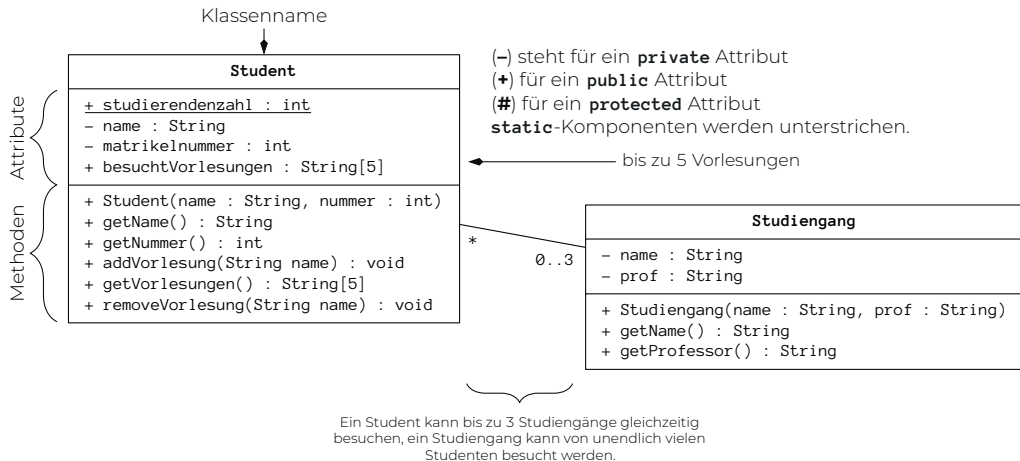
← bis zu 5 Vorlesungen



UML – KLASSENDIAGRAMME



UML – KLASSENDIAGRAMME



UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow \triangleright B).

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A \diamond — B).

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A \diamond — B). Markiert meist „Besitz“: „A besitzt ein B“.

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A ◇ — B). Markiert meist „Besitz“: „A besitzt ein B“.
 - Kompositionen (A ◆ — B).

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A \diamond — B). Markiert meist „Besitz“: „A besitzt ein B“.
 - Kompositionen (A \blacklozenge — B). Markiert meist „ist ein Teil von, mit gleicher Lebenszeit“: „A besteht aus B“.

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A \diamond — B). Markiert meist „Besitz“: „A besitzt ein B“.
 - Kompositionen (A \blacklozenge — B). Markiert meist „ist ein Teil von, mit gleicher Lebenszeit“: „A besteht aus B“.
- Pakete werden durch einen extra Kasten gekennzeichnet.

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A \diamond — B). Markiert meist „Besitz“: „A besitzt ein B“.
 - Kompositionen (A \blacklozenge — B). Markiert meist „ist ein Teil von, mit gleicher Lebenszeit“: „A besteht aus B“.
- Pakete werden durch einen extra Kasten gekennzeichnet.
- *Wichtig:* Attribute deren Typ eine andere Klasse ist werden durch eine Assoziation gekennzeichnet,

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A \diamond — B). Markiert meist „Besitz“: „A besitzt ein B“.
 - Kompositionen (A \blacklozenge — B). Markiert meist „ist ein Teil von, mit gleicher Lebenszeit“: „A besteht aus B“.
- Pakete werden durch einen extra Kasten gekennzeichnet.
- *Wichtig:* Attribute deren Typ eine andere Klasse ist werden durch eine Assoziation gekennzeichnet, dies gilt auch, wenn eine Klasse sich selbst referenziert

UML – KLASSENDIAGRAMME, II

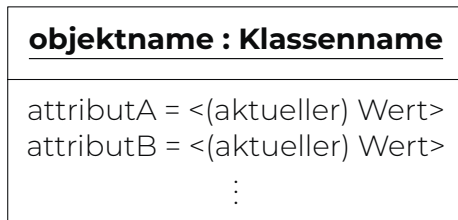
- Es gibt noch weitere Assoziationen.
- So gibt es:
 - gerichtete Assoziationen (A \longrightarrow B).
 - Abhängigkeiten (A - - - - - B).
 - Vererbungen (A \longrightarrow ▷ B).
 - Aggregationen (A \diamond — B). Markiert meist „Besitz“: „A besitzt ein B“.
 - Kompositionen (A \blacklozenge — B). Markiert meist „ist ein Teil von, mit gleicher Lebenszeit“: „A besteht aus B“.
- Pakete werden durch einen extra Kasten gekennzeichnet.
- *Wichtig:* Attribute deren Typ eine andere Klasse ist werden durch eine Assoziation gekennzeichnet, dies gilt auch, wenn eine Klasse sich selbst referenziert (LinkedList, ...).

- Objektdiagramme ähneln Klassendiagrammen.

- Objektdiagramme ähneln Klassendiagrammen.
- Allerdings handelt es sich immer um explizite Ausprägungen einer Klasse

- Objektdiagramme ähneln Klassendiagrammen.
- Allerdings handelt es sich immer um explizite Ausprägungen einer Klasse (\Rightarrow keine Methoden).

- Objektdiagramme ähneln Klassendiagrammen.
- Allerdings handelt es sich immer um explizite Ausprägungen einer Klasse (\Rightarrow keine Methoden).



UML – SEQUENZDIAGRAMME

- Bilden den Nachrichtenaustausch ab.

- Bilden den Nachrichtenaustausch ab. Das Senden und Empfangen wird auf Basis von Ereignissen ausgelöst,

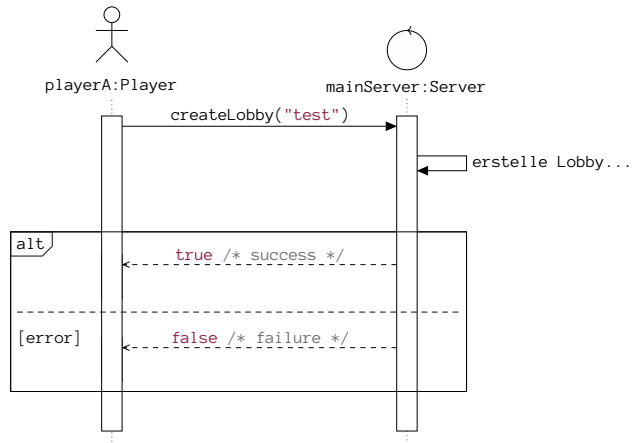
- Bilden den Nachrichtenaustausch ab. Das Senden und Empfangen wird auf Basis von Ereignissen ausgelöst, diese rufen wiederum Reaktionen hervor.

- Bilden den Nachrichtenaustausch ab. Das Senden und Empfangen wird auf Basis von Ereignissen ausgelöst, diese rufen wiederum Reaktionen hervor.
- Die Nachrichten können jeweils synchron (durchgezogene Linie) oder asynchron (gestrichelte Linie) ausgetauscht werden.

- Bilden den Nachrichtenaustausch ab. Das Senden und Empfangen wird auf Basis von Ereignissen ausgelöst, diese rufen wiederum Reaktionen hervor.
- Die Nachrichten können jeweils synchron (durchgezogene Linie) oder asynchron (gestrichelte Linie) ausgetauscht werden.
- Nachrichten selbst können Methodenaufrufe, Rückgabewerte oder externe Ereignisse sein (wie Zeitereignisse).

- Bilden den Nachrichtenaustausch ab. Das Senden und Empfangen wird auf Basis von Ereignissen ausgelöst, diese rufen wiederum Reaktionen hervor.
- Die Nachrichten können jeweils synchron (durchgezogene Linie) oder asynchron (gestrichelte Linie) ausgetauscht werden.
- Nachrichten selbst können Methodenaufrufe, Rückgabewerte oder externe Ereignisse sein (wie Zeitereignisse).
- Der Zeitliche Ablauf wird hierbei durch „Lebenslinien“ gekennzeichnet.

UML – SEQUENZDIAGRAMME, EIN BEISPIEL



DARSTELLUNG GANZER ZAHLEN

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

Definition 8: Stellenwertsystem

Bei einer Basis b und einer Zahl z aus den Ziffern $z = z_n z_{n-1} \dots z_0$

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

Definition 8: Stellenwertsystem

Bei einer Basis b und einer Zahl z aus den Ziffern $z = z_n z_{n-1} \dots z_0$ ergibt sich ihr Wert im uns bekannten Dezimalsystem ($b = 10$) durch:

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

Definition 8: Stellenwertsystem

Bei einer Basis b und einer Zahl z aus den Ziffern $z = z_n z_{n-1} \dots z_0$ ergibt sich ihr Wert im uns bekannten Dezimalsystem ($b = 10$) durch:

$$z_b = \sum_{i=0}^n z_i \cdot b^i$$

DARSTELLUNG GANZER ZAHLEN, II

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind:

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär),

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal)

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren,

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „**mod** b “ generieren

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „**mod** b “ generieren und die Zahl dann (ohne Rest) durch b teilen.

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „**mod** b “ generieren und die Zahl dann (ohne Rest) durch b teilen.
- Beispiel: $z = 10$ soll ins Dualsystem konvertiert werden:

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „**mod** b “ generieren und die Zahl dann (ohne Rest) durch b teilen.
- Beispiel: $z = 10$ soll ins Dualsystem konvertiert werden:

$$10 \div 2 = 5 \quad 10 \bmod 2 = 0 \quad (\leftarrow \text{LSB})$$

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „**mod** b “ generieren und die Zahl dann (ohne Rest) durch b teilen.
- Beispiel: $z = 10$ soll ins Dualsystem konvertiert werden:

$$10 \div 2 = 5 \quad 10 \bmod 2 = 0 \quad (\leftarrow \text{LSB})$$

$$5 \div 2 = 2 \quad 5 \bmod 2 = 1$$

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „**mod** b “ generieren und die Zahl dann (ohne Rest) durch b teilen.
- Beispiel: $z = 10$ soll ins Dualsystem konvertiert werden:

$$10 \div 2 = 5 \quad 10 \bmod 2 = 0 \quad (\leftarrow \text{LSB})$$

$$5 \div 2 = 2 \quad 5 \bmod 2 = 1$$

$$2 \div 2 = 1 \quad 2 \bmod 2 = 0$$

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \hat{=} 10$ bis $F \hat{=} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „**mod** b “ generieren und die Zahl dann (ohne Rest) durch b teilen.
- Beispiel: $z = 10$ soll ins Dualsystem konvertiert werden:

$$\begin{array}{ll} 10 \div 2 = 5 & 10 \bmod 2 = 0 \quad (\leftarrow \text{LSB}) \\ 5 \div 2 = 2 & 5 \bmod 2 = 1 \\ 2 \div 2 = 1 & 2 \bmod 2 = 0 \\ 1 \div 2 = 0 & 1 \bmod 2 = 1 \quad (\leftarrow \text{MSB}) \end{array}$$

DARSTELLUNG GANZER ZAHLEN, III

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren,

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren, die es zum Beispiel erlauben, das Hexadezimalsystem direkt ins Dualsystem zu konvertieren.

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren, die es zum Beispiel erlauben, das Hexadezimalsystem direkt ins Dualsystem zu konvertieren. Da jede Ziffer im Hexadezimalsystem vier Bits einnimmt, geht die Konvertierung schneller:

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren, die es zum Beispiel erlauben, das Hexadezimalsystem direkt ins Dualsystem zu konvertieren. Da jede Ziffer im Hexadezimalsystem vier Bits einnimmt, geht die Konvertierung schneller:

$$AFFE_{(16)} = \overbrace{1010}^A \overbrace{1111}^F \overbrace{1111}^F \overbrace{1110}^E \quad (2)$$

Aufgabe 32: Zahlen konvertieren

(3 Minuten)

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren, die es zum Beispiel erlauben, das Hexadezimalsystem direkt ins Dualsystem zu konvertieren. Da jede Ziffer im Hexadezimalsystem vier Bits einnimmt, geht die Konvertierung schneller:

$$AFFE_{(16)} = \overbrace{1010}^A \overbrace{1111}^F \overbrace{1111}^F \overbrace{1110}^E \quad (2)$$

Aufgabe 32: Zahlen konvertieren

(3 Minuten)

Konvertieren Sie $42_{(8)}$ zur Basis $b = 16$, $b = 3$ und ins Binärsystem.

ÜBUNGSAUFGABE – LÖSUNG

Lösung 32: Zahlen konvertieren

Konvertieren wir die Zahl zuerst ins Binärsystem (schnelle Konvertierung mit je drei Bits),

Lösung 32: Zahlen konvertieren

Konvertieren wir die Zahl zuerst ins Binärsystem (schnelle Konvertierung mit je drei Bits), dann von dort aus ins Hexadezimalsystem,

Lösung 32: Zahlen konvertieren

Konvertieren wir die Zahl zuerst ins Binärsystem (schnelle Konvertierung mit je drei Bits), dann von dort aus ins Hexadezimalsystem, nun ins Dezimalsystem (nicht gefordert) und dann in $b = 3$:

Lösung 32: Zahlen konvertieren

Konvertieren wir die Zahl zuerst ins Binärsystem (schnelle Konvertierung mit je drei Bits), dann von dort aus ins Hexadezimalsystem, nun ins Dezimalsystem (nicht gefordert) und dann in $b = 3$:

$$B = 2: 42_{(8)} = \overbrace{1000}^4 \overbrace{010}^2_{(2)}$$

$$B = 16: 100010_{(2)} = \overbrace{0010}^2 \overbrace{0010}^2_{(16)} \text{ (wir füllen also links mit Nullen auf, wenn die Bits kein Vielfaches von vier sind)}$$

$$B = 10: 4 \cdot 8^1 + 2 \cdot 8^0 = 32 + 2 = 34_{(10)}$$

Lösung 32: Zahlen konvertieren

(Fortsetzung)

$b = 3$: Wir verwenden sukzessive Division um von $34_{(10)}$ auf $b = 3$ zu kommen:

Lösung 32: Zahlen konvertieren

(Fortsetzung)

$b = 3$: Wir verwenden sukzessive Division um von $34_{(10)}$ auf $b = 3$ zu kommen:

$$34 \div 3 = 11 \quad 34 \bmod 3 = 1 (\leftarrow \text{LSB})$$

Damit ergibt sich:

Lösung 32: Zahlen konvertieren

(Fortsetzung)

$b = 3$: Wir verwenden sukzessive Division um von $34_{(10)}$ auf $b = 3$ zu kommen:

$$\begin{array}{rcl} 34 \div 3 = 11 & 34 \bmod 3 = 1 & (\leftarrow \text{LSB}) \\ 11 \div 3 = 3 & 11 \bmod 3 = 2 & \end{array}$$

Damit ergibt sich:

Lösung 32: Zahlen konvertieren

(Fortsetzung)

$b = 3$: Wir verwenden sukzessive Division um von $34_{(10)}$ auf $b = 3$ zu kommen:

$$\begin{array}{rcl} 34 \div 3 = 11 & 34 \bmod 3 = 1 & (\leftarrow \text{LSB}) \\ 11 \div 3 = 3 & 11 \bmod 3 = 2 & \\ 3 \div 3 = 1 & 3 \bmod 3 = 0 & \end{array}$$

Damit ergibt sich:

Lösung 32: Zahlen konvertieren

(Fortsetzung)

$b = 3$: Wir verwenden sukzessive Division um von $34_{(10)}$ auf $b = 3$ zu kommen:

$34 \div 3 = 11$	$34 \bmod 3 = 1$ (\leftarrow <i>LSB</i>)
$11 \div 3 = 3$	$11 \bmod 3 = 2$
$3 \div 3 = 1$	$3 \bmod 3 = 0$
$1 \div 3 = 0$	$1 \bmod 3 = 1$ (\leftarrow <i>MSB</i>)

Damit ergibt sich:

Lösung 32: Zahlen konvertieren

(Fortsetzung)

$b = 3$: Wir verwenden sukzessive Division um von $34_{(10)}$ auf $b = 3$ zu kommen:

$34 \div 3 = 11$	$34 \bmod 3 = 1$ (\leftarrow <i>LSB</i>)
$11 \div 3 = 3$	$11 \bmod 3 = 2$
$3 \div 3 = 1$	$3 \bmod 3 = 0$
$1 \div 3 = 0$	$1 \bmod 3 = 1$ (\leftarrow <i>MSB</i>)

Damit ergibt sich:

Lösung 32: Zahlen konvertieren

(Fortsetzung)

$b = 3$: Wir verwenden sukzessive Division um von $34_{(10)}$ auf $b = 3$ zu kommen:

$$\begin{array}{rcl} 34 \div 3 = 11 & 34 \bmod 3 = 1 & (\leftarrow \text{LSB}) \\ 11 \div 3 = 3 & 11 \bmod 3 = 2 & \\ 3 \div 3 = 1 & 3 \bmod 3 = 0 & \\ 1 \div 3 = 0 & 1 \bmod 3 = 1 & (\leftarrow \text{MSB}) \end{array}$$

Damit ergibt sich: $42_{(8)} = 100010_{(2)} = 22_{(16)} = 34_{(10)} = 1021_{(3)}$

FUNKTION ZU REKURSION

Aufgabe 33: Schleife zu Rekursion

(4 Minuten)

Wandeln Sie folgenden Code in einen rekursiven Algorithmus um:

Aufgabe 33: Schleife zu Rekursion

(4 Minuten)

Wandeln Sie folgenden Code in einen rekursiven Algorithmus um:

```
public int ggT(int a, int b){  
    while(b != 0){  
  
    }  
  
}
```

Aufgabe 33: Schleife zu Rekursion

(4 Minuten)

Wandeln Sie folgenden Code in einen rekursiven Algorithmus um:

```
public int ggT(int a, int b){  
    while(b != 0){  
        final int tmp = b;  
        b = a % b;  
        a = tmp;  
    }  
  
}
```

Aufgabe 33: Schleife zu Rekursion

(4 Minuten)

Wandeln Sie folgenden Code in einen rekursiven Algorithmus um:

```
public int ggT(int a, int b){  
    while(b != 0){  
        final int tmp = b;  
        b = a % b;  
        a = tmp;  
    }  
    return a;  
}
```


FUNKTION ZU REKURSION – LÖSUNG

Lösung 33: Schleife zu Rekursion

Lösung 33: Schleife zu Rekursion

```
public int ggT(int a, int b){  
  
}
```

Lösung 33: Schleife zu Rekursion

```
public int ggT(int a, int b){  
    if(b == 0) return a;  
  
}
```

Lösung 33: Schleife zu Rekursion

```
public int ggT(int a, int b){  
    if(b == 0) return a;  
    else return ggT(b, a % b);  
}
```

REKURSIVER KONVERTER

Aufgabe 34: Rekursion zur Konvertierung

(4 Minuten)

Schreiben Sie eine rekursive Java-Methode `int2bin(int)`, die einen positiven Integer in dessen Binärdarstellung (als `String`) verwandelt. Beispiel:

Aufgabe 34: Rekursion zur Konvertierung

(4 Minuten)

Schreiben Sie eine rekursive Java-Methode `int2bin(int)`, die einen positiven Integer in dessen Binärdarstellung (als `String`) verwandelt. Beispiel:

```
int2bin(42) // → "101010"
```


Aufgabe 34: Rekursion zur Konvertierung

(4 Minuten)

Schreiben Sie eine rekursive Java-Methode `int2bin(int)`, die einen positiven Integer in dessen Binärdarstellung (als `String`) verwandelt.

Beispiel:

```
int2bin(42) // → "101010"
```

Hinweis: Um eine Zahl in einen String zu konvertieren, kann Konkatination oder `Integer.toString(int)` verwendet werden.

REKURSION ZUR KONVERTIERUNG – LÖSUNG

Lösung 34: Rekursion zur Konvertierung

Lösung 34: Rekursion zur Konvertierung

```
String int2bin(int num){  
  
}
```

Lösung 34: Rekursion zur Konvertierung

```
String int2bin(int num){  
    if(num < 2) return Integer.toString(num);  
  
}
```

Lösung 34: Rekursion zur Konvertierung

```
String int2bin(int num){  
    if(num < 2) return Integer.toString(num);  
    return int2bin(num / 2) + Integer.toString(num % 2);  
}
```

NUMERISCHES PALINDROM

Aufgabe 35: Numerisches Palindrom

(5 Minuten)

Schreiben Sie eine Java-Methode `boolean check(int n)`, die eine positive Zahl `n` erhält und entscheidet, ob diese ein Palindrom ist.

Aufgabe 35: Numerisches Palindrom

(5 Minuten)

Schreiben Sie eine Java-Methode `boolean check(int n)`, die eine positive Zahl `n` erhält und entscheidet, ob diese ein Palindrom ist. Eine positive Zahl ist ein Palindrom, wenn sie von vorne und hinten gelesen den gleichen Wert repräsentiert: 42324 ist eines, 192 aber nicht. Führende Nullen sind zu ignorieren (010 ist kein Palindrom). Verwenden Sie nur Rekursion und keine Iteration.

Sie dürfen Hilfsmethoden und die Funktionen der `Math`-Klasse verwenden, die Konvertierung der Zahl in einen String oder ein Array ist nicht gestattet.

Vereinfachung: Wenn Sie Probleme mit führenden Nullen in der Rekursion haben, können Sie auch nur Ziffern $z_j \in \{1, \dots, 9\}$ unterstützen.

NUMERISCHES PALINDROM – LÖSUNG

Lösung 35: Numerisches Palindrom

Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

```
boolean check(int n) {  
  
}
```

Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

```
boolean check(int n) {  
    return n == reverseNum(n, 0);  
}
```

Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

```
int reverseNum(int number, int current) {  
  
}  
  
boolean check(int n) {  
    return n == reverseNum(n, 0);  
}
```

Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

```
int reverseNum(int number, int current) {  
    if (number == 0) return current;  
  
}  
  
boolean check(int n) {  
    return n == reverseNum(n, 0);  
}
```


Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

```
int reverseNum(int number, int current) {  
    if (number == 0) return current;  
    return reverseNum(  
    );  
}  
  
boolean check(int n) {  
    return n == reverseNum(n, 0);  
}
```

Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

```
int reverseNum(int number, int current) {  
    if (number == 0) return current;  
    return reverseNum(number / 10,          );  
}
```

```
boolean check(int n) {  
    return n == reverseNum(n, 0);  
}
```

Wir schneiden die letzte Ziffer ab ($\text{number} / 10$), fügen diese „von vorne nach hinten“ an die neue Zahl an ($\text{current} * 10 + \text{number} \% 10$).

Lösung 35: Numerisches Palindrom

Wir können die Zahl rekursiv umdrehen und vergleichen (mit Null):

```
int reverseNum(int number, int current) {  
    if (number == 0) return current;  
    return reverseNum(number / 10, current * 10 + number % 10);  
}
```

```
boolean check(int n) {  
    return n == reverseNum(n, 0);  
}
```

Wir schneiden die letzte Ziffer ab ($\text{number} / 10$), fügen diese „von vorne nach hinten“ an die neue Zahl an ($\text{current} * 10 + \text{number} \% 10$).

Lösung 35: Numerisches Palindrom

(Fortsetzung)

Lösung 35: Numerisches Palindrom

(Fortsetzung)

Man kann auch analog zur `String::substring` Version vorgehen und mit der Stellenanzahl `Math.log10(double)` immer die erste und letzte Zahl abschneiden. Die Variablen sind hier zur Übersicht:

Lösung 35: Numerisches Palindrom

(Fortsetzung)

Man kann auch analog zur `String::substring` Version vorgehen und mit der Stellenanzahl `Math.log10(double)` immer die erste und letzte Zahl abschneiden. Die Variablen sind hier zur Übersicht:

```
static boolean check(int n) {  
    if(n <= 9) return true;  
    int length = (int) Math.log10(n) + 1; // Anzahl der Ziffern in n  
  
}
```


Lösung 35: Numerisches Palindrom

(Fortsetzung)

Man kann auch analog zur `String::substring` Version vorgehen und mit der Stellenanzahl `Math.log10(double)` immer die erste und letzte Zahl abschneiden. Die Variablen sind hier zur Übersicht:

```
static boolean check(int n) {  
    if(n <= 9) return true;  
    int length = (int) Math.log10(n) + 1; // Anzahl der Ziffern in n  
    int largest = (int) Math.pow(10, length - 1); // Höchste Wertigkeit  
  
}
```

Lösung 35: Numerisches Palindrom

(Fortsetzung)

Man kann auch analog zur `String::substring` Version vorgehen und mit der Stellenanzahl `Math.log10(double)` immer die erste und letzte Zahl abschneiden. Die Variablen sind hier zur Übersicht:

```
static boolean check(int n) {  
    if(n <= 9) return true;  
    int length = (int) Math.log10(n) + 1; // Anzahl der Ziffern in n  
    int largest = (int) Math.pow(10, length - 1); // Höchste Wertigkeit  
    int first = n / largest; // Größte Ziffer  
    int last = n % 10; // Kleinste Ziffer  
  
}
```

Lösung 35: Numerisches Palindrom

(Fortsetzung)

Man kann auch analog zur `String::substring` Version vorgehen und mit der Stellenanzahl `Math.log10(double)` immer die erste und letzte Zahl abschneiden. Die Variablen sind hier zur Übersicht:

```
static boolean check(int n) {  
    if(n <= 9) return true;  
    int length = (int) Math.log10(n) + 1; // Anzahl der Ziffern in n  
    int largest = (int) Math.pow(10, length - 1); // Höchste Wertigkeit  
    int first = n / largest; // Größte Ziffer  
    int last = n % 10; // Kleinste Ziffer  
    if(first != last)  
        return false;  
  
}
```

Lösung 35: Numerisches Palindrom

(Fortsetzung)

Man kann auch analog zur `String::substring` Version vorgehen und mit der Stellenanzahl `Math.log10(double)` immer die erste und letzte Zahl abschneiden. Die Variablen sind hier zur Übersicht:

```
static boolean check(int n) {
    if(n <= 9) return true;
    int length = (int) Math.log10(n) + 1; // Anzahl der Ziffern in n
    int largest = (int) Math.pow(10, length - 1); // Höchste Wertigkeit
    int first = n / largest; // Größte Ziffer
    int last = n % 10; // Kleinste Ziffer
    if(first != last)
        return false;
    return check((n % largest) / 10); // Schneide größte & kleinste ab
}
```

Lösung 35: Numerisches Palindrom

(Fortsetzung)

Man kann auch analog zur `String::substring` Version vorgehen und mit der Stellenanzahl `Math.log10(double)` immer die erste und letzte Zahl abschneiden. Die Variablen sind hier zur Übersicht:

```
static boolean check(int n) {
    if(n <= 9) return true;
    int length = (int) Math.log10(n) + 1; // Anzahl der Ziffern in n
    int largest = (int) Math.pow(10, length - 1); // Höchste Wertigkeit
    int first = n / largest; // Größte Ziffer
    int last = n % 10; // Kleinste Ziffer
    if(first != last)
        return false;
    return check((n % largest) / 10); // Schneide größte & kleinste ab
}
```

Das Problem: Nullen in `10055001` gehen hier verloren.

ÜBUNGSAUFGABE

Aufgabe 36: Rechenaufwand berechnen

(6 Minuten)

Wie viele Rechenschritte benötigt das folgende Verfahren einmal im *worst-* und im *best-case*? Geben Sie jeweils auch die \mathcal{O} -Notation an:

Aufgabe 36: Rechenaufwand berechnen

(6 Minuten)

Wie viele Rechenschritte benötigt das folgende Verfahren einmal im *worst*- und im *best-case*? Geben Sie jeweils auch die \mathcal{O} -Notation an:

```
int getDistance(String a, String b){
    if(a == null || b == null) return -1;
    if(a.length() != b.length()) return -1;
    int dist = 0x0;
    for(int i = 0b0; i < a.length(); i++)
        if(a.charAt(i) != b.charAt(i))
            dist++;
    return dist;
}
```


ÜBUNGSAUFGABE – LÖSUNG

Lösung 36: Rechenaufwand berechnen

Handeln wir zuerst den *best-case* ab.

Lösung 36: Rechenaufwand berechnen

Handeln wir zuerst den *best-case* ab. Hier ist der String `a` **null** und die erste **if**-Bedingung terminiert mit **-1**.

Lösung 36: Rechenaufwand berechnen

Handeln wir zuerst den *best-case* ab. Hier ist der String **a** **null** und die erste **if**-Bedingung terminiert mit **-1**. Wir haben also im *best-case* mit einem Aufwand von einem Vergleich und damit: $\mathcal{O}(1)$.

Lösung 36: Rechenaufwand berechnen

Handeln wir zuerst den *best-case* ab. Hier ist der String **a** **null** und die erste **if**-Bedingung terminiert mit **-1**. Wir haben also im *best-case* mit einem Aufwand von einem Vergleich und damit: $\mathcal{O}(1)$. (Auch wenn dieser *best-case* relativ sinnfrei ist.)

Lösung 36: Rechenaufwand berechnen

Handeln wir zuerst den *best-case* ab. Hier ist der String **a** **null** und die erste **if**-Bedingung terminiert mit **-1**. Wir haben also im *best-case* mit einem Aufwand von einem Vergleich und damit: $\mathcal{O}(1)$. (Auch wenn dieser *best-case* relativ sinnfrei ist. Was wäre denn der *best-case* bei einer „gültigen“ Eingabe

Lösung 36: Rechenaufwand berechnen

Handeln wir zuerst den *best-case* ab. Hier ist der String **a** **null** und die erste **if**-Bedingung terminiert mit **-1**. Wir haben also im *best-case* mit einem Aufwand von einem Vergleich und damit: $\mathcal{O}(1)$. (Auch wenn dieser *best-case* relativ sinnfrei ist. Was wäre denn der *best-case* bei einer „gültigen“ Eingabe, also zwei gleichlangen Strings, die nicht **null** sind?)

Lösung 36: Rechenaufwand berechnen

Handeln wir zuerst den *best-case* ab. Hier ist der String **a** **null** und die erste **if**-Bedingung terminiert mit **-1**. Wir haben also im *best-case* mit einem Aufwand von einem Vergleich und damit: $\mathcal{O}(1)$. (Auch wenn dieser *best-case* relativ sinnfrei ist. Was wäre denn der *best-case* bei einer „gültigen“ Eingabe, also zwei gleichlangen Strings, die nicht **null** sind?)
Kommen wir nun zum *worst-case*...

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

```
int getDistance(String a, String b){
    if(a == null || b == null) return -1;
    if(a.length() != b.length()) return -1;
    int dist = 0x0;
    for(int i = 0b0; i < a.length(); i++)
        if(a.charAt(i) != b.charAt(i))
            dist++;
    return dist;
}
```

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von a durch n notiert ($a.length() \hat{=} n$)

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von a durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von `a` durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2
2. Vergleiche: $3 + n + n + 1$

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von `a` durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2
2. Vergleiche: $3 + n + n + 1$
3. Inkrementierungen: $n + n$

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von `a` durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2
2. Vergleiche: $3 + n + n + 1$
3. Inkrementierungen: $n + n$
4. Dekrementierungen: 0

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von `a` durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2
2. Vergleiche: $3 + n + n + 1$
3. Inkrementierungen: $n + n$
4. Dekrementierungen: 0

Insgesamt ergibt sich damit: $(2) + (4 + 2n) + (2n) + (0) = 6 + 4n$.

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von `a` durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2
2. Vergleiche: $3 + n + n + 1$
3. Inkrementierungen: $n + n$
4. Dekrementierungen: 0

Insgesamt ergibt sich damit: $(2) + (4 + 2n) + (2n) + (0) = 6 + 4n$. Dieser Rechenaufwand liegt in $6 + 4n \in \mathcal{O}(n)$.

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von `a` durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2
2. Vergleiche: $3 + n + n + 1$
3. Inkrementierungen: $n + n$
4. Dekrementierungen: 0

Insgesamt ergibt sich damit: $(2) + (4 + 2n) + (2n) + (0) = 6 + 4n$. Dieser Rechenaufwand liegt in $6 + 4n \in \mathcal{O}(n)$.

Hinweis: Der Vergleich im Kopf einer **for**-Schleife wird einmal öfters ausgeführt,

Lösung 36: Rechenaufwand berechnen

(Fortsetzung)

Sei die Länge von `a` durch n notiert ($a.length() \hat{=} n$)

1. Zuweisungen: 2
2. Vergleiche: $3 + n + n + 1$
3. Inkrementierungen: $n + n$
4. Dekrementierungen: 0

Insgesamt ergibt sich damit: $(2) + (4 + 2n) + (2n) + (0) = 6 + 4n$. Dieser Rechenaufwand liegt in $6 + 4n \in \mathcal{O}(n)$.

Hinweis: Der Vergleich im Kopf einer **for**-Schleife wird einmal öfters ausgeführt, als die **for**-Schleife selbst.

ÜBUNGSAUFGABE

Aufgabe 37: Komplexitätsklassen ordnen

(4 Minuten)

Ordnen Sie die folgenden Komplexitätsklassen (ohne Beweis), von der am geringsten skalierenden zur am stärksten skalierenden.

Aufgabe 37: Komplexitätsklassen ordnen

(4 Minuten)

Ordnen Sie die folgenden Komplexitätsklassen (ohne Beweis), von der am geringsten skalierenden zur am stärksten skalierenden.

Aufgabe 37: Komplexitätsklassen ordnen

(4 Minuten)

Ordnen Sie die folgenden Komplexitätsklassen (ohne Beweis), von der am geringsten skalierenden zur am stärksten skalierenden.

1. $\mathcal{O}(n \cdot \log(n^2))$

Aufgabe 37: Komplexitätsklassen ordnen

(4 Minuten)

Ordnen Sie die folgenden Komplexitätsklassen (ohne Beweis), von der am geringsten skalierenden zur am stärksten skalierenden.

1. $\mathcal{O}(n \cdot \log(n^2))$
2. $\mathcal{O}(4^n) - \mathcal{O}(2^n)$

Aufgabe 37: Komplexitätsklassen ordnen

(4 Minuten)

Ordnen Sie die folgenden Komplexitätsklassen (ohne Beweis), von der am geringsten skalierenden zur am stärksten skalierenden.

1. $\mathcal{O}(n \cdot \log(n^2))$

3. $\mathcal{O}\left(\frac{n}{12}\right) + \mathcal{O}(n^3)$

2. $\mathcal{O}(4^n) - \mathcal{O}(2^n)$

Aufgabe 37: Komplexitätsklassen ordnen

(4 Minuten)

Ordnen Sie die folgenden Komplexitätsklassen (ohne Beweis), von der am geringsten skalierenden zur am stärksten skalierenden.

1. $\mathcal{O}(n \cdot \log(n^2))$

3. $\mathcal{O}\left(\frac{n}{12}\right) + \mathcal{O}(n^3)$

2. $\mathcal{O}(4^n) - \mathcal{O}(2^n)$

4. $\mathcal{O}\left(\frac{n!}{n}\right)$

Aufgabe 37: Komplexitätsklassen ordnen

(4 Minuten)

Ordnen Sie die folgenden Komplexitätsklassen (ohne Beweis), von der am geringsten skalierenden zur am stärksten skalierenden.

1. $\mathcal{O}(n \cdot \log(n^2))$

3. $\mathcal{O}\left(\frac{n}{12}\right) + \mathcal{O}(n^3)$

5. $\mathcal{O}(14n + 12)$

2. $\mathcal{O}(4^n) - \mathcal{O}(2^n)$

4. $\mathcal{O}\left(\frac{n!}{n}\right)$

ÜBUNGSAUFGABE – LÖSUNG

Lösung 37: Komplexitätsklassen ordnen

Es ergibt sich:

Lösung 37: Komplexitätsklassen ordnen

Es ergibt sich:

5. $\mathcal{O}(14n + 12) = \mathcal{O}(n)$

Lösung 37: Komplexitätsklassen ordnen

Es ergibt sich:

$$5. \mathcal{O}(14n + 12) = \mathcal{O}(n)$$

$$1. \mathcal{O}(n \cdot \log(n^2)) = \mathcal{O}(n \cdot 2 \cdot \log(n)) = \mathcal{O}(n \log n)$$

Lösung 37: Komplexitätsklassen ordnen

Es ergibt sich:

$$5. \mathcal{O}(14n + 12) = \mathcal{O}(n)$$

$$1. \mathcal{O}(n \cdot \log(n^2)) = \mathcal{O}(n \cdot 2 \cdot \log(n)) = \mathcal{O}(n \log n)$$

$$3. \mathcal{O}\left(\frac{n}{12}\right) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$$

Lösung 37: Komplexitätsklassen ordnen

Es ergibt sich:

$$5. \mathcal{O}(14n + 12) = \mathcal{O}(n)$$

$$1. \mathcal{O}(n \cdot \log(n^2)) = \mathcal{O}(n \cdot 2 \cdot \log(n)) = \mathcal{O}(n \log n)$$

$$3. \mathcal{O}\left(\frac{n}{12}\right) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$$

$$2. \mathcal{O}(4^n) - \mathcal{O}(2^n) = \mathcal{O}(4^n - 2^n) = \mathcal{O}(4^n)$$

Lösung 37: Komplexitätsklassen ordnen

Es ergibt sich:

$$5. \mathcal{O}(14n + 12) = \mathcal{O}(n)$$

$$1. \mathcal{O}(n \cdot \log(n^2)) = \mathcal{O}(n \cdot 2 \cdot \log(n)) = \mathcal{O}(n \log n)$$

$$3. \mathcal{O}\left(\frac{n}{12}\right) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$$

$$2. \mathcal{O}(4^n) - \mathcal{O}(2^n) = \mathcal{O}(4^n - 2^n) = \mathcal{O}(4^n)$$

$$4. \mathcal{O}\left(\frac{n!}{n}\right) = \mathcal{O}((n-1)!) = \mathcal{O}(n!)$$

Lösung 37: Komplexitätsklassen ordnen

Es ergibt sich:

$$5. \mathcal{O}(14n + 12) = \mathcal{O}(n)$$

$$1. \mathcal{O}(n \cdot \log(n^2)) = \mathcal{O}(n \cdot 2 \cdot \log(n)) = \mathcal{O}(n \log n)$$

$$3. \mathcal{O}\left(\frac{n}{12}\right) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$$

$$2. \mathcal{O}(4^n) - \mathcal{O}(2^n) = \mathcal{O}(4^n - 2^n) = \mathcal{O}(4^n)$$

$$4. \mathcal{O}\left(\frac{n!}{n}\right) = \mathcal{O}((n-1)!) = \mathcal{O}(n!)$$

Eine derartige Kategorisierung kann auch (noch) genauer präzisiert werden.

ÜBUNGSAUFGABE

Aufgabe 38: Von Pseudocode zu Java

(4 Minuten)

Aufgabe 38: Von Pseudocode zu Java

(4 Minuten)

Übersetzen Sie den Pseudocode in eine Java-Methode „`magic(int, int)`“. Behalten Sie die Vorgehensweise bei. \mathbb{N} darf durch einen Integer eingegrenzt, ungültige Eingaben müssen nicht abgefangen werden.

Aufgabe 38: Von Pseudocode zu Java

(4 Minuten)

Übersetzen Sie den Pseudocode in eine Java-Methode „`magic(int, int)`“. Behalten Sie die Vorgehensweise bei. \mathbb{N} darf durch einen Integer eingegrenzt, ungültige Eingaben müssen nicht abgefangen werden.

Beschreiben Sie auch kurz, was der Algorithmus berechnet:

Aufgabe 38: Von Pseudocode zu Java

(4 Minuten)

Übersetzen Sie den Pseudocode in eine Java-Methode „`magic(int, int)`“. Behalten Sie die Vorgehensweise bei. \mathbb{N} darf durch einen Integer eingegrenzt, ungültige Eingaben müssen nicht abgefangen werden.

Beschreiben Sie auch kurz, was der Algorithmus berechnet:

Eingabe: $a, b \in \mathbb{N}$ ($b \geq 0$)

1 **Wenn** b *ist* 0 **tue:**

2 **Gebe zurück:** 1;

3 $wert :=$ **Rekursion mit neu** a **ist alt** a **und neu** b **ist alt** $b - 1$;

4 **Gebe zurück:** $a * wert$;

ÜBUNGSAUFGABE – LÖSUNG

Lösung 38: Von Pseudocode zu Java

Lösung 38: Von Pseudocode zu Java

Der Pseudocode lässt sich glücklicherweise fast direkt übernehmen:

Lösung 38: Von Pseudocode zu Java

Der Pseudocode lässt sich glücklicherweise fast direkt übernehmen:

```
int magic(int a, int b) {  
  
    if(b == 0) return 1;  
  
}
```

Lösung 38: Von Pseudocode zu Java

Der Pseudocode lässt sich glücklicherweise fast direkt übernehmen:

```
int magic(int a, int b) {  
  
    if(b == 0) return 1;  
  
    int wert = magic(a, b - 1);  
  
}
```

Lösung 38: Von Pseudocode zu Java

Der Pseudocode lässt sich glücklicherweise fast direkt übernehmen:

```
int magic(int a, int b) {  
  
    if(b == 0) return 1;  
  
    int wert = magic(a, b - 1);  
    return a * wert;  
}
```

Lösung 38: Von Pseudocode zu Java

Der Pseudocode lässt sich glücklicherweise fast direkt übernehmen:

```
int magic(int a, int b) {  
  
    if(b == 0) return 1;  
  
    int wert = magic(a, b - 1);  
    return a * wert;  
}
```

Die Methode berechnet a^b der Eingabe — a wird b mal mit sich selbst multipliziert.

Lösung 38: Von Pseudocode zu Java

Der Pseudocode lässt sich glücklicherweise fast direkt übernehmen:

```
int magic(int a, int b) {  
    // Abfangen ungültiger Eingaben:  
    // if(b < 0) throw IllegalArgumentException();  
    if(b == 0) return 1;  
  
    int wert = magic(a, b - 1);  
    return a * wert;  
}
```

Die Methode berechnet a^b der Eingabe — a wird b mal mit sich selbst multipliziert.

ÜBUNGSAUFGABE

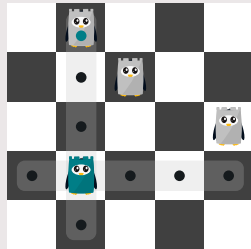
Aufgabe 39: Das n -Türme Problem

(7 Minuten)

Aufgabe 39: Das n -Türme Problem

(7 Minuten)

Ein Turm kann einen anderen auf einer vertikalen und horizontalen Linie schlagen. Im rechten 5×5 Feld, zum Beispiel der blaue Turm den am blauen Punkt.

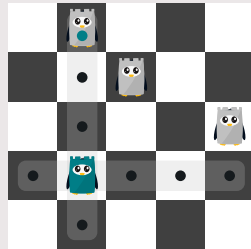


Aufgabe 39: Das n -Türme Problem

(7 Minuten)

Ein Turm kann einen anderen auf einer vertikalen und horizontalen Linie schlagen. Im rechten 5×5 Feld, zum Beispiel der blaue Turm den am blauen Punkt.

Schreiben Sie eine Methode, welche ein quadratisches Array erhält und genau dann `true` zurückliefert, wenn mindestens ein Turm einen anderen schlagen kann:
`boolean check (boolean[][] arr).`

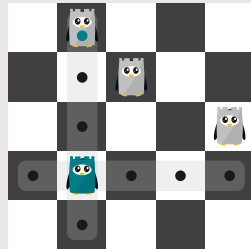


Aufgabe 39: Das n -Türme Problem

(7 Minuten)

Ein Turm kann einen anderen auf einer vertikalen und horizontalen Linie schlagen. Im rechten 5×5 Feld, zum Beispiel der blaue Turm den am blauen Punkt.

Schreiben Sie eine Methode, welche ein quadratisches Array erhält und genau dann `true` zurückliefert, wenn mindestens ein Turm einen anderen schlagen kann: `boolean check (boolean[][] arr)`. Ist `arr[i][j]` `true`, so markiert dies einen Turm in der i -ten Zeile und j -ten Spalte.



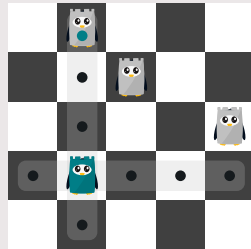
Aufgabe 39: Das n -Türme Problem

(7 Minuten)

Ein Turm kann einen anderen auf einer vertikalen und horizontalen Linie schlagen. Im rechten 5×5 Feld, zum Beispiel der blaue Turm den am blauen Punkt.

Schreiben Sie eine Methode, welche ein quadratisches Array erhält und genau dann `true` zurückliefert, wenn mindestens ein Turm einen anderen schlagen kann: `boolean check (boolean[][] arr)`. Ist `arr[i][j]` `true`, so markiert dies einen Turm in der i -ten Zeile und j -ten Spalte.

Es ist Ihnen für diese Aufgabe **nicht** gestattet, Schleifen zu verwenden. Verwenden Sie *ausschließlich* Fallunterscheidungen und Methodenaufrufe für den Kontrollfluss. Nehmen Sie an, dass `arr` wirklich quadratisch und mindestens ein Feld groß ist. Hilfsmethoden sind gestattet.



ÜBUNGSAUFGABE – LÖSUNG

Lösung 39: Das n -Türme Problem

Lösung 39: Das n -Türme Problem

```
boolean check(boolean[][] arr) {  
    return check(arr, 0, 0); // Start left upper corner  
}
```

Lösung 39: Das n -Türme Problem

```
boolean check(boolean[][] arr) {  
    return check(arr, 0, 0); // Start left upper corner  
}
```

```
boolean check(boolean[][] arr, int y, int x) {
```

```
}
```

Lösung 39: Das n -Türme Problem

```
boolean check(boolean[][] arr) {  
    return check(arr, 0, 0); // Start left upper corner  
}
```

```
boolean check(boolean[][] arr, int y, int x) {  
    if(y >= arr.length) return false; // done
```

```
}
```

Lösung 39: Das n -Türme Problem

```
boolean check(boolean[][] arr) {  
    return check(arr, 0, 0); // Start left upper corner  
}
```

```
boolean check(boolean[][] arr, int y, int x) {  
    if(y >= arr.length) return false; // done  
    // Next line  
    if(x >= arr.length) return check(arr, y + 1, 0);  
  
}
```

Lösung 39: Das n -Türme Problem

```
boolean check(boolean[][] arr) {  
    return check(arr, 0, 0); // Start left upper corner  
}  
  
boolean check(boolean[][] arr, int y, int x) {  
    if(y >= arr.length) return false; // done  
    // Next line  
    if(x >= arr.length) return check(arr, y + 1, 0);  
    if(arr[y][x] && canBeatOtherRook(arr, y, x)) return true;  
  
}
```

Lösung 39: Das n -Türme Problem

```
boolean check(boolean[][] arr) {  
    return check(arr, 0, 0); // Start left upper corner  
}  
  
boolean check(boolean[][] arr, int y, int x) {  
    if(y >= arr.length) return false; // done  
    // Next line  
    if(x >= arr.length) return check(arr, y + 1, 0);  
    if(arr[y][x] && canBeatOtherRook(arr, y, x)) return true;  
  
    return check(arr, y, x + 1);  
}
```


ÜBUNGSAUFGABE – LÖSUNG

Lösung 39: Das n -Türme Problem

(Fortsetzung)

Lösung 39: Das n -Türme Problem

(Fortsetzung)

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x) {  
  
}
```

Lösung 39: Das n -Türme Problem

(Fortsetzung)

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x) {  
    return canBeatOtherRook(arr, y, x, 0);  
}
```


Lösung 39: Das n -Türme Problem

(Fortsetzung)

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x) {  
    return canBeatOtherRook(arr, y, x, 0);  
}
```

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x, int i) {  
    if(i >= arr.length) // checked all vertical and horizontal  
        return false;  
  
}
```

Lösung 39: Das n -Türme Problem

(Fortsetzung)

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x) {  
    return canBeatOtherRook(arr, y, x, 0);  
}
```

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x, int i) {  
    if(i >= arr.length) // checked all vertical and horizontal  
        return false;  
    if(x != i && y != i) // different position  
  
}
```

Lösung 39: Das n -Türme Problem

(Fortsetzung)

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x) {  
    return canBeatOtherRook(arr, y, x, 0);  
}
```

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x, int i) {  
    if(i >= arr.length) // checked all vertical and horizontal  
        return false;  
    if(x != i && y != i) // different position  
        if(arr[y][i] || arr[i][x]) // rook in line or column  
            return true;  
    return canBeatOtherRook(arr, y, x, i + 1);  
}
```


Lösung 39: Das n -Türme Problem

(Fortsetzung)

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x) {  
    return canBeatOtherRook(arr, y, x, 0);  
}
```

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x, int i) {  
    if(i >= arr.length) // checked all vertical and horizontal  
        return false;  
    if(x != i && y != i) // different position  
        if(arr[y][i] || arr[i][x]) // rook in line or column  
            return true;  
}
```

Lösung 39: Das n -Türme Problem

(Fortsetzung)

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x) {  
    return canBeatOtherRook(arr, y, x, 0);  
}
```

```
boolean canBeatOtherRook(boolean[][] arr, int y, int x, int i) {  
    if(i >= arr.length) // checked all vertical and horizontal  
        return false;  
    if(x != i && y != i) // different position  
        if(arr[y][i] || arr[i][x]) // rook in line or column  
            return true;  
    return canBeatOtherRook(arr, y, x, i + 1);  
}
```

Weiterführende Konzepte



SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen,

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.
- Es gibt zwei Varianten:

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.
- Es gibt zwei Varianten:
 - EINFACH: Durchlaufen Suchraum auf Basis einer Datenstruktur.

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.
- Es gibt zwei Varianten:
 - EINFACH: Durchlaufen Suchraum auf Basis einer Datenstruktur.
 - HEURISTISCH: Besitzen zusätzliches Wissen (Verteilung, ...) über die Daten, die sie zur Beschleunigung der Suche verwenden.

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.
- Es gibt zwei Varianten:
 - EINFACH: Durchlaufen Suchraum auf Basis einer Datenstruktur.
 - HEURISTISCH: Besitzen zusätzliches Wissen (Verteilung, ...) über die Daten, die sie zur Beschleunigung der Suche verwenden.
- Die zu suchenden *Eigenschaften*, nennt man auch *Merkmale*.

KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

- Uns sind zwei Verfahren bekannt, Elemente zu suchen:

KLASSISCHE SUCHVERFAHREN

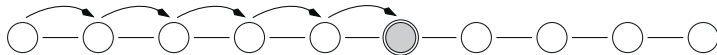
- Uns sind zwei Verfahren bekannt, Elemente zu suchen:
 - LINEAR: Die lineare (naive/sequentielle) Suche beginnt am Anfang der Folge und prüft Element für Element ob es sich um das gesuchte handelt.

KLASSISCHE SUCHVERFAHREN

- Uns sind zwei Verfahren bekannt, Elemente zu suchen:
 - LINEAR: Die lineare (naive/sequentielle) Suche beginnt am Anfang der Folge und prüft Element für Element ob es sich um das gesuchte handelt. Die Komplexität beträgt $\mathcal{O}(n)$.

KLASSISCHE SUCHVERFAHREN

- Uns sind zwei Verfahren bekannt, Elemente zu suchen:
LINEAR: Die lineare (naive/sequentielle) Suche beginnt am Anfang der Folge und prüft Element für Element ob es sich um das gesuchte handelt. Die Komplexität beträgt $\mathcal{O}(n)$.



LINEARE SUCHE

Aufgabe 40: Lineare Suche

(4 Minuten)

Schreiben Sie eine Methode `int linearSearch(int[] arr, int key)`, die ein Array mit der linearen Suche nach einem gegebenen Schlüssel durchsucht und den (ersten) Index des Schlüssels zurückliefert. Wird der Schlüssel nicht gefunden, so soll die Methode einen negativen Wert zurückliefern.

LINEARE SUCHE – LÖSUNG

Lösung 40: Lineare Suche

Lösung 40: Lineare Suche

```
int linearSearch(int[] arr, int key){  
    for(int i = 0; i < arr.length; i++)  
        if(arr[i] == key)  
            return i;  
  
}
```

Lösung 40: Lineare Suche

```
int linearSearch(int[] arr, int key){  
    for(int i = 0; i < arr.length; i++)  
        if(arr[i] == key)  
            return i;  
    return -1; // Magic Number für "nicht gefunden"  
}
```

KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.

KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.
Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff.

KLASSISCHE SUCHVERFAHREN

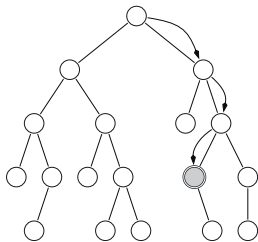
BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.
Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht.

KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig. Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht. Die Komplexität beträgt $\mathcal{O}(\log n)$ (ohne Sortieren), da jeder Vergleich den restlichen Suchbereich halbiert.

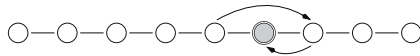
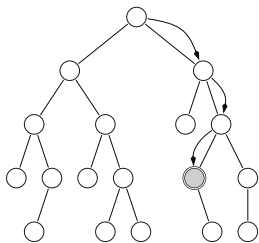
KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig. Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht. Die Komplexität beträgt $\mathcal{O}(\log n)$ (ohne Sortieren), da jeder Vergleich den restlichen Suchbereich halbiert.



KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.
Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht. Die Komplexität beträgt $\mathcal{O}(\log n)$ (ohne Sortieren), da jeder Vergleich den restlichen Suchbereich halbiert.



BINÄRE SUCHE

Aufgabe 41: Binäre Suche

(6 Minuten)

Schreiben Sie eine Methode `int binarySearch(int[] arr, int key)`, die ein bereits aufsteigend sortiertes Array mit der binären Suche nach einem gegebenen Schlüssel durchsucht und den Index des Schlüssels zurückliefert. Wird der Schlüssel nicht gefunden, so soll die Methode einen beliebigen negativen Integer zurückliefern.

BINÄRE SUCHE – LÖSUNG

Lösung 41: Binäre Suche

Lösung 41: Binäre Suche

```
int binarySearch(int[] arr, int key){
    int min = 0; // Suche von min
    int max = arr.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.

    }

}
```

Lösung 41: Binäre Suche

```
int binarySearch(int[] arr, int key){
    int min = 0; // Suche von min
    int max = arr.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte

    }
}
```

Lösung 41: Binäre Suche

```
int binarySearch(int[] arr, int key){
    int min = 0; // Suche von min
    int max = arr.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(arr[middle] == key) // gefunden
            return middle;

    }
}
```


Lösung 41: Binäre Suche

```
int binarySearch(int[] arr, int key){
    int min = 0; // Suche von min
    int max = arr.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(arr[middle] == key) // gefunden
            return middle;
        else if(key < arr[middle]) // Suche links [min, middle-1]
            max = middle - 1;

    }
}
```

Lösung 41: Binäre Suche

```
int binarySearch(int[] arr, int key){
    int min = 0; // Suche von min
    int max = arr.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(arr[middle] == key) // gefunden
            return middle;
        else if(key < arr[middle]) // Suche links [min, middle-1]
            max = middle - 1;
        else // Suche rechts [middle+1, max]
            min = middle + 1;
    }
}
```

Lösung 41: Binäre Suche

```
int binarySearch(int[] arr, int key){
    int min = 0; // Suche von min
    int max = arr.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(arr[middle] == key) // gefunden
            return middle;
        else if(key < arr[middle]) // Suche links [min, middle-1]
            max = middle - 1;
        else // Suche rechts [middle+1, max]
            min = middle + 1;
    }
    return -1; // Magic Number für "nicht gefunden"
}
```

SORTIERVERFAHREN FORMAL

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:
INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher,

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:
 - INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, ...)

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:
 - INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, ...)
 - EXTERN: Der Datenbestand ist (überwiegend) ausgelagert.

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:
 - INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, ...)
 - EXTERN: Der Datenbestand ist (überwiegend) ausgelagert. Für das Sortieren werden meist nur die obersten Element eines Stapels betrachtet (Dateien, ...).

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:
 - INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, ...)
 - EXTERN: Der Datenbestand ist (überwiegend) ausgelagert. Für das Sortieren werden meist nur die obersten Element eines Stapels betrachtet (Dateien, ...). Oft erlauben Hilfsmethoden die selben Verfahren wie bei der internen Variante.

SORTIERVERFAHREN FORMAL, II

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

SUKZESSIV: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:
 - SUKZESSIV: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

SUKZESSIV: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

DIVIDE-AND-CONQUER: Daten werden aufgeteilt,

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

SUKZESSIV: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

DIVIDE-AND-CONQUER: Daten werden aufgeteilt, in Teilen sortiert und dann aus sortierten Teilmengen zusammengefügt.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

SUKZESSIV: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

DIVIDE-AND-CONQUER: Daten werden aufgeteilt, in Teilen sortiert und dann aus sortierten Teilmengen zusammengefügt. In der Regel rekursiv implementiert.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

SUKZESSIV: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

DIVIDE-AND-CONQUER: Daten werden aufgeteilt, in Teilen sortiert und dann aus sortierten Teilmengen zusammengefügt. In der Regel rekursiv implementiert.

HALDE: Sortieren mit dafür geeigneten Datenstrukturen wie dem *Heap*.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:
 - SUKZESSIV: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.
 - DIVIDE-AND-CONQUER: Daten werden aufgeteilt, in Teilen sortiert und dann aus sortierten Teilmengen zusammengefügt. In der Regel rekursiv implementiert.
 - HALDE: Sortieren mit dafür geeigneten Datenstrukturen wie dem *Heap*. In der Regel rekursiv implementiert.

SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort

(iterativ)

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil,

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).

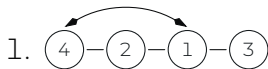
SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).



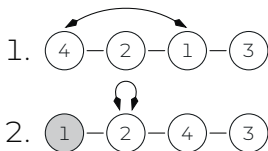
SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).



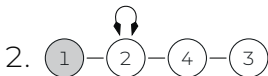
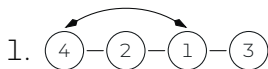
SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).



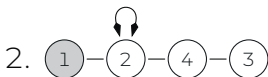
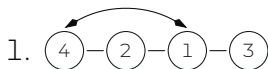
SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort

(iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).



SELECTIONSORT IMPLEMENTIEREN

Aufgabe 42: Selectionsort implementieren

(6 Minuten)

Gegeben sei eine Methode `int getMin(int[] arr, int i, int j)`, die korrekt das Minimum im Teilarray $[i,j]$ von `arr` findet. Weiter gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

SELECTIONSORT IMPLEMENTIEREN

Aufgabe 42: Selectionsort implementieren

(6 Minuten)

Gegeben sei eine Methode `int getMin(int[] arr, int i, int j)`, die korrekt das Minimum im Teilarray $[i,j]$ von `arr` findet. Weiter gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Schreiben Sie eine Methode `int[] selectionSort(int[] arr)`, die das übergebene Array mittels Selectionsort als *neues* aufsteigend sortiertes Array zurückliefert. Das übergebene Array darf nicht verändert werden.

SELECTIONSORT IMPLEMENTIEREN – LÖSUNG

Lösung 42: Selectionsort implementieren

Lösung 42: Selectionsort implementieren

```
public static int[] selectionSort(int[] arr) {  
    // Erzeuge Kopie  
    int[] newArr = new int[arr.length];  
    for(int i = 0; i < arr.length; i++)  
        newArr[i] = arr[i];  
  
}
```


Lösung 42: Selectionsort implementieren

```
public static int[] selectionSort(int[] arr) {  
    // Erzeuge Kopie  
    int[] newArr = new int[arr.length];  
    for(int i = 0; i < arr.length; i++)  
        newArr[i] = arr[i];  
  
    // Sortiere nach Selectionsort  
  
}
```

Lösung 42: Selectionsort implementieren

```
public static int[] selectionSort(int[] arr) {  
    // Erzeuge Kopie  
    int[] newArr = new int[arr.length];  
    for(int i = 0; i < arr.length; i++)  
        newArr[i] = arr[i];  
  
    // Sortiere nach Selectionsort  
    for(int i = 0; i < newArr.length - 1; i++) {  
  
    }  
  
}
```

Lösung 42: Selectionsort implementieren

```
public static int[] selectionSort(int[] arr) {  
    // Erzeuge Kopie  
    int[] newArr = new int[arr.length];  
    for(int i = 0; i < arr.length; i++)  
        newArr[i] = arr[i];  
  
    // Sortiere nach Selectionsort  
    for(int i = 0; i < newArr.length - 1; i++) {  
        int min = getMin(newArr, i, newArr.length);  
  
    }  
  
}
```

Lösung 42: Selectionsort implementieren

```
public static int[] selectionSort(int[] arr) {  
    // Erzeuge Kopie  
    int[] newArr = new int[arr.length];  
    for(int i = 0; i < arr.length; i++)  
        newArr[i] = arr[i];  
  
    // Sortiere nach Selectionsort  
    for(int i = 0; i < newArr.length - 1; i++) {  
        int min = getMin(newArr, i, newArr.length);  
        swap(newArr, i, min);  
    }  
  
}
```

Lösung 42: Selectionsort implementieren

```
public static int[] selectionSort(int[] arr) {  
    // Erzeuge Kopie  
    int[] newArr = new int[arr.length];  
    for(int i = 0; i < arr.length; i++)  
        newArr[i] = arr[i];  
  
    // Sortiere nach Selectionsort  
    for(int i = 0; i < newArr.length - 1; i++) {  
        int min = getMin(newArr, i, newArr.length);  
        swap(newArr, i, min);  
    }  
  
    return arr;  
}
```

SORTIERVERFAHREN: SELECTIONSORT

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length - 1; i++) {  
        int min = i; // Suche Minimum  
  
    }  
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length - 1; i++) {  
        int min = i; // Suche Minimum  
        for (int j = i + 1; j < arr.length; j++)  
  
        }  
    }  
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length - 1; i++) {  
        int min = i; // Suche Minimum  
        for (int j = i + 1; j < arr.length; j++)  
            if(arr[j] < arr[min])  
                min = j;  
    }  
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length - 1; i++) {  
        int min = i; // Suche Minimum  
        for (int j = i + 1; j < arr.length; j++)  
            if(arr[j] < arr[min])  
                min = j;  
        // Tausche 'i' und 'min'  
  
    }  
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length - 1; i++) {  
        int min = i; // Suche Minimum  
        for (int j = i + 1; j < arr.length; j++)  
            if(arr[j] < arr[min])  
                min = j;  
        // Tausche 'i' und 'min'  
        int tmp = arr[i];  
  
    }  
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length - 1; i++) {  
        int min = i; // Suche Minimum  
        for (int j = i + 1; j < arr.length; j++)  
            if(arr[j] < arr[min])  
                min = j;  
        // Tausche 'i' und 'min'  
        int tmp = arr[i];  
        arr[i] = arr[min];  
    }  
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {  
    for(int i = 0; i < arr.length - 1; i++) {  
        int min = i; // Suche Minimum  
        for (int j = i + 1; j < arr.length; j++)  
            if(arr[j] < arr[min])  
                min = j;  
        // Tausche 'i' und 'min'  
        int tmp = arr[i];  
        arr[i] = arr[min];  
        arr[min] = tmp;  
    }  
}
```

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort

(iterativ)

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil,

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.

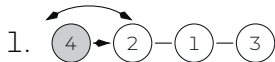
SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.



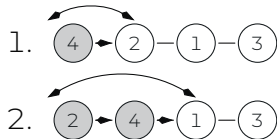
SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.



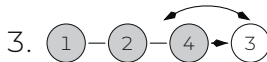
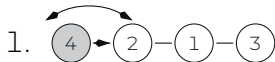
SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.



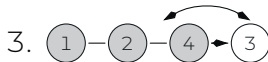
SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort

(iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.



INSERTIONSORT IMPLEMENTIEREN

Aufgabe 43: Insertionsort implementieren

(6 Minuten)

Gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Aufgabe 43: Insertionsort implementieren

(6 Minuten)

Gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Schreiben Sie eine Methode `void insertionSort(int[])`, die das übergebene Array mittels Insertionsort aufsteigend sortiert. Dabei darf das Array *nur* durch die übergebenen `swap`-Methode verändert werden.

INSERTIONSORT IMPLEMENTIEREN – LÖSUNG

Lösung 43: Insertionsort implementieren

```
public static void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
        int pos = i;  
  
    }  
}
```

Lösung 43: Insertionsort implementieren

```
public static void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
        int pos = i;  
        while(pos > 0 && arr[pos] <= arr[pos - 1]) {  
  
        }  
    }  
}
```

Lösung 43: Insertionsort implementieren

```
public static void insertionSort(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int pos = i;
        while(pos > 0 && arr[pos] <= arr[pos - 1]) {
            swap(arr, pos, pos - 1);
            pos -= 1;
        }
    }
}
```

SORTIERVERFAHREN: INSERTIONSORT

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
  
  
  
  
  
  
  
  
  
    }  
}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
        int pos = i - 1, elem = arr[i];  
  
        }  
}
```


SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
        int pos = i - 1, elem = arr[i];  
        while(pos >= 0 && arr[pos] > elem) { // Verschiebe  
  
            }  
  
    }  
}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
        int pos = i - 1, elem = arr[i];  
        while(pos >= 0 && arr[pos] > elem) { // Verschiebe  
            arr[pos + 1] = arr[pos];  
            pos--;  
        }  
    }  
}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {  
    for(int i = 1; i < arr.length; i++) {  
        int pos = i - 1, elem = arr[i];  
        while(pos >= 0 && arr[pos] > elem) { // Verschiebe  
            arr[pos + 1] = arr[pos];  
            pos--;  
        }  
        arr[pos+1] = elem;  
    }  
}
```

SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort

(iterativ)

Definition 11: Bubblesort

(iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

Definition 11: Bubblesort

(iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

Definition 11: Bubblesort

(iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Mit jedem Durchgang über das Array wird (mindestens) ein Element an die richtige Position getauscht.

Definition 11: Bubblesort

(iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Mit jedem Durchgang über das Array wird (mindestens) ein Element an die richtige Position getauscht.
- (Geringfügige) Verbesserung: Shaker-Sort, schiebt abwechselnd das maximale und minimale Element an die (Ziel-)Position.

SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort

(iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Mit jedem Durchgang über das Array wird (mindestens) ein Element an die richtige Position getauscht.
- (Geringfügige) Verbesserung: Shaker-Sort, schiebt abwechselnd das maximale und minimale Element an die (Ziel-)Position.



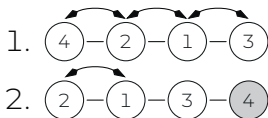
SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort

(iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Mit jedem Durchgang über das Array wird (mindestens) ein Element an die richtige Position getauscht.
- (Geringfügige) Verbesserung: Shaker-Sort, schiebt abwechselnd das maximale und minimale Element an die (Ziel-)Position.



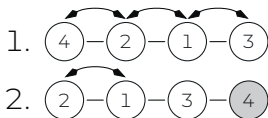
SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort

(iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Mit jedem Durchgang über das Array wird (mindestens) ein Element an die richtige Position getauscht.
- (Geringfügige) Verbesserung: Shaker-Sort, schiebt abwechselnd das maximale und minimale Element an die (Ziel-)Position.



SORTIERVERFAHREN: BUBBLESORT

```
public static void bubbleSort(int[] arr) {
    for(int i = arr.length - 1; i >= 1; i--) {
        for(int j = 0; j < i; j++) {
            if(arr[j] > arr[j+1]) { // Vertausche
                int tmp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = tmp;
            }
        }
    }
}
```

SHAKERSORT IMPLEMENTIEREN

SHAKERSORT IMPLEMENTIEREN

Aufgabe 44: Shakersort implementieren

(6 Minuten)

Gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Aufgabe 44: Shakersort implementieren

(6 Minuten)

Gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Schreiben Sie eine Methode `void shakerSort(int[] arr)`, die das übergebene Array mittels Shakersort aufsteigend sortiert. Dabei darf das Array *nur* durch die übergebenen `swap`-Methode verändert werden.

Ausgehend von Bubblesort, geht Shakersort *abwechselnd* von links nach rechts und von rechts nach links durch das noch zu sortierende Array. In beiden Richtungen werden Vertauschungen nach Bubblesort durchgeführt. Jede Aufwärts- und jede Abwärtsbewegung verringert dabei den noch zu sortierenden Bereich um 1.

SHAKERSORT IMPLEMENTIEREN – LÖSUNG

Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit l und r ein:

Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit `l` und `r` ein:

```
public void shakerSort(int[] arr) {
```

```
}
```


Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit l und r ein:

```
public void shakerSort(int[] arr) {  
    int l = 0, r = arr.length - 1;  
    while(l < r) {  
        for(int i = l; i < r; i++) { // aufwärts  
  
        }  
  
        }  
}
```

Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit `l` und `r` ein:

```
public void shakerSort(int[] arr) {  
    int l = 0, r = arr.length - 1;  
    while(l < r) {  
        for(int i = l; i < r; i++) { // aufwärts  
            if(arr[i] > arr[i + 1]) swap(arr, i, i + 1);  
        }  
  
        }  
}
```

Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit `l` und `r` ein:

```
public void shakerSort(int[] arr) {  
    int l = 0, r = arr.length - 1;  
    while(l < r) {  
        for(int i = l; i < r; i++) { // aufwärts  
            if(arr[i] > arr[i + 1]) swap(arr, i, i + 1);  
        }  
        r -= 1;  
    }  
}
```


Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit `l` und `r` ein:

```
public void shakerSort(int[] arr) {
    int l = 0, r = arr.length - 1;
    while(l < r) {
        for(int i = l; i < r; i++) { // aufwärts
            if(arr[i] > arr[i + 1]) swap(arr, i, i + 1);
        }
        r -= 1;
        for(int i = r - 1; i >= l; i--) { // abwärts

        }

    }
}
```

Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit l und r ein:

```
public void shakerSort(int[] arr) {
    int l = 0, r = arr.length - 1;
    while(l < r) {
        for(int i = l; i < r; i++) { // aufwärts
            if(arr[i] > arr[i + 1]) swap(arr, i, i + 1);
        }
        r -= 1;
        for(int i = r - 1; i >= l; i--) { // abwärts
            if(arr[i] > arr[i + 1]) swap(arr, i, i + 1);
        }
    }
}
```

Lösung 44: Shakersort implementieren

Den noch zu sortierenden Bereich grenzen wir mit l und r ein:

```
public void shakerSort(int[] arr) {
    int l = 0, r = arr.length - 1;
    while(l < r) {
        for(int i = l; i < r; i++) { // aufwärts
            if(arr[i] > arr[i + 1]) swap(arr, i, i + 1);
        }
        r -= 1;
        for(int i = r - 1; i >= l; i--) { // abwärts
            if(arr[i] > arr[i + 1]) swap(arr, i, i + 1);
        }
        l += 1;
    }
}
```

SHAKERSORT VISUALISIEREN

Aufgabe 45: Shakersort visualisieren

(5 Minuten)

Aufgabe 45: Shakersort visualisieren

(5 Minuten)

Gegeben sei das folgende Tupel:

$(13, -15, 9, 8, 23, 8, 0)$

Aufgabe 45: Shakersort visualisieren

(5 Minuten)

Gegeben sei das folgende Tupel:

$$(13, -15, 9, 8, 23, 8, 0)$$

Visualisieren Sie die einzelnen Durchläufe des Shaker-Sort, indem Sie für jeden auf- und für jeden absteigenden Durchlauf nacheinander alle getätigten Vertauschungen und den (aus Sicht des Algorithmus) noch unsortierten Teil des Tupels angeben.

Eine beispielhafte erste Ausgabe für das Tupel $(5, 2, 3, 7, 8, 9, 3)$:

SHAKERSORT VISUALISIEREN

Aufgabe 45: Shakersort visualisieren

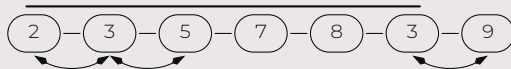
(5 Minuten)

Gegeben sei das folgende Tupel:

$(13, -15, 9, 8, 23, 8, 0)$

Visualisieren Sie die einzelnen Durchläufe des Shaker-Sort, indem Sie für jeden auf- und für jeden absteigenden Durchlauf nacheinander alle getätigten Vertauschungen und den (aus Sicht des Algorithmus) noch unsortierten Teil des Tupels angeben.

Eine beispielhafte erste Ausgabe für das Tupel $(5, 2, 3, 7, 8, 9, 3)$:



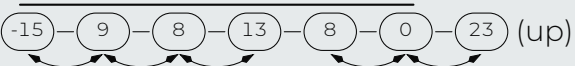
SHAKERSORT VISUALISIEREN – LÖSUNG

Lösung 45: Shakersort visualisieren

0.  13 — -15 — 9 — 8 — 23 — 8 — 0

Lösung 45: Shakersort visualisieren

0.  13 -15 9 8 23 8 0

1.  -15 9 8 13 8 0 23 (up)

SHAKERSORT VISUALISIEREN – LÖSUNG

Lösung 45: Shakersort visualisieren

0. (13) — (-15) — (9) — (8) — (23) — (8) — (0)

1. (13) — (-15) — (9) — (8) — (23) — (8) — (0) (up)

2. (-15) — (0) — (9) — (8) — (13) — (8) — (23) (down)

SHAKERSORT VISUALISIEREN – LÖSUNG

Lösung 45: Shakersort visualisieren

0. (13) — (-15) — (9) — (8) — (23) — (8) — (0)

1. (13) — (-15) — (9) — (8) — (23) — (8) — (0) (up)

2. (-15) — (0) — (9) — (8) — (13) — (8) — (23) (down)

3. (-15) — (0) — (8) — (9) — (8) — (13) — (23) (up)

SHAKERSORT VISUALISIEREN – LÖSUNG

Lösung 45: Shakersort visualisieren

0. (13) — (-15) — (9) — (8) — (23) — (8) — (0)

1. (13) — (-15) — (9) — (8) — (23) — (8) — (0) (up)

2. (-15) — (0) — (9) — (8) — (13) — (8) — (23) (down)

3. (-15) — (0) — (8) — (9) — (8) — (13) — (23) (up)

4. (-15) — (0) — (8) — (8) — (9) — (13) — (23) (down)

SHAKERSORT VISUALISIEREN – LÖSUNG

Lösung 45: Shakersort visualisieren


(Fortsetzung)

4.  (down)

Lösung 45: Shakersort visualisieren

(Fortsetzung)

4.  (down)

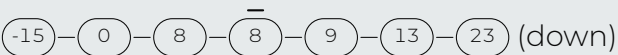
5.  (up)

Lösung 45: Shakersort visualisieren

(Fortsetzung)

4.  (down)

5.  (up)


6.  (down)

Lösung 45: Shakersort visualisieren

(Fortsetzung)

4.  (down)

5.  (up)

6.  (down)

Dieser Schritt kann theoretisch weggelassen werden, wenn man die optimierte Version von Shakersort benutzt. Diese bricht ab, sobald in einem Durchlauf nicht mehr getauscht wird.

SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort

(rekursiv)

SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort

(rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen.

Definition 12: Mergesort

(rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen.
Zusammenfügen sortierter Teilfolgen.

Definition 12: Mergesort

(rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen.
Zusammenfügen sortierter Teilfolgen.

- Verfolgt *easy split, hard join*. Das Aufteilen ist leicht (trivial), das Zusammenfügen aufwändig, da hier die Sortierung erfolgen muss.

Definition 12: Mergesort

(rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen.
Zusammenfügen sortierter Teilfolgen.

- Verfolgt *easy split, hard join*. Das Aufteilen ist leicht (trivial), das Zusammenfügen aufwändig, da hier die Sortierung erfolgen muss.
- Teilt sich auf in: *Split* (aufteilen) und *Merge* (verschmelzen).

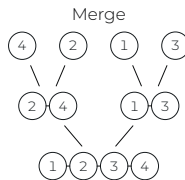
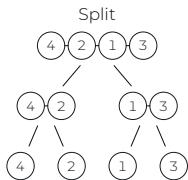
SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort

(rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen.
Zusammenfügen sortierter Teilfolgen.

- Verfolgt *easy split, hard join*. Das Aufteilen ist leicht (trivial), das Zusammenfügen aufwändig, da hier die Sortierung erfolgen muss.
- Teilt sich auf in: *Split* (aufteilen) und *Merge* (verschmelzen).



MERGESORT PSEUDOCODE

Aufgabe 46: Mergesort Pseudocode

(7 Minuten)

Aufgabe 46: Mergesort Pseudocode

(7 Minuten)

Formulieren Sie Pseudocode, welcher eine gegebene Liste l mit Elementen $l_i \in \mathbb{R}$ und Länge n durch Mergesort aufsteigend sortiert.

Sie dürfen weitere Routinen formulieren. Es sei eine korrekt implementierte Routine $\text{cut}(1, i, j)$ mit folgendem Vertrag gegeben:

Aufgabe 46: Mergesort Pseudocode

(7 Minuten)

Formulieren Sie Pseudocode, welcher eine gegebene Liste l mit Elementen $l_i \in \mathbb{R}$ und Länge n durch Mergesort aufsteigend sortiert.

Sie dürfen weitere Routinen formulieren. Es sei eine korrekt implementierte Routine $\text{cut}(l, i, j)$ mit folgendem Vertrag gegeben:

Input: Liste $l = (l_1, \dots, l_n)$, $i, j \in \{1, \dots, n\}$

Output: Teilliste (l_i, \dots, l_j) , falls $i \leq j$. Sonst (wenn $i > j$) eine leere Liste.

Aufgabe 46: Mergesort Pseudocode

(7 Minuten)

Formulieren Sie Pseudocode, welcher eine gegebene Liste l mit Elementen $l_i \in \mathbb{R}$ und Länge n durch Mergesort aufsteigend sortiert.

Sie dürfen weitere Routinen formulieren. Es sei eine korrekt implementierte Routine $\text{cut}(l, i, j)$ mit folgendem Vertrag gegeben:

Input: Liste $l = (l_1, \dots, l_n)$, $i, j \in \{1, \dots, n\}$

Output: Teilliste (l_i, \dots, l_j) , falls $i \leq j$. Sonst (wenn $i > j$) eine leere Liste.

Weiter dürfen Sie mit x_i direkt auf das Listenelement an Stelle i in x zugreifen (sofern $1 \leq i \leq n_x$ mit Länge n_x). Der Ausdruck „ $x + 1_i$ “ erzeugt eine neue Liste (x_1, \dots, x_n, l_i) . Ebenso „ $x + 1$ “ mit $(x_1, \dots, x_n, l_1, \dots, l_n)$.

MERGESORT PSEUDOCODE – LÖSUNG

Lösung 46: Mergesort Pseudocode

Lösung 46: Mergesort Pseudocode

Algorithm 6: mergesort(t)

Lösung 46: Mergesort Pseudocode

Algorithm 7: mergesort(t)

Input: Liste $l = (l_1, \dots, l_n)$

Output: Aufsteigend sortierte Liste l .

Lösung 46: Mergesort Pseudocode

Algorithm 8: mergesort(t)

Input: Liste $l = (l_1, \dots, l_n)$

Output: Aufsteigend sortierte Liste l .

// Ein- oder Nullelementig, bereits sortiert:

- 2 $left := \text{cut}(l, 1, \lfloor n/2 \rfloor);$
- 3 $right := \text{cut}(l, \lfloor n/2 \rfloor, n);$

Lösung 46: Mergesort Pseudocode

Algorithm 9: mergesort(t)

Input: Liste $l = (l_1, \dots, l_n)$

Output: Aufsteigend sortierte Liste l .

// Ein- oder Nullelementig, bereits sortiert:

- 1 **if** $n \leq 1$ **then return** l ;
- 2 $left := cut(l, 1, \lfloor n/2 \rfloor)$;
- 3 $right := cut(l, \lfloor n/2 \rfloor, n)$;

Lösung 46: Mergesort Pseudocode

Algorithm 10: mergesort(t)

Input: Liste $l = (l_1, \dots, l_n)$

Output: Aufsteigend sortierte Liste l .

// Ein- oder Nullelementig, bereits sortiert:

- 1 **if** $n \leq 1$ **then return** l ;
 - 2 $left := \text{cut}(l, 1, \lfloor n/2 \rfloor)$;
 - 3 $right := \text{cut}(l, \lfloor n/2 \rfloor, n)$;
- // Rekursion
- 4 $left := \text{mergesort}(left)$;
 - 5 $right := \text{mergesort}(right)$;

Lösung 46: Mergesort Pseudocode

Algorithm 11: mergesort(t)

Input: Liste $l = (l_1, \dots, l_n)$

Output: Aufsteigend sortierte Liste l .

// Ein- oder Nullelementig, bereits sortiert:

1 **if** $n \leq 1$ **then return** l ;

2 $left := cut(l, 1, \lfloor n/2 \rfloor)$;

3 $right := cut(l, \lfloor n/2 \rfloor, n)$;

// Rekursion

4 $left := mergesort(left)$;

5 $right := mergesort(right)$;

6 **return** $merge(left, right)$; // Auf der nächsten Folie

MERGESORT PSEUDOCODE – LÖSUNG

Lösung 46: Mergesort Pseudocode

(Fortsetzung)

Lösung 46: Mergesort Pseudocode

(Fortsetzung)

Algorithm 14: merge(left, right)

Algorithm 15: merge(left, right)

Input: Aufsteigend sortierte Listen *left* und *right* (Längen n_{left} , n_{right})

Output: Aufsteigend sortierte Liste aus *left* und *right*

Algorithm 16: merge(left, right)

Input: Aufsteigend sortierte Listen *left* und *right* (Längen n_{left} , n_{right})

Output: Aufsteigend sortierte Liste aus *left* und *right*

1 *res* := (); // Initialisiere leere merge-Liste

Algorithm 17: merge(left, right)

Input: Aufsteigend sortierte Listen *left* und *right* (Längen n_{left} , n_{right})

Output: Aufsteigend sortierte Liste aus *left* und *right*

- 1 *res* := (); // Initialisiere leere merge-Liste
- 2 **while** *left* \neq () **and** *right* \neq () **do** // Beide Listen enthalten noch Elemente

8 **end**

Algorithm 19: merge(left, right)

Input: Aufsteigend sortierte Listen *left* und *right* (Längen n_{left} , n_{right})

Output: Aufsteigend sortierte Liste aus *left* und *right*

```
1 res := (); // Initialisiere leere merge-Liste
2 while left ≠ () and right ≠ () do // Beide Listen enthalten noch Elemente
3   | if left1 ≤ right1 then
4   |   | res := res + left1; left = cut(left, 2, nleft);
5   |   else
6   |   |
7   |   end
8 end
```

Algorithm 20: merge(left, right)

Input: Aufsteigend sortierte Listen *left* und *right* (Längen n_{left} , n_{right})

Output: Aufsteigend sortierte Liste aus *left* und *right*

```
1 res := (); // Initialisiere leere merge-Liste
2 while left ≠ () and right ≠ () do // Beide Listen enthalten noch Elemente
3   | if left1 ≤ right1 then
4   |   | res := res + left1; left = cut(left, 2, nleft);
5   | else
6   |   | res := res + right1; right = cut(right, 2, nright);
7   | end
8 end
```


Algorithm 21: merge(left, right)

Input: Aufsteigend sortierte Listen *left* und *right* (Längen n_{left} , n_{right})

Output: Aufsteigend sortierte Liste aus *left* und *right*

```
1 res := (); // Initialisiere leere merge-Liste
2 while left ≠ () and right ≠ () do // Beide Listen enthalten noch Elemente
3   | if left1 ≤ right1 then
4   |   | res := res + left1; left = cut(left, 2, nleft);
5   |   else
6   |   | res := res + right1; right = cut(right, 2, nright);
7   |   end
8 end
9 res := (res + left) + right; // Es gilt left = () ∨ right = ()
```

Algorithm 22: merge(left, right)

Input: Aufsteigend sortierte Listen *left* und *right* (Längen n_{left} , n_{right})

Output: Aufsteigend sortierte Liste aus *left* und *right*

```
1 res := (); // Initialisiere leere merge-Liste
2 while left ≠ () and right ≠ () do // Beide Listen enthalten noch Elemente
3   | if left1 ≤ right1 then
4   |   | res := res + left1; left = cut(left, 2, nleft);
5   |   else
6   |   | res := res + right1; right = cut(right, 2, nright);
7   |   end
8 end
9 res := (res + left) + right; // Es gilt left = () ∨ right = ()
10 return res;
```

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort

(rekursiv)

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort

(rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte).

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort

(rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements.

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort

(rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

Definition 13: Quicksort

(rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

Definition 13: Quicksort

(rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Verfolgt *hard split, easy join*. Das Aufteilen ist aufwändig (sortieren), das Zusammenfügen trivial.

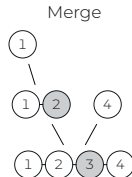
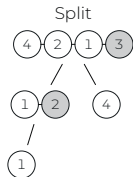
SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort

(rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Verfolgt *hard split, easy join*. Das Aufteilen ist aufwändig (sortieren), das Zusammenfügen trivial.



QUICKSORT IMPLEMENTIEREN

Aufgabe 47: Quicksort implementieren

(6 Minuten)

Gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Aufgabe 47: Quicksort implementieren

(6 Minuten)

Gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Schreiben Sie eine Methode `void quickSort(int[] arr)`, die das übergebene Array mittels Quicksort aufsteigend sortiert. Als Pivot-Element soll stets das erste Element gewählt werden.

Aufgabe 47: Quicksort implementieren

(6 Minuten)

Gegeben sei eine Methode `void swap(int[] arr, int i, int j)`, die die Elemente mit Index i und j in `arr` vertauscht.

Schreiben Sie eine Methode `void quickSort(int[] arr)`, die das übergebene Array mittels Quicksort aufsteigend sortiert. Als Pivot-Element soll stets das erste Element gewählt werden.

Tip: Arbeiten Sie mit einer Hilfsmethode `void quickSort(int[] arr, int l, int r)`, welche das Quicksort-Verfahren auf den Array-Ausschnitt $[l, r]$ anwendet.

QUICKSORT IMPLEMENTIEREN – LÖSUNG

Lösung 47: Quicksort implementieren

```
public static void quickSort(int[] arr) {  
  
}
```


Lösung 47: Quicksort implementieren

```
public static void quickSort(int[] arr) {  
    quickSort(arr, 0, arr.length - 1);  
}
```


Lösung 47: Quicksort implementieren

```
public static void quickSort(int[] arr) {  
    quickSort(arr, 0, arr.length - 1);  
}  
  
static void quickSort(int[] arr, int l, int r) {  
    if(l >= r) return;  
  
}
```

Lösung 47: Quicksort implementieren

```
public static void quickSort(int[] arr) {  
    quickSort(arr, 0, arr.length - 1);  
}  
  
static void quickSort(int[] arr, int l, int r) {  
    if(l >= r) return;  
    int div = partition(arr, l, r); // Nächste Folie  
  
}
```

Lösung 47: Quicksort implementieren

```
public static void quickSort(int[] arr) {
    quickSort(arr, 0, arr.length - 1);
}

static void quickSort(int[] arr, int l, int r) {
    if(l >= r) return;
    int div = partition(arr, l, r); // Nächste Folie
    quickSort(arr, l, div - 1);
    quickSort(arr, div + 1, r);
}
```

Lösung 47: Quicksort implementieren

(Fortsetzung)

Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {  
    int end = r;  
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen  
  
}
```


Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {  
    int end = r;  
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen  
    int pivot = arr[end];  
    l -= 1; // durch do wird l++ einmal ausgeführt: wir setzen l zurück  
    // ebenso, wird so zugesichert, dass r eins links von l ist.  
  
}
```

Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {
    int end = r;
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen
    int pivot = arr[end];
    l -= 1; // durch do wird l++ einmal ausgeführt: wir setzen l zurück
    // ebenso, wird so zugesichert, dass r eins links von l ist.

    while (l < r) {

    }

}
```

Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {
    int end = r;
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen
    int pivot = arr[end];
    l -= 1; // durch do wird l++ einmal ausgeführt: wir setzen l zurück
    // ebenso, wird so zugesichert, dass r eins links von l ist.

    while (l < r) {
        do { l++; } while (arr[l] < pivot);

    }

}
```

Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {
    int end = r;
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen
    int pivot = arr[end];
    l -= 1; // durch do wird l++ einmal ausgeführt: wir setzen l zurück
    // ebenso, wird so zugesichert, dass r eins links von l ist.

    while (l < r) {
        do { l++; } while (arr[l] < pivot);
        do { r--; } while (arr[r] >= pivot && l < r);
    }
}
```

Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {
    int end = r;
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen
    int pivot = arr[end];
    l -= 1; // durch do wird l++ einmal ausgeführt: wir setzen l zurück
    // ebenso, wird so zugesichert, dass r eins links von l ist.

    while (l < r) {
        do { l++; } while (arr[l] < pivot);
        do { r--; } while (arr[r] >= pivot && l < r);
        if (l < r) swap(arr, l, r);
    }
}
```

Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {
    int end = r;
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen
    int pivot = arr[end];
    l -= 1; // durch do wird l++ einmal ausgeführt: wir setzen l zurück
    // ebenso, wird so zugesichert, dass r eins links von l ist.

    while (l < r) {
        do { l++; } while (arr[l] < pivot);
        do { r--; } while (arr[r] >= pivot && l < r);
        if (l < r) swap(arr, l, r);
    }
    swap(arr, l, end); // Tausche Pivot zu Marker
}
```


Lösung 47: Quicksort implementieren

(Fortsetzung)

```
static int partition(int[] arr, int l, int r) {
    int end = r;
    swap(arr, l, end); // Einfach: Pivotelement ans Ende tauschen
    int pivot = arr[end];
    l -= 1; // durch do wird l++ einmal ausgeführt: wir setzen l zurück
    // ebenso, wird so zugesichert, dass r eins links von l ist.

    while (l < r) {
        do { l++; } while (arr[l] < pivot);
        do { r--; } while (arr[r] >= pivot && l < r);
        if (l < r) swap(arr, l, r);
    }
    swap(arr, l, end); // Tausche Pivot zu Marker
    return l;
}
```

SORTIERVERFAHREN: HEAPSORT

Definition 14: Heapsort

(rekursiv)

Definition 14: Heapsort

(rekursiv)

Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente.

Definition 14: Heapsort

(rekursiv)

Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente. Sukzessives Entfernen des Wurzelements (aktuelles Minimum), zur Wiederherstellung der Heap-Eigenschaft.

Definition 14: Heapsort

(rekursiv)

Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente. Sukzessives Entfernen des Wurzelements (aktuelles Minimum), zur Wiederherstellung der Heap-Eigenschaft.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

Definition 14: Heapsort

(rekursiv)

Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente. Sukzessives Entfernen des Wurzelements (aktuelles Minimum), zur Wiederherstellung der Heap-Eigenschaft.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Die genaue Funktionsweise eines Heaps veranschaulichen wir gleich separat.

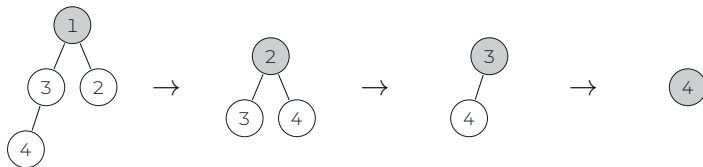
SORTIERVERFAHREN: HEAPSORT

Definition 14: Heapsort

(rekursiv)

Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente. Sukzessives Entfernen des Wurzelements (aktuelles Minimum), zur Wiederherstellung der Heap-Eigenschaft.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Die genaue Funktionsweise eines Heaps veranschaulichen wir gleich separat.



EXKURS: HEAP

Definition 15: Heap

Definition 15: Heap

Binärbaum, mit: Eltern \leq Kinder (*Min-Heap*, für *Max-Heap*: \geq).

Definition 15: Heap

Binärbaum, mit: Eltern \leq Kinder (*Min-Heap*, für *Max-Heap*: \geq).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch \circ).

Definition 15: Heap

Binärbaum, mit: Eltern \leq Kinder (*Min-Heap*, für *Max-Heap*: \geq).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch \circ). Verletzen diese die Heap-Eigenschaft, werden solange kleinere Kinder nach oben getauscht, bis sie wiederhergestellt ist.

Definition 15: Heap

Binärbaum, mit: Eltern \leq Kinder (*Min-Heap*, für *Max-Heap*: \geq).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch \circ). Verletzen diese die Heap-Eigenschaft, werden solange kleinere Kinder nach oben getauscht, bis sie wiederhergestellt ist.
- Beim Entfernen eines Elements wird das „letzte Blatt“ als neue Wurzel getauscht. Anschließend wird wieder getauscht...

Definition 15: Heap

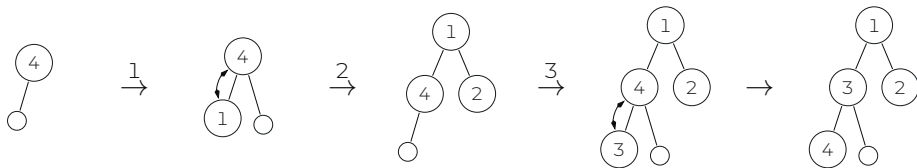
Binärbaum, mit: Eltern \leq Kinder (*Min-Heap*, für *Max-Heap*: \geq).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch \circ). Verletzen diese die Heap-Eigenschaft, werden solange kleinere Kinder nach oben getauscht, bis sie wiederhergestellt ist.
- Beim Entfernen eines Elements wird das „letzte Blatt“ als neue Wurzel getauscht. Anschließend wird wieder getauscht...
- Beispiel: Einfügen der Elemente 4, 2, 1, 3.

Definition 15: Heap

Binärbaum, mit: Eltern \leq Kinder (*Min-Heap*, für *Max-Heap*: \geq).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch \circ). Verletzen diese die Heap-Eigenschaft, werden solange kleinere Kinder nach oben getauscht, bis sie wiederhergestellt ist.
- Beim Entfernen eines Elements wird das „letzte Blatt“ als neue Wurzel getauscht. Anschließend wird wieder getauscht...
- Beispiel: Einfügen der Elemente 4, 2, 1, 3.



ZUR VERTIEFUNG

FANTASTISCHE HEAPS

... und wo sie zu finden sind.
Datastructure Love!

Florian Sihler

27. Juni 2022
SP, Universität Ulm

FANTASTISCHE HEAPS

... und wo sie zu finden sind.
Datastructure Love!

Florian Sihler

27. Juni 2022
SP, Universität Ulm



Mehr zu „Heaps“ per Klick...

SORTIERVERFAHREN: ZUSAMMENFASSUNG

SORTIERVERFAHREN: ZUSAMMENFASSUNG

Verfahren	Laufzeit			Speicher	Ansatz
	best	average	worst		
Selectionsort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	iterativ
Insertionsort	$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	iterativ
Bubblesort	$\mathcal{O}(n^2), \mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	iterativ
Mergesort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	rekursiv
Quicksort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log n)$	rekursiv
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	rekursiv

Dies folgt den Implementationen der Vorlesung. Bereits leichte Modifikationen (wie: prüfe zuerst ob die Liste bereits sortiert ist) können die Daten verändern (beispielsweise einen best-case von $\mathcal{O}(n)$ im Falle von Bubblesort).

EIGENSCHAFT: STABIL

Definition 16: Stabiles Sortierverfahren

Definition 16: Stabiles Sortierverfahren

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge (aus Sicht des Sortierschlüssels) gleicher Element nicht ändert.

Definition 16: Stabiles Sortierverfahren

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge (aus Sicht des Sortierschlüssels) gleicher Element nicht ändert.

- Beispiel: Eine alphabetisch nach Namen sortierte Liste an Personen wird nach dem Alter sortiert.

Definition 16: Stabiles Sortierverfahren

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge (aus Sicht des Sortierschlüssels) gleicher Element nicht ändert.

- Beispiel: Eine alphabetisch nach Namen sortierte Liste an Personen wird nach dem Alter sortiert.
Stabil: die alphabetische Sortierung bleibt bei selbem Alter erhalten.

Definition 16: Stabiles Sortierverfahren

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge (aus Sicht des Sortierschlüssels) gleicher Element nicht ändert.

- Beispiel: Eine alphabetisch nach Namen sortierte Liste an Personen wird nach dem Alter sortiert.

Stabil: die alphabetische Sortierung bleibt bei selbem Alter erhalten.

32	Apfeltine		24	H ans	
24	H ans		24	I nga	<i>Hans sicher</i>
27	Hansine	⇒	27	Hansine	<i>vor Inga.</i>
24	I nga		32	Apfeltine	

Definition 16: Stabiles Sortierverfahren

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge (aus Sicht des Sortierschlüssels) gleicher Element nicht ändert.

- Beispiel: Eine alphabetisch nach Namen sortierte Liste an Personen wird nach dem Alter sortiert.

Stabil: die alphabetische Sortierung bleibt bei selbem Alter erhalten.

32	Apfeltine		24	H ans	
24	H ans		24	I nga	<i>Hans sicher</i>
27	Hansine	⇒	27	Hansine	<i>vor Inga.</i>
24	I nga		32	Apfeltine	

- Stabile Sortierverfahren (nach Implementation der Vorlesung):

Definition 16: Stabiles Sortierverfahren

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge (aus Sicht des Sortierschlüssels) gleicher Element nicht ändert.

- Beispiel: Eine alphabetisch nach Namen sortierte Liste an Personen wird nach dem Alter sortiert.

Stabil: die alphabetische Sortierung bleibt bei selbem Alter erhalten.

32	Apfeltine		24	H ans	
24	H ans		24	I nga	<i>Hans sicher</i>
27	Hansine	⇒	27	Hansine	<i>vor Inga.</i>
24	I nga		32	Apfeltine	

- Stabile Sortierverfahren (nach Implementation der Vorlesung): Insertionsort, Bubblesort und Mergesort.

ÜBUNGSAUFGABE

Aufgabe 48: Sortieralgorithmen abwägen

(2 Minuten)

Quicksort hat eine schlechtere *worst-case* Laufzeit als Mergesort, worin ist es besser als Mergesort?

Lösung 48: Sortieralgorithmen abwägen

Mergesort benötigt einen zusätzlichen Speicherplatz von $\mathcal{O}(n)$, Quicksort ist in-place implementiert und benötigt lediglich $\mathcal{O}(\log n)$ Speicherplatz zum Verwalten der rekursiven Aufrufe. Weiter hängt alles an der Wahl des Pivotelements. Gute Wahlen des Pivotelements (Median der Mediane) erlauben es, die *worst-case* Laufzeit auf $\mathcal{O}(n \log n)$ zu reduzieren (ebenso, wie sich Mergesort In-Place implementieren lässt).

ÜBUNGSAUFGABE

Aufgabe 49: Sortieralgorithmen zeichnen, I (5 Minuten)

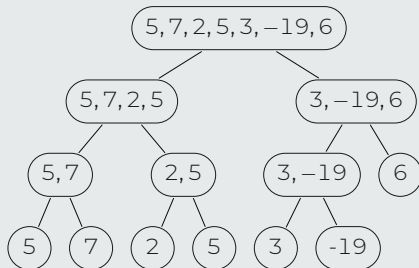
Veranschaulichen Sie den Sortierprozess der Zahlen „5, 7, 2, 5, 3, -19, 6“ unter der Verwendung von Mergesort grafisch. Geben Sie jeden Schritt an und sortieren Sie aufsteigend.

Lösung 49: Sortieralgorithmen zeichnen, I

Hier sind Split- und Mergephase wieder aufgespalten:

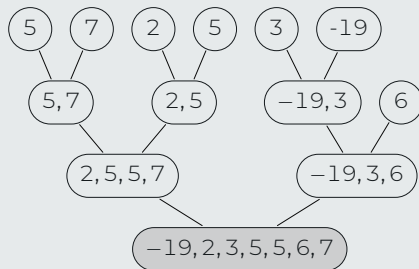
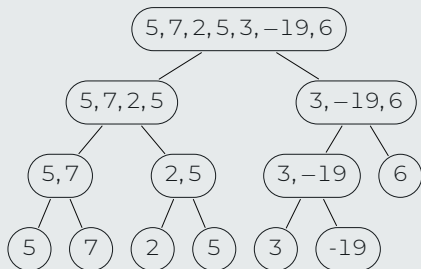
Lösung 49: Sortieralgorithmen zeichnen, I

Hier sind Split- und Mergephase wieder aufgespalten:



Lösung 49: Sortieralgorithmen zeichnen, I

Hier sind Split- und Mergephase wieder aufgespalten:



ÜBUNGSaufGABE

Aufgabe 50: Sortieralgorithmen zeichnen, II

(5 Minuten)

Aufgabe 50: Sortieralgorithmen zeichnen, II

(5 Minuten)

Veranschaulichen Sie den Sortierprozess der Zahlenfolge $3, -5, -4, 2, 6$ unter der Verwendung von Quicksort grafisch. Nutzen Sie das in-place Verfahren durch Vertauschung aus der Vorlesung und sortieren Sie aufsteigend.

Aufgabe 50: Sortieralgorithmen zeichnen, II

(5 Minuten)

Veranschaulichen Sie den Sortierprozess der Zahlenfolge $3, -5, -4, 2, 6$ unter der Verwendung von Quicksort grafisch. Nutzen Sie das in-place Verfahren durch Vertauschung aus der Vorlesung und sortieren Sie aufsteigend. Geben Sie den Zustand der Folge nach jeder Partitionierung aus und markieren Sie die Partitionen.

Aufgabe 50: Sortieralgorithmen zeichnen, II

(5 Minuten)

Veranschaulichen Sie den Sortierprozess der Zahlenfolge $3, -5, -4, 2, 6$ unter der Verwendung von Quicksort grafisch. Nutzen Sie das in-place Verfahren durch Vertauschung aus der Vorlesung und sortieren Sie aufsteigend. Geben Sie den Zustand der Folge nach jeder Partitionierung aus und markieren Sie die Partitionen.

Dabei soll stets das erste Element als Pivot gewählt werden.

Lösung 50: Sortieralgorithmen zeichnen, II

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3}$ — $\textcircled{-5}$ — $\textcircled{-4}$ — $\textcircled{2}$ — $\textcircled{6}$ Tausch ans Ende: $\textcircled{6}$ — $\textcircled{-5}$ — $\textcircled{-4}$ — $\textcircled{2}$ — $\textcircled{3}$



Lösung 50: Sortialgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$



Tausche l und Pivot: $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$



Tausche l und Pivot: $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$



Tausche l und Pivot: $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$

2. Links: $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$



Tausche l und Pivot: $2 - 5 - 4 - \textcircled{3} - 6$

2. Links: $\textcircled{2} - \textcircled{-5} - \textcircled{-4}$ $\textcircled{3} - \textcircled{6}$ Tausch ans Ende: $\textcircled{-4} - \textcircled{-5} - \textcircled{2}$ $\textcircled{3} - \textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

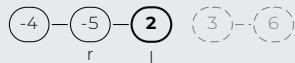
Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$



Tausche l und Pivot: $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$

2. Links: $\textcircled{2} - \textcircled{-5} - \textcircled{-4}$ $\textcircled{3} - \textcircled{6}$ Tausch ans Ende: $\textcircled{-4} - \textcircled{-5} - \textcircled{2}$ $\textcircled{3} - \textcircled{6}$



Lösung 50: Sortieralgorithmen zeichnen, II

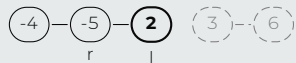
Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$



Tausche l und Pivot: $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$

2. Links: $\textcircled{2} - \textcircled{-5} - \textcircled{-4}$ $\textcircled{3} - \textcircled{6}$ Tausch ans Ende: $\textcircled{-4} - \textcircled{-5} - \textcircled{2}$ $\textcircled{3} - \textcircled{6}$



Tausche l und Pivot: $\textcircled{-4} - \textcircled{-5} - \textcircled{2} - \textcircled{3} - \textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den l - und r -Markern geschieht:

1. $\textcircled{3} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{6}$ Tausch ans Ende: $\textcircled{6} - \textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3}$



Tausche l und Pivot: $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$

2. Links: $\textcircled{2} - \textcircled{-5} - \textcircled{-4}$ $\textcircled{3} - \textcircled{6}$ Tausch ans Ende: $\textcircled{-4} - \textcircled{-5} - \textcircled{2}$ $\textcircled{3} - \textcircled{6}$



Tausche l und Pivot: $\textcircled{-4} - \textcircled{-5} - \textcircled{2} - \textcircled{3} - \textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: 

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4}$ — $\textcircled{-5}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$ Tausch ans Ende: $\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4}$ — $\textcircled{-5}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$ Tausch ans Ende: $\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$

$\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$

r l

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4}$ — $\textcircled{-5}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$ Tausch ans Ende: $\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$

$\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$
r l

Tausche l und Pivot: $\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4}$ — $\textcircled{-5}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$ Tausch ans Ende: $\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$

$\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$
r |

Tausche / und Pivot: $\textcircled{-5}$ — $\textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ — $\textcircled{6}$

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4} - \textcircled{-5}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$ Tausch ans Ende: $\textcircled{-5} - \textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$
 $\textcircled{-5} - \textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$
 r l

Tausche l und Pivot: $\textcircled{-5} - \textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$

4. Rechts in $\textcircled{-4} - \textcircled{-5} - \textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$ ist leer.

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4} - \textcircled{-5}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$ Tausch ans Ende: $\textcircled{-5} - \textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$
 $\textcircled{-5} - \textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$
r l

Tausche l und Pivot: $\textcircled{-5} - \textcircled{-4}$ $\textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$

4. Rechts in $\textcircled{-4} - \textcircled{-5} - \textcircled{2}$ $\textcircled{3}$ $\textcircled{6}$ ist leer. Vertauschungen ändern nichts.

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4} - \textcircled{-5} \textcircled{2} \textcircled{3} - \textcircled{6}$ Tausch ans Ende: $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$
 $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$
r |

Tausche / und Pivot: $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$

4. Rechts in $\textcircled{-4} - \textcircled{-5} - \textcircled{2} \textcircled{3} - \textcircled{6}$ ist leer. Vertauschungen ändern nichts.

5. Rechts in $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$ besteht mit $\textcircled{6}$ nur aus einem Element.

Lösung 50: Sortialgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4} - \textcircled{-5} \textcircled{2} \textcircled{3} - \textcircled{6}$ Tausch ans Ende: $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$
 $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$
 r l

Tausche l und Pivot: $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$

4. Rechts in $\textcircled{-4} - \textcircled{-5} - \textcircled{2} \textcircled{3} - \textcircled{6}$ ist leer. Vertauschungen ändern nichts.

5. Rechts in $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$ besteht mit $\textcircled{6}$ nur aus einem Element. Vertauschungen ändern nichts.

Lösung 50: Sortieralgorithmen zeichnen, II

(Fortsetzung)

3. Links: $\textcircled{-4} - \textcircled{-5} \textcircled{2} \textcircled{3} - \textcircled{6}$ Tausch ans Ende: $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$
 $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$
 r |

Tausche / und Pivot: $\textcircled{-5} - \textcircled{-4} \textcircled{2} \textcircled{3} - \textcircled{6}$

4. Rechts in $\textcircled{-4} - \textcircled{-5} - \textcircled{2} \textcircled{3} - \textcircled{6}$ ist leer. Vertauschungen ändern nichts.

5. Rechts in $\textcircled{2} - \textcircled{-5} - \textcircled{-4} - \textcircled{3} - \textcircled{6}$ besteht mit $\textcircled{6}$ nur aus einem Element. Vertauschungen ändern nichts.

Finales Ergebnis: $\textcircled{-5} - \textcircled{-4} - \textcircled{2} - \textcircled{3} - \textcircled{6}$

Dynamische Datenstrukturen



LISTEN – DYNAMISCHE ARRAYS

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.🌐

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.🌐
- Eine ArrayList ist die einfachste Implementation einer Liste.

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.🌐
- Eine ArrayList ist die einfachste Implementation einer Liste.
- Wir halten intern ein Array, welches wir dynamisch vergrößern oder Verkleinern.

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.🌐
- Eine ArrayList ist die einfachste Implementation einer Liste.
- Wir halten intern ein Array, welches wir dynamisch vergrößern oder Verkleinern.
- Das Problem: Einfüge- und Löschooperationen bedeuten einen großen Aufwand (durch das Kopieren der Elemente).

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.🌐
- Eine ArrayList ist die einfachste Implementation einer Liste.
- Wir halten intern ein Array, welches wir dynamisch vergrößern oder Verkleinern.
- Das Problem: Einfüge- und Löschooperationen bedeuten einen großen Aufwand (durch das Kopieren der Elemente).
- Der Vorteil: Der Zugriff auf ein bestimmtes Element erfolgt (da ein Array zugrunde liegt) in $\mathcal{O}(1)$.

LISTEN – EINFACH VERKETTETE LISTE

LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.

LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **null**) zeigt, nennen wir *Tail* oder *Foot*.

LISTEN – EINFACH VERKETTETE LISTE

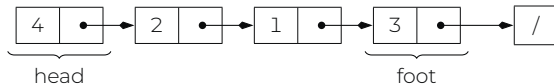
- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **null**) zeigt, nennen wir *Tail* oder *Foot*.

```
class Element {  
    public int value;  
    public Element next;  
    // ...  
}
```


LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **null**) zeigt, nennen wir *Tail* oder *Foot*.

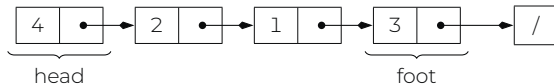
```
class Element {  
    public int value;  
    public Element next;  
    // ...  
}
```



LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **null**) zeigt, nennen wir *Tail* oder *Foot*.

```
class Element {  
    public int value;  
    public Element next;  
    // ...  
}
```



- Doppelt verkettete Listen verbrauchen mehr Speicher und sind komplizierter (da zwei Verweise).

LISTEN – EINFACH VERKETTETE LISTE, II

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste:

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$).

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.
- Hinweis: Manche Operationen benötigen eine Sonderbehandlung im Falle einer leeren Liste.

LISTEN – EINFACH VERKETTETE LISTE, II

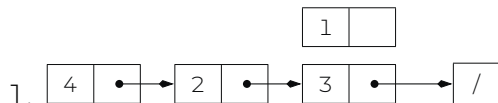
- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.
- Hinweis: Manche Operationen benötigen eine Sonderbehandlung im Falle einer leeren Liste. So zum Beispiel das Einfügen.

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.
- Hinweis: Manche Operationen benötigen eine Sonderbehandlung im Falle einer leeren Liste. So zum Beispiel das Einfügen.
- Betrachten wir einmal das Einfügen eines neuen Elements:

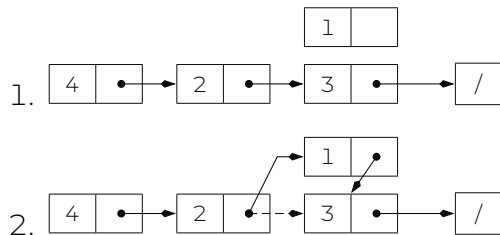
LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.
- Hinweis: Manche Operationen benötigen eine Sonderbehandlung im Falle einer leeren Liste. So zum Beispiel das Einfügen.
- Betrachten wir einmal das Einfügen eines neuen Elements:



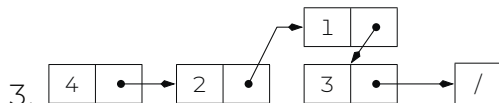
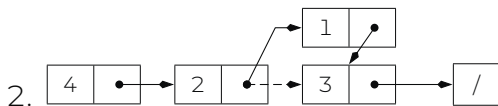
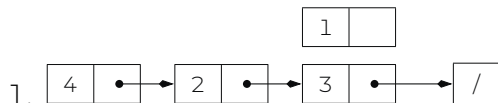
LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.
- Hinweis: Manche Operationen benötigen eine Sonderbehandlung im Falle einer leeren Liste. So zum Beispiel das Einfügen.
- Betrachten wir einmal das Einfügen eines neuen Elements:



LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketteten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.
- Hinweis: Manche Operationen benötigen eine Sonderbehandlung im Falle einer leeren Liste. So zum Beispiel das Einfügen.
- Betrachten wir einmal das Einfügen eines neuen Elements:



ÜBUNGSAUFGABE

Aufgabe 51: Einfach verkettete Liste - Einfügen

(6 Minuten)

Sei der folgende Code gegeben:

Aufgabe 51: Einfach verkettete Liste - Einfügen

(6 Minuten)

Sei der folgende Code gegeben:

```
class Element {
    public int value;
    public Element next;
    public Element(int value, Element next) {
        this.value = value; this.next = next;
    }
}
```

Aufgabe 51: Einfach verkettete Liste - Einfügen

(6 Minuten)

Sei der folgende Code gegeben:

```
class Element {
    public int value;
    public Element next;
    public Element(int value, Element next) {
        this.value = value; this.next = next;
    }
}
```

Schreiben Sie eine Methode `Element appendLast(Element root, int i)`, die ein neues `Element` mit Wert `i` an das Ende der Liste, anhängt und diese Liste zurückliefert. Die Liste soll durch den Kopf `root` gekennzeichnet sein. Sie dürfen die ursprüngliche Liste verändern.

Lösung 51: Einfach verkettete Liste - Einfügen

Die erste Methode behandelt den Sonderfall „leere Liste“. Die Zweite traversiert die Liste so lange, bis wir am *Foot* angekommen sind und ersetzt dann den Verweis des vorletzten Elements:

Lösung 51: Einfach verkettete Liste - Einfügen

Die erste Methode behandelt den Sonderfall „leere Liste“. Die Zweite traversiert die Liste so lange, bis wir am *Foot* angekommen sind und ersetzt dann den Verweis des vorletzten Elements:

```
public Element appendLast(Element root, int i) {
    if(root == null) {
        return new Element(i, null);
    } else {
        appendLastRec(root, i);
        return root;
    }
}
```


Lösung 51: Einfach verkettete Liste - Einfügen

(Fortsetzung)

Wir suchen das Ende der Liste:

```
public void appendLastRec(Element current, int i){
    if(current.next == null) // found last
        current.next = new Element(i, null);
    else
        appendLastRec(current.next, i);
}
```

LISTEN – DOPPELT VERKETTETE LISTE

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“.

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

LISTEN – DOPPELT VERKETTETE LISTE

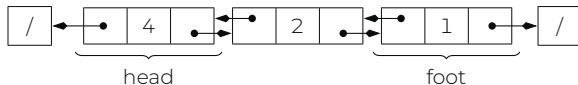
- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

```
class Element {  
    public int value;  
    public Element next;  
    public Element prev;  
    // ...  
}
```

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

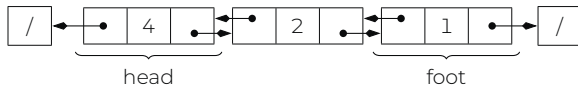
```
class Element {  
    public int value;  
    public Element next;  
    public Element prev;  
    // ...  
}
```



LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

```
class Element {  
    public int value;  
    public Element next;  
    public Element prev;  
    // ...  
}
```



- Den „Datentyp“ einfach verkettete Liste lagern wir in eine neue Klasse `LinkedList` aus, die auch die Operationen wie hinzufügen und löschen übernimmt.

DATENSTRUKTUR: STACK

DATENSTRUKTUR: STACK

- Ein Stack arbeitet nach dem LIFO-Prinzip (Last-In, First-Out).

DATENSTRUKTUR: STACK

- Ein Stack arbeitet nach dem LIFO-Prinzip (Last-In, First-Out).
- Elemente können nur oben auf dem Stapel abgelegt und von dort entnommen werden.

DATENSTRUKTUR: STACK

- Ein Stack arbeitet nach dem LIFO-Prinzip (Last-In, First-Out).
- Elemente können nur oben auf dem Stapel abgelegt und von dort entnommen werden.
- Die Verwaltung rekursiver Methoden erfolgt über eine Art Stack. So „überlagern“ die Parameter im rekursiven Aufruf die alten, bis die Methode wieder verlassen wird.

DATENSTRUKTUR: STACK, II

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren,

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.
- Auf Stacks kennzeichnen wir zwei wichtige Operationen:

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.
- Auf Stacks kennzeichnen wir zwei wichtige Operationen:
PUSH: Legt das Element oben auf dem Stack ab.

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.
- Auf Stacks kennzeichnen wir zwei wichtige Operationen:
 - `PUSH`: Legt das Element oben auf dem Stack ab.
 - `POP`: Entfernt das oberste Element und liefert es zurück.

DATENSTRUKTUR: QUEUE

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).
- Elemente können nur hinten in der Schlange eingereiht und nur vorne von der Schlange entfernen werden.

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).
- Elemente können nur hinten in der Schlange eingereiht und nur vorne von der Schlange entfernen werden.
- Queues können zum Puffern verwendet werden (analog zum Schalter).

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).
- Elemente können nur hinten in der Schlange eingereiht und nur vorne von der Schlange entfernen werden.
- Queues können zum Puffern verwendet werden (analog zum Schalter).
- Es existieren Derivate, wie die *double-ended queue* (ein- und ausreihen auf beiden Seiten) oder die *priority queue* (Elemente mit höherer Priorität werden bei `enqueue` zuerst aus der Schlange genommen).

DATENSTRUKTUR: QUEUE, II

DATENSTRUKTUR: QUEUE, II

- Queues lassen sich als eine Liste implementieren,

DATENSTRUKTUR: QUEUE, II

- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.

DATENSTRUKTUR: QUEUE, II

- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.
- Auf Queues kennzeichnen wir zwei wichtige Operationen:

DATENSTRUKTUR: QUEUE, II

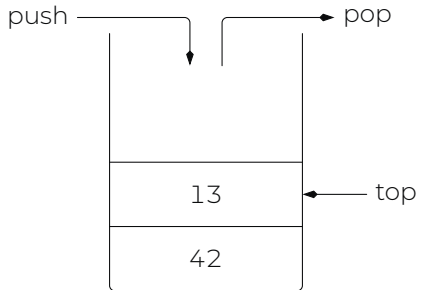
- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.
- Auf Queues kennzeichnen wir zwei wichtige Operationen:
ENQUEUE: Reiht ein Element (hinten) in die Schlange ein.

DATENSTRUKTUR: QUEUE, II

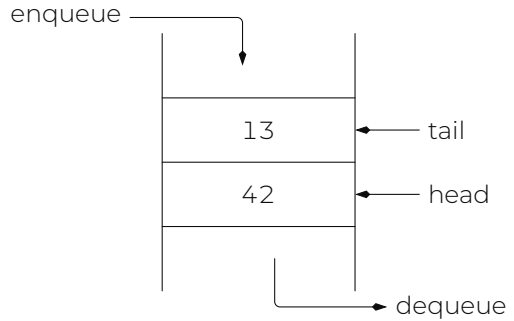
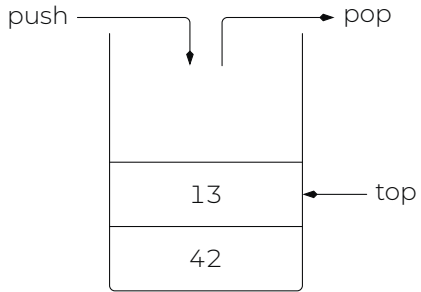
- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.
- Auf Queues kennzeichnen wir zwei wichtige Operationen:
ENQUEUE: Reiht ein Element (hinten) in die Schlange ein.
DEQUEUE: Nimmt ein Element (vorne) aus der Schlange.

VERGLEICH: STACK UND QUEUE

VERGLEICH: STACK UND QUEUE



VERGLEICH: STACK UND QUEUE



BÄUME: FORMAL

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:
WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:
 - WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.
 - BLATT: Ein Blatt ist ein Knoten ohne Nachfolger.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:
 - WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.
 - BLATT: Ein Blatt ist ein Knoten ohne Nachfolger.
 - INNERER: Ein innerer Knoten ist jeder Knoten, der weder Blatt noch Wurzel ist.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:
 - WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.
 - BLATT: Ein Blatt ist ein Knoten ohne Nachfolger.
 - INNERER: Ein innerer Knoten ist jeder Knoten, der weder Blatt noch Wurzel ist. Ein innerer Knoten hat genau einen Elternknoten.

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$.

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.
- Hier ist jeder Knoten gleichzeitig die „Wurzel“ eines Teilbaumes

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.
- Hier ist jeder Knoten gleichzeitig die „Wurzel“ eines Teilbaumes der entweder leer sein ($\hat{=}$ Blatt)

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.
- Hier ist jeder Knoten gleichzeitig die „Wurzel“ eines Teilbaumes der entweder leer sein ($\hat{=}$ Blatt) oder noch weitere Knoten enthalten kann ($\hat{=}$ innerer Knoten).

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.
- Hier ist jeder Knoten gleichzeitig die „Wurzel“ eines Teilbaumes der entweder leer sein ($\hat{=}$ Blatt) oder noch weitere Knoten enthalten kann ($\hat{=}$ innerer Knoten).
- Jedem Knoten ordnen wir eine *Ebene* zu.

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.
- Hier ist jeder Knoten gleichzeitig die „Wurzel“ eines Teilbaumes der entweder leer sein ($\hat{=}$ Blatt) oder noch weitere Knoten enthalten kann ($\hat{=}$ innerer Knoten).
- Jedem Knoten ordnen wir eine *Ebene* zu. Sie entspricht der Länge des Pfades von der Wurzel zum Knoten.

Definition 17: Baum

Ein Baum B ist ein zusammenhängender, azyklischer Graph $T = (V, E)$ mit einer endlichen Menge an Knoten V und Kanten $E \subseteq V \times V$. Die Anzahl der eingehenden Kanten muss für jeden Knoten maximal 1 sein.

- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.
- Hier ist jeder Knoten gleichzeitig die „Wurzel“ eines Teilbaumes der entweder leer sein ($\hat{=}$ Blatt) oder noch weitere Knoten enthalten kann ($\hat{=}$ innerer Knoten).
- Jedem Knoten ordnen wir eine *Ebene* zu. Sie entspricht der Länge des Pfades von der Wurzel zum Knoten. Der Wurzelknoten hat die Ebene 1.

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.
- Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kinder.

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.
- Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kinder.
- Binärbäume sind von einer besonderen Bedeutung für die Informatik.

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.
- Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kinder.
- Binärbäume sind von einer besonderen Bedeutung für die Informatik.
(Sie lassen sich z.B. einfach in einem Array repräsentieren.)

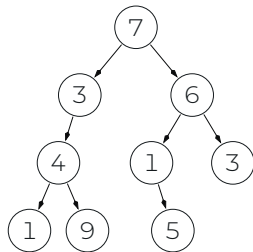
BÄUME: IMPLEMENTATION

BÄUME: IMPLEMENTATION

```
class Node {  
    public int value;  
    public Node left;  
    public Node right;  
    // ...  
}
```

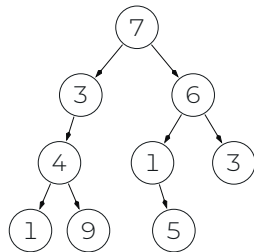
BÄUME: IMPLEMENTATION

```
class Node {  
    public int value;  
    public Node left;  
    public Node right;  
    // ...  
}
```



BÄUME: IMPLEMENTATION

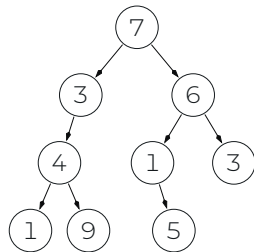
```
class Node {  
    public int value;  
    public Node left;  
    public Node right;  
    // ...  
}
```



- Wie bei einer einfach verketteten Liste können wir den Baum traversieren.

BÄUME: IMPLEMENTATION

```
class Node {  
    public int value;  
    public Node left;  
    public Node right;  
    // ...  
}
```



- Wie bei einer einfach verketteten Liste können wir den Baum traversieren.
- Analog zur doppelt verketteten Liste gibt es einen Binärbaum mit Verweis auf den Elternknoten.

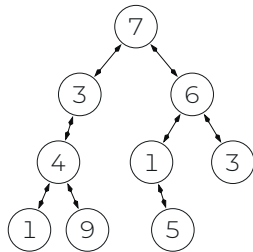
BÄUME: IMPLEMENTATION, II

BÄUME: IMPLEMENTATION, II

```
class Node {  
    public Node parent;  
  
    public int value;  
    public Node left;  
    public Node right;  
    // ...  
}
```

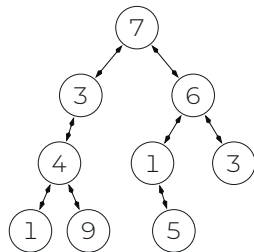
BÄUME: IMPLEMENTATION, II

```
class Node {  
    public Node parent;  
  
    public int value;  
    public Node left;  
    public Node right;  
    // ...  
}
```



BÄUME: IMPLEMENTATION, II

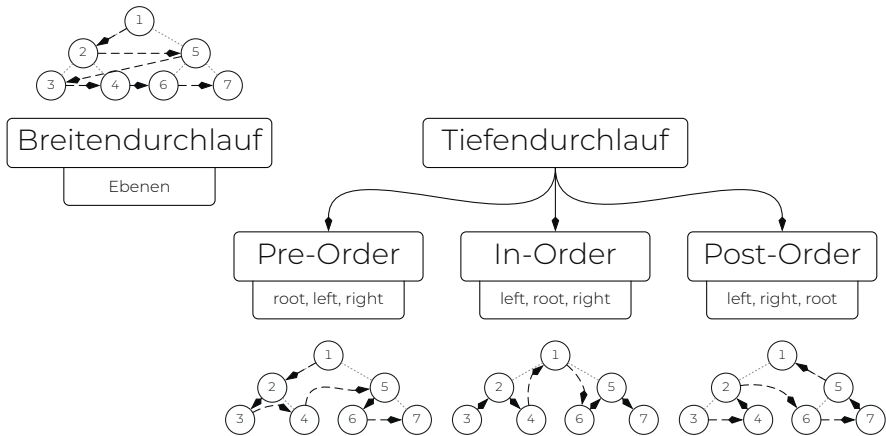
```
class Node {  
    public Node parent;  
  
    public int value;  
    public Node left;  
    public Node right;  
    // ...  
}
```



- Diese Verkettung kann vorteilhaft sein und erspart beispielsweise die Rekursion bei der Traversierung.

TRAVERSIERUNGSVERFAHREN

TRAVERSIERUNGSVERFAHREN



TRAVERSIERUNG: LEVEL-ORDER

TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.

TRAVERSIERUNG: LEVEL-ORDER

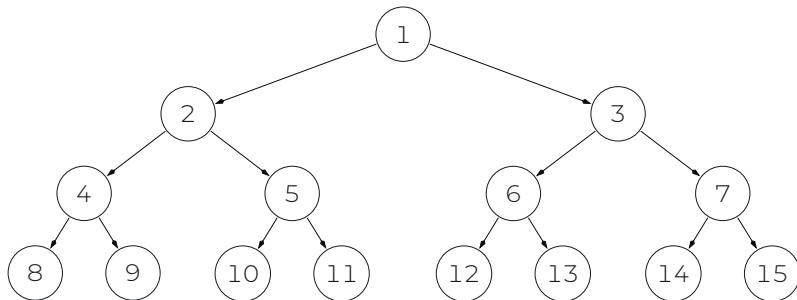
- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.

TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.
- Es handelt sich um einen Breitendurchlauf (*breadth-first*).

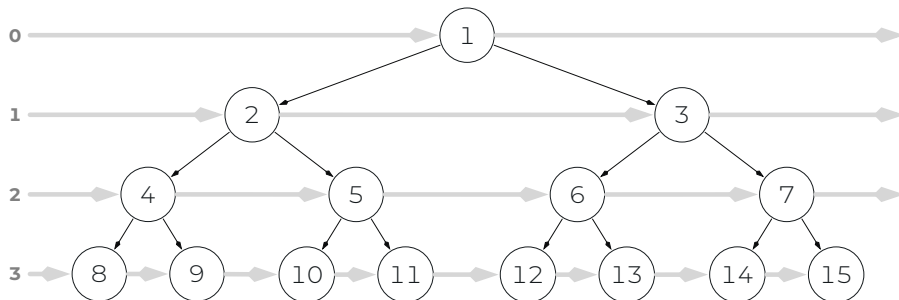
TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.
- Es handelt sich um einen Breitendurchlauf (*breadth-first*).



TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.
- Es handelt sich um einen Breitendurchlauf (*breadth-first*).



TRAVERSIERUNG: PRE-ORDER

TRAVERSIERUNG: PRE-ORDER

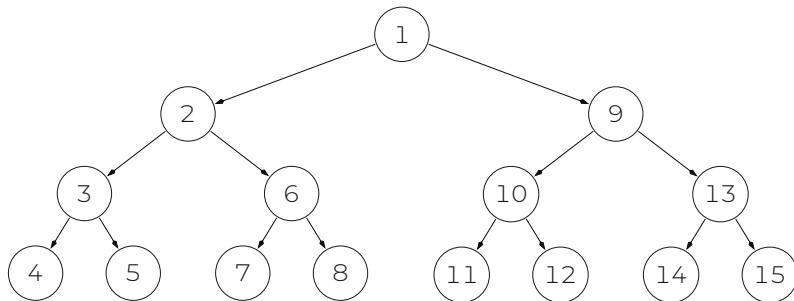
- Es handelt sich um einen Tiefendurchlauf (*depth-first*).

TRAVERSIERUNG: PRE-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit der Pre-Order Strategie werden erst der aktuelle Knoten und dann rekursiv beide Teilbäume nach der gleichen Regel besucht.

TRAVERSIERUNG: PRE-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit der Pre-Order Strategie werden erst der aktuelle Knoten und dann rekursiv beide Teilbäume nach der gleichen Regel besucht.



TRAVERSIERUNG: IN-ORDER

TRAVERSIERUNG: IN-ORDER

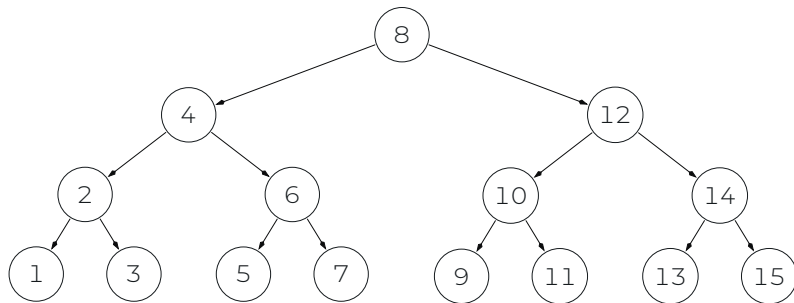
- Es handelt sich um einen Tiefendurchlauf (*depth-first*).

TRAVERSIERUNG: IN-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit dieser Strategie wird erst rekursiv ein Teilbaum bearbeitet, dann der Knoten selbst und dann rekursiv der andere Teilbaum.

TRAVERSIERUNG: IN-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit dieser Strategie wird erst rekursiv ein Teilbaum bearbeitet, dann der Knoten selbst und dann rekursiv der andere Teilbaum.



TRAVERSIERUNG: POST-ORDER

TRAVERSIERUNG: POST-ORDER

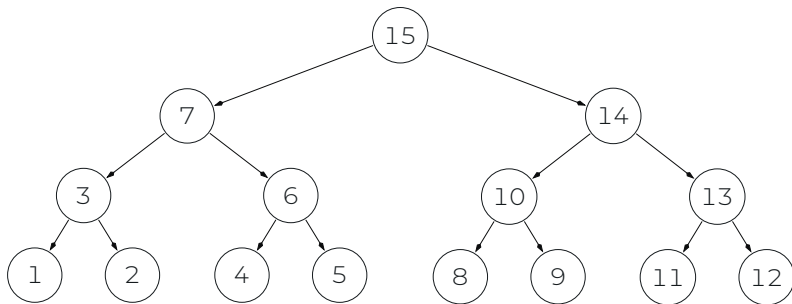
- Es handelt sich um einen Tiefendurchlauf (*depth-first*).

TRAVERSIERUNG: POST-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit der Post-Order Strategie werden erst rekursiv beide Teilbäume und dann der Knoten selbst bearbeitet.

TRAVERSIERUNG: POST-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit der Post-Order Strategie werden erst rekursiv beide Teilbäume und dann der Knoten selbst bearbeitet.



ZUR VERTIEFUNG

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Sihler

13. Juli 2022
SP, Universität Ulm

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Sihler

13. Juli 2022
SP, Universität Ulm



Mehr zu „Traversierungen“ per Klick...

SUCHBÄUME

- Bäume können auch zur Sortierung nützliche Datenstrukturen sein.

SUCHBÄUME

- Bäume können auch zur Sortierung nützliche Datenstrukturen sein.
- So existieren Suchbäume. (Alle Elemente im linken Teilbaum sind kleiner oder gleich der Wurzel, alle rechten größer \Rightarrow binäre Suche.)

SUCHBÄUME

- Bäume können auch zur Sortierung nützliche Datenstrukturen sein.
- So existieren Suchbäume. (Alle Elemente im linken Teilbaum sind kleiner oder gleich der Wurzel, alle rechten größer \Rightarrow binäre Suche.)
- Vergleiche hierzu auch die Datenstruktur Heap.

- Bäume können auch zur Sortierung nützliche Datenstrukturen sein.
- So existieren Suchbäume. (Alle Elemente im linken Teilbaum sind kleiner oder gleich der Wurzel, alle rechten größer \Rightarrow binäre Suche.)
- Vergleiche hierzu auch die Datenstruktur Heap.
- Die Verfahren lassen sich auch auf Bäume mit höherem Verzweigungsgrad anwenden!

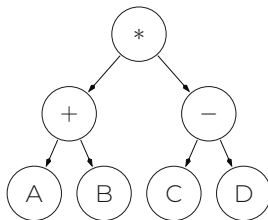
ARITHMETISCHE BÄUME

ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.

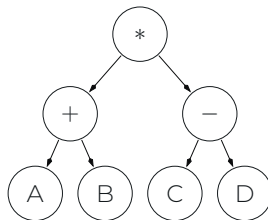
ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.



ARITHMETISCHE BÄUME

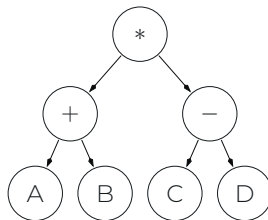
- Binärbäume helfen uns, arithmetische Operationen darzustellen.



- Die Traversierungsverfahren entsprechen Notationsschemata:

ARITHMETISCHE BÄUME

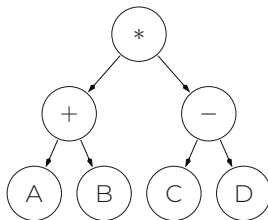
- Binärbäume helfen uns, arithmetische Operationen darzustellen.



- Die Traversierungsverfahren entsprechen Notationsschemata:
 - IN-ORDER: repräsentiert die geläufige Infix-Notation: $(A + B) * (C - D)$.
(Die Klammern entsprechen der Ausführreihenfolge und sind kein Teil.)

ARITHMETISCHE BÄUME

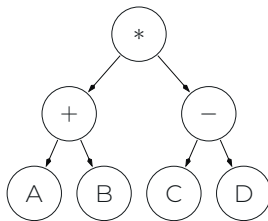
- Binärbäume helfen uns, arithmetische Operationen darzustellen.



- Die Traversierungsverfahren entsprechen Notationsschemata:
 - IN-ORDER: repräsentiert die geläufige Infix-Notation: $(A + B) * (C - D)$.
(Die Klammern entsprechen der Ausführreihenfolge und sind kein Teil.)
 - PRE-ORDER: entspricht der Präfix-Notation: $* + AB - CD$.

ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.



- Die Traversierungsverfahren entsprechen Notationsschemata:
 - IN-ORDER: repräsentiert die geläufige Infix-Notation: $(A + B) * (C - D)$.
(Die Klammern entsprechen der Ausführreihenfolge und sind kein Teil.)
 - PRE-ORDER: entspricht der Präfix-Notation: $* + AB - CD$.
 - POST-ORDER: entspricht der Postfix-Notation: $AB + CD - *$.

GRAPHEN

- Im Gegensatz zu Bäumen, können Knoten in Graphen auch mehrere Vorgänger haben.

- Im Gegensatz zu Bäumen, können Knoten in Graphen auch mehrere Vorgänger haben.

Definition 18: Ungerichteter Graph

Ist ein Tupel $G = (V, E)$ aus Knoten V und Kanten $E \subseteq \binom{V}{2}$.

- Im Gegensatz zu Bäumen, können Knoten in Graphen auch mehrere Vorgänger haben.

Definition 18: Ungerichteter Graph

Ist ein Tupel $G = (V, E)$ aus Knoten V und Kanten $E \subseteq \binom{V}{2}$.

- Die Graphdefinition gleicht der aus „Formale Grundlagen“. Sie wird hier zusammengefasst.

- Im Gegensatz zu Bäumen, können Knoten in Graphen auch mehrere Vorgänger haben.

Definition 18: Ungerichteter Graph

Ist ein Tupel $G = (V, E)$ aus Knoten V und Kanten $E \subseteq \binom{V}{2}$.

- Die Graphdefinition gleicht der aus „Formale Grundlagen“. Sie wird hier zusammengefasst.

Definition 19: Gerichteter Graph

Ein gerichteter Graph definiert sich ähnlich zum ungerichteten, definiert die Kanten aber als Tupel $E \subseteq V \times V$.

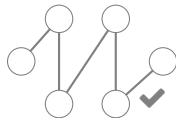
BEGRIFFE IN EINEM GRAPHEN

BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.

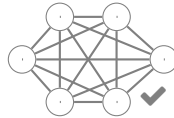
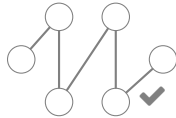
BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



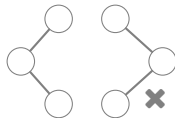
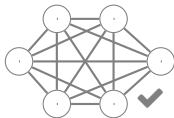
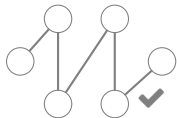
BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



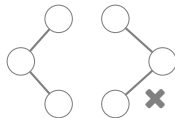
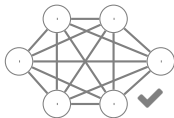
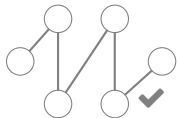
BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



BEGRIFFE IN EINEM GRAPHEN

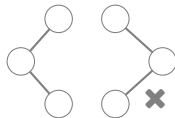
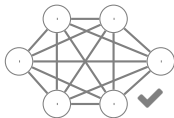
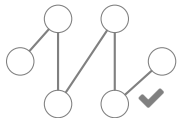
ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind.

BEGRIFFE IN EINEM GRAPHEN

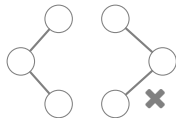
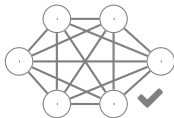
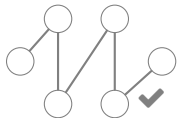
ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



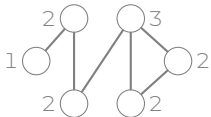
GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.

BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.

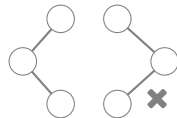
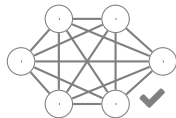
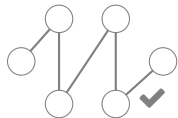


GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.

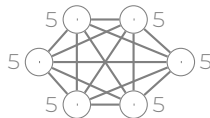
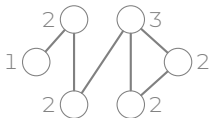


BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.

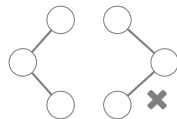
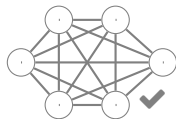
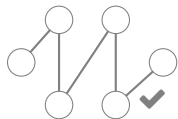


GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.

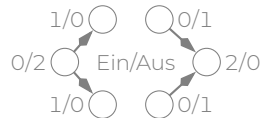
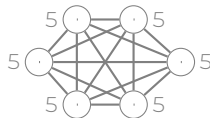
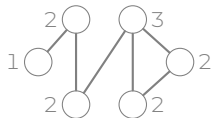


BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.



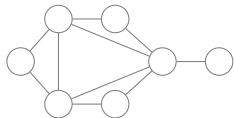
BEGRIFFE IN EINEM GRAPHEN, II

BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.

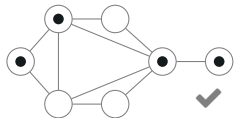
BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



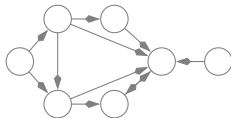
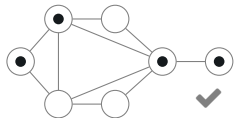
BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



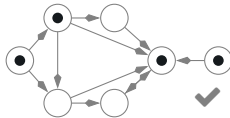
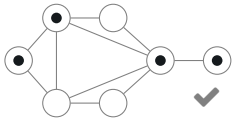
BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



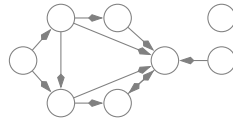
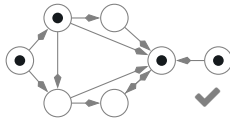
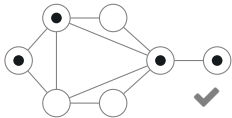
BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



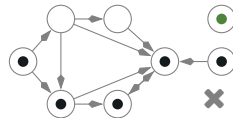
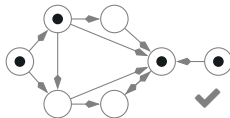
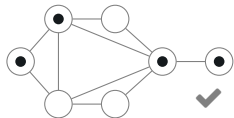
BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



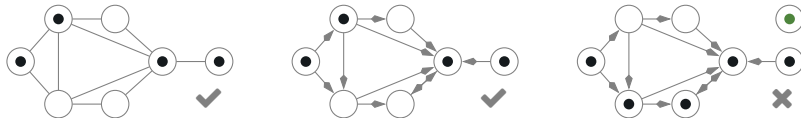
BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



BEGRIFFE IN EINEM GRAPHEN, II

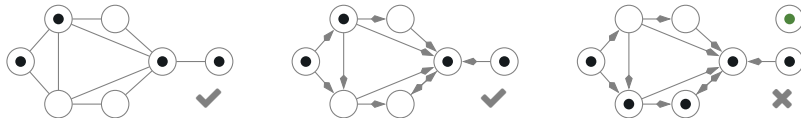
WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



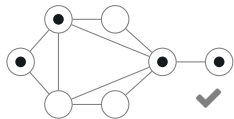
PFAD: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.

BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.

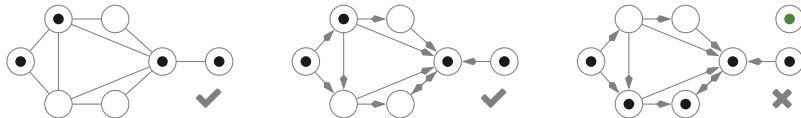


PFAD: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.

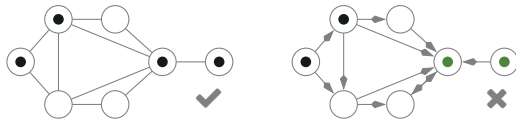


BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.

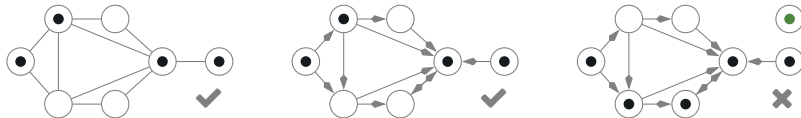


PFAD: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.

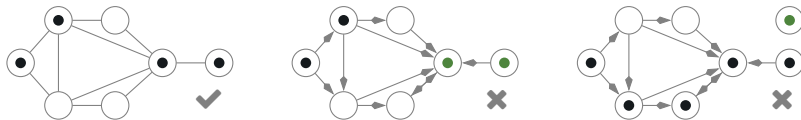


BEGRIFFE IN EINEM GRAPHEN, II

WEG: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



PFAD: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.



VARIANTEN VON GRAPHEN

VARIANTEN VON GRAPHEN

- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \rightarrow M$ einen Wert aus einer Menge M zu. (Beispielsweise $M = \mathbb{N}$)

VARIANTEN VON GRAPHEN

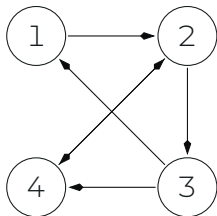
- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \rightarrow M$ einen Wert aus einer Menge M zu. (Beispielsweise $M = \mathbb{N}$)
- Wir können einen Graphen durch eine *Adjazenzmatrix* beschreiben.

VARIANTEN VON GRAPHEN

- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \rightarrow M$ einen Wert aus einer Menge M zu. (Beispielsweise $M = \mathbb{N}$)
- Wir können einen Graphen durch eine *Adjazenzmatrix* beschreiben. Hierbei geben die jeweiligen Zellen m_{ij} , ob eine Verbindung vom Knoten i zum Knoten j besteht ($m_{ij} > 0$ für eine Kante).

VARIANTEN VON GRAPHEN

- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \rightarrow M$ einen Wert aus einer Menge M zu. (Beispielsweise $M = \mathbb{N}$)
- Wir können einen Graphen durch eine *Adjazenzmatrix* beschreiben. Hierbei geben die jeweiligen Zellen m_{ij} , ob eine Verbindung vom Knoten i zum Knoten j besteht ($m_{ij} > 0$ für eine Kante).



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \left(\begin{array}{cccc} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{array} \right) \end{array}$$

ADJAZENZLISTEN

ADJAZENZLISTEN

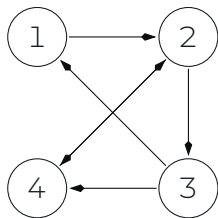
- Adjazenzlisten sind eine andere Variante Graphen zu repräsentieren.

ADJAZENZLISTEN

- Adjazenzlisten sind eine andere Variante Graphen zu repräsentieren.
- Hier hält jeder Knoten eine (verkettete) Liste an seinen benachbarten Knoten.

ADJAZENZLISTEN

- Adjazenzlisten sind eine andere Variante Graphen zu repräsentieren.
- Hier hält jeder Knoten eine (verkettete) Liste an seinen benachbarten Knoten.



1. → 2
2. → 3 → 4
3. → 1 → 4
4. → 2

ALGORITHMEN AUF GRAPHEN

ALGORITHMEN AUF GRAPHEN

- Eine gängiges Problem ist es herauszufinden ob ein Graph Zyklen enthält und/oder ob er zusammenhängend ist.

ALGORITHMEN AUF GRAPHEN

- Eine gängiges Problem ist es herauszufinden ob ein Graph Zyklen enthält und/oder ob er zusammenhängend ist.
- Wegfindungsproblem wie das kürzeste Wege Probleme (Routenfindung, ...), Traveling Salesman Problem.

ALGORITHMEN AUF GRAPHEN

- Eine gängiges Problem ist es herauszufinden ob ein Graph Zyklen enthält und/oder ob er zusammenhängend ist.
- Wegfindungsproblem wie das kürzeste Wege Probleme (Routenfindung, ...), Traveling Salesman Problem.
- Finden des minimalen, aufspannenden Baum.

ALGORITHMEN AUF GRAPHEN

- Eine gängiges Problem ist es herauszufinden ob ein Graph Zyklen enthält und/oder ob er zusammenhängend ist.
- Wegfindungsproblem wie das kürzeste Wege Probleme (Routenfindung, ...), Traveling Salesman Problem.
- Finden des minimalen, aufspannenden Baum.
- Exemplarisch betrachtet die Vorlesung die Breiten- und die Tiefensuche.

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

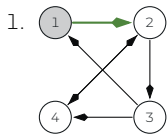
- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.

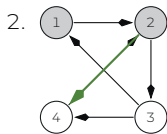
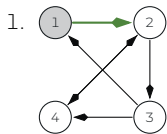
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.



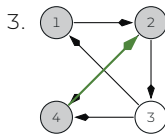
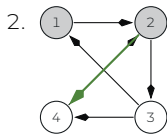
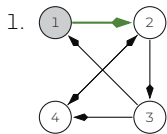
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.



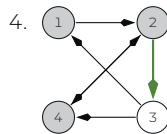
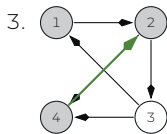
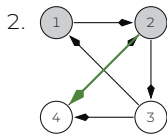
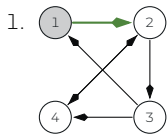
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.



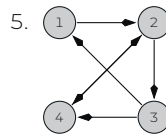
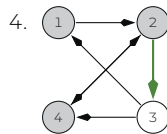
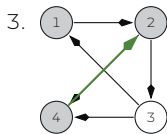
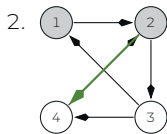
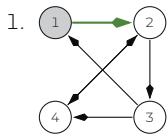
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.



GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.



EIN BEISPIEL FÜR DIE TIEFENSUCHE

EIN BEISPIEL FÜR DIE TIEFENSUCHE

TRAVERSIERUNGSVARIANTEN

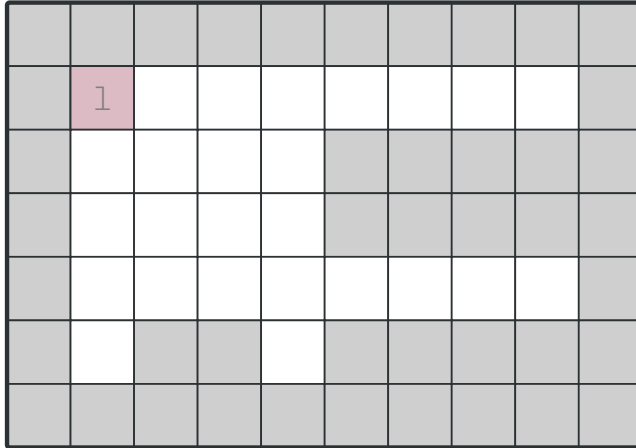
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

		1							
	21	22							

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	●		1						
	20	21	22						

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•		1					
	19	20	21	6					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•		1				
	18	19	20	5					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•		1			
	17	18	19	4					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•		1		
	16	17	18	3					

TRAVERSIERUNGSVARIANTEN

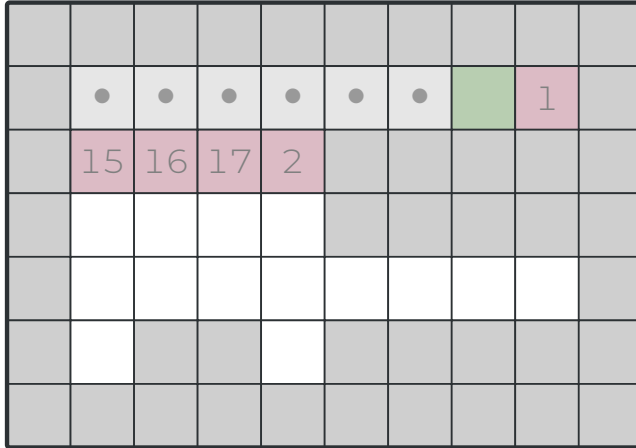
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

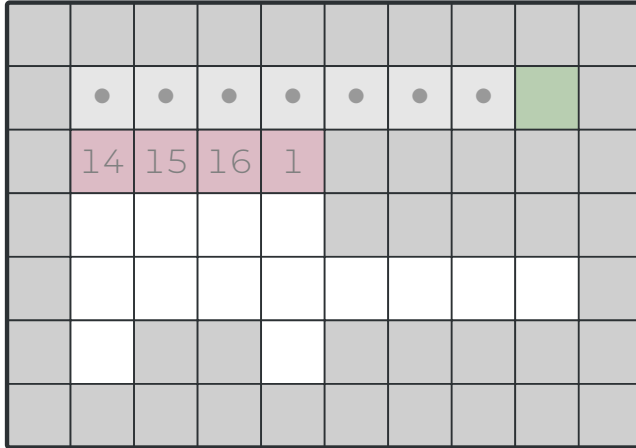
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	13	14	15						
			16	1					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	12	13	14	•					
			15						
			7	1	3				

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	11	12	13	•					
			14	•					
			6		2				
				1					

TRAVERSIERUNGSVARIANTEN

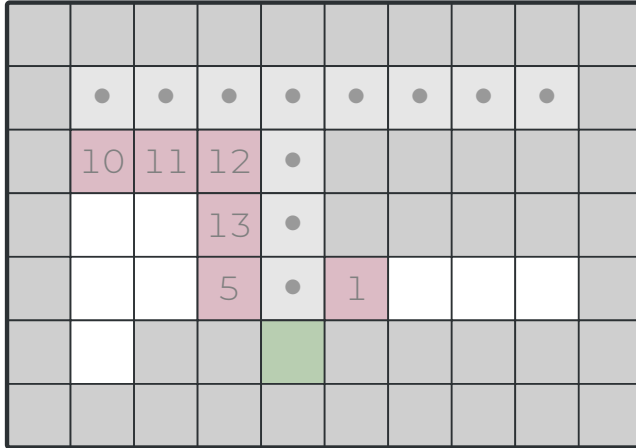
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	9	10	11	•					
			12	•					
			4	•		1			
				•					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	8	9	10	•					
			11	•					
			3	•	•		1		
				•					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	7	8	9	•					
			10	•					
			2	•	•	•		1	
				•					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	6	7	8	•					
			9	•					
			1	•	•	•	•		
				•					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	5	6	7	•					
		9	8	•					
		1		•	•	•	•	•	
				•					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	4	5	6	•					
	3	8	7	•					
	1		•	•	•	•	•	•	
	2			•					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE

	•	•	•	•	•	•	•	•	
	3	4	5	•					
	2	7	6	•					
		•	•	•	•	•	•	•	
	1			•					

TRAVERSIERUNGSVARIANTEN

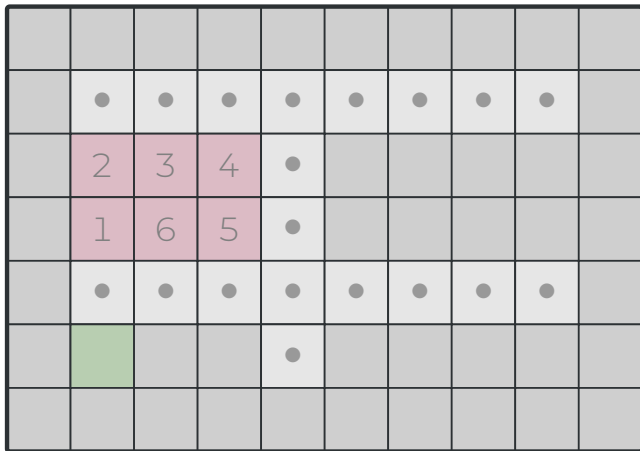
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

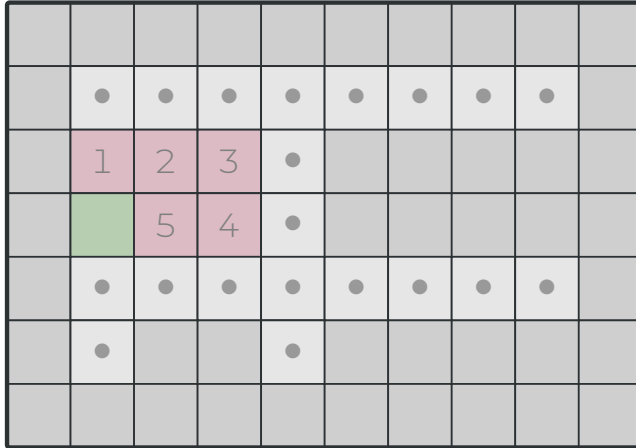
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

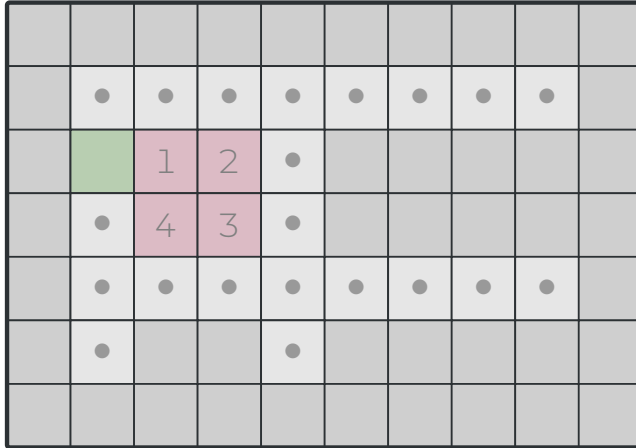
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

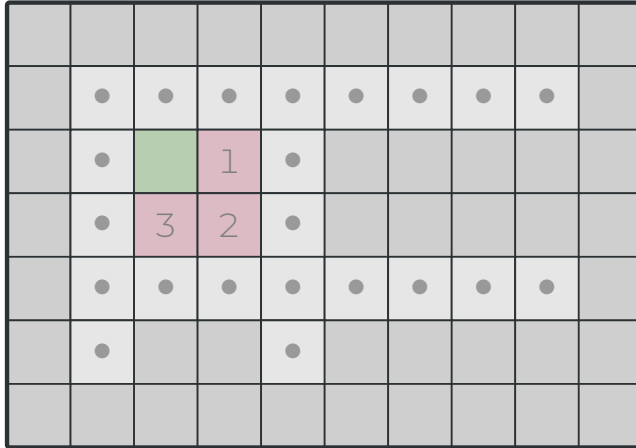
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

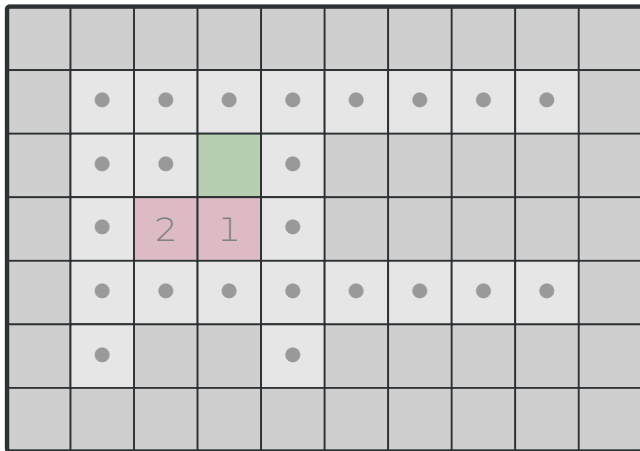
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

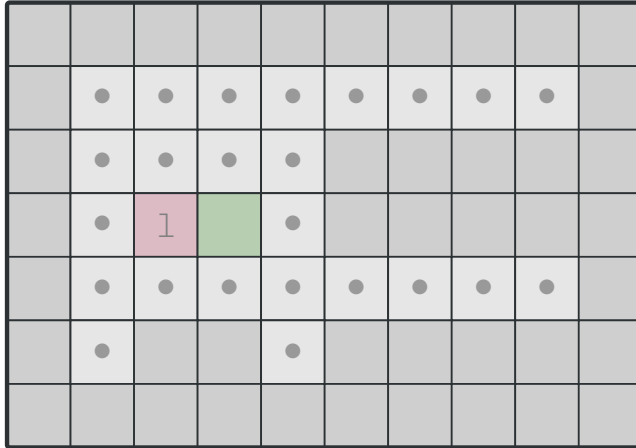
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

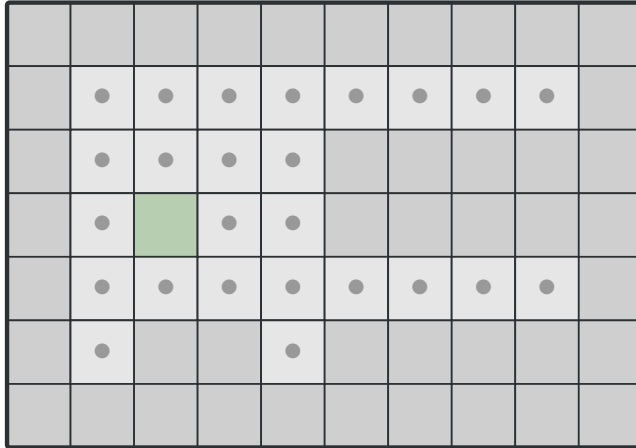
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

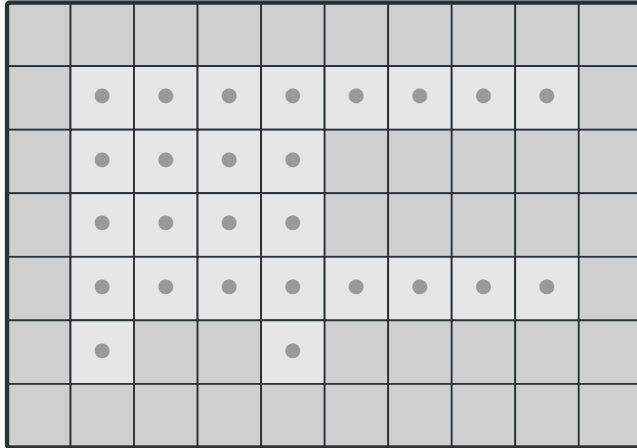
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE TIEFENSUCHE



TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



GRAPHEN DURCHSUCHEN: BREITENSUCHE

GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.

GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.

GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil:

GRAPHEN DURCHSUCHEN: BREITENSUCHE

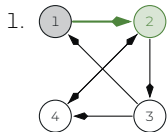
- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.

GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.

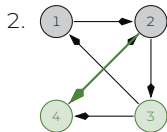
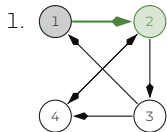
GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.



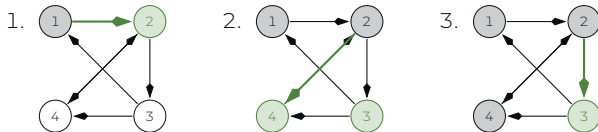
GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.



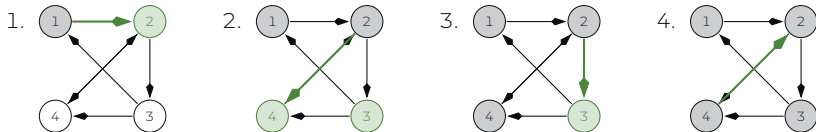
GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.



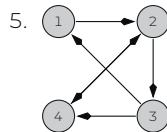
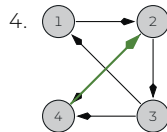
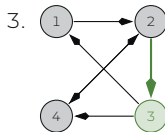
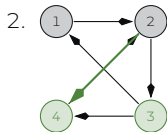
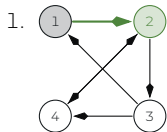
GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.



GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.



EIN BEISPIEL FÜR DIE BREITENSUCHE

EIN BEISPIEL FÜR DIE BREITENSUCHE

TRAVERSIERUNGSVARIANTEN

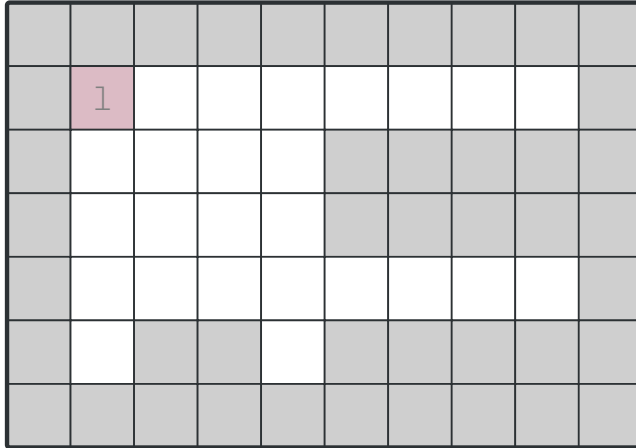
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE

TRAVERSIERUNGSVARIANTEN

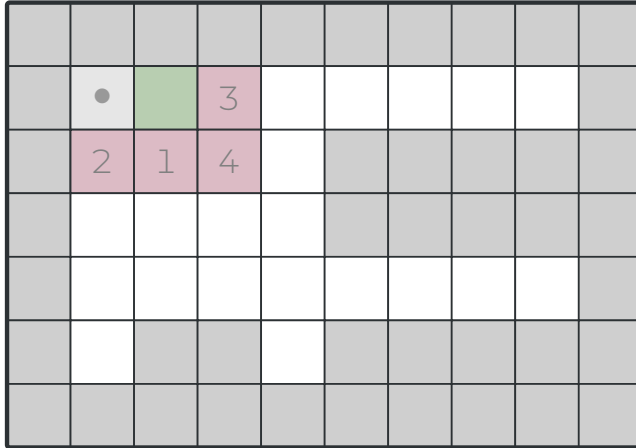
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

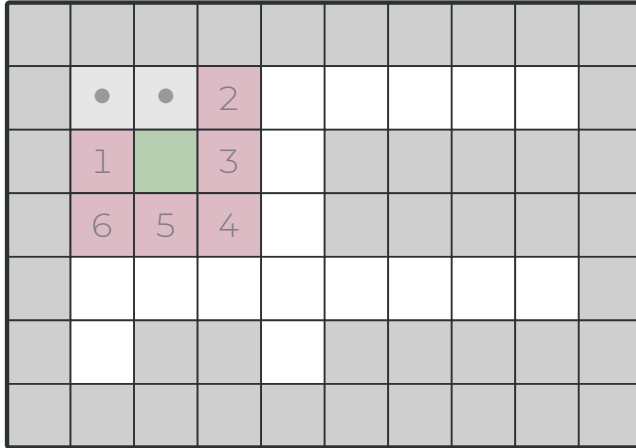
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

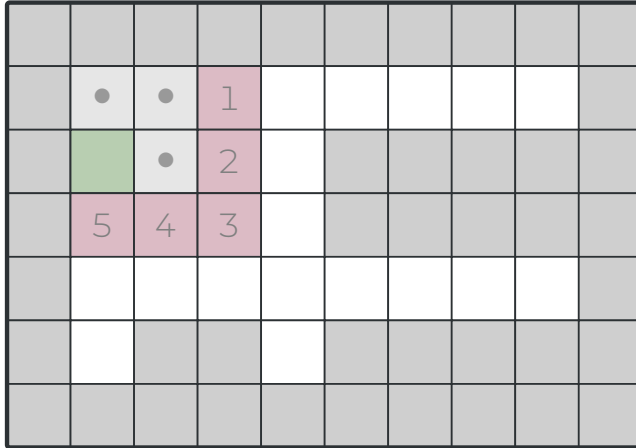
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

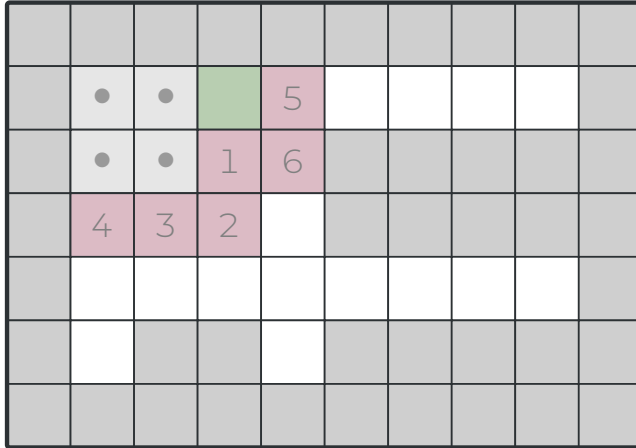
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

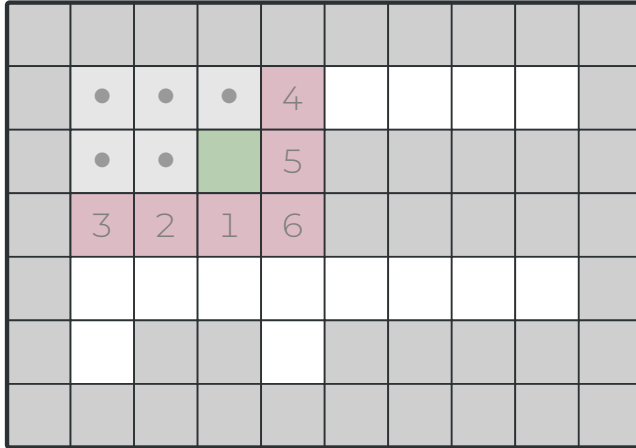
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schar

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE

	•	•	•	3					
	•	•	•	4					
	2	1		5					
		8	7	6					

TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE

	•	•	•	2					
	•	•	•	3					
	1		•	4					
	8	7	6	5					

TRAVERSIERUNGSVARIANTEN

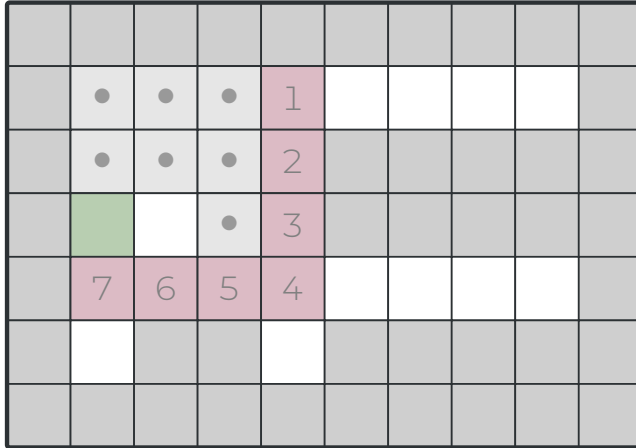
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

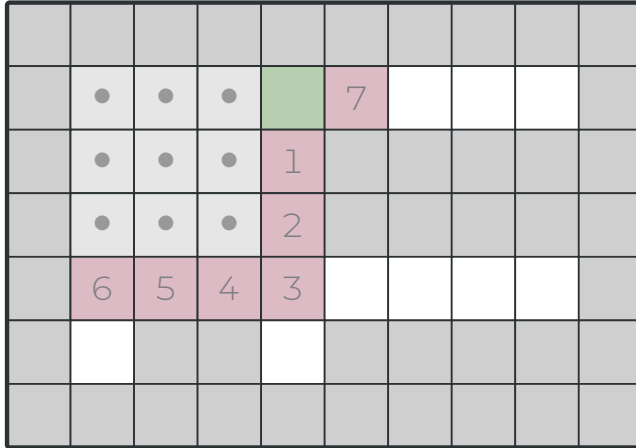
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

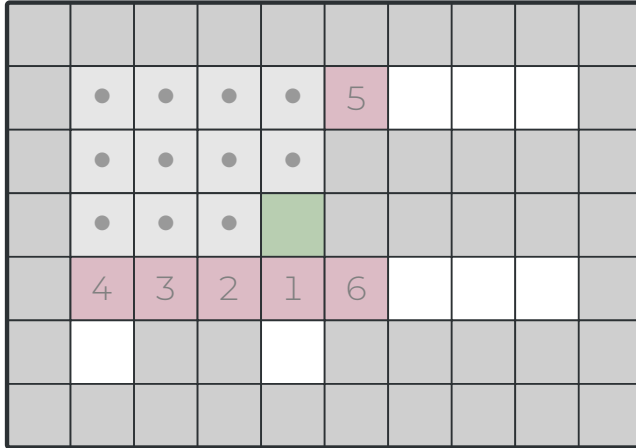
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

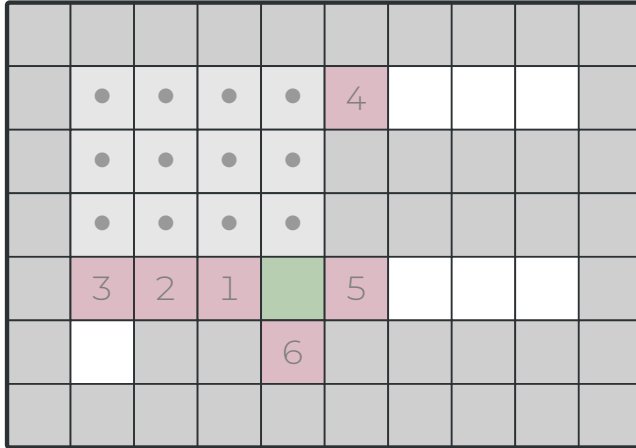
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

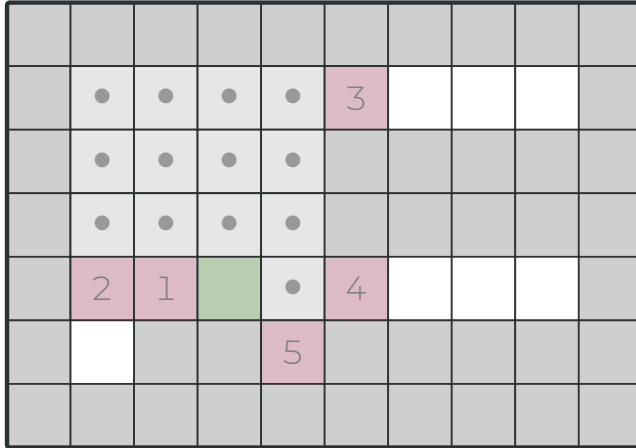
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

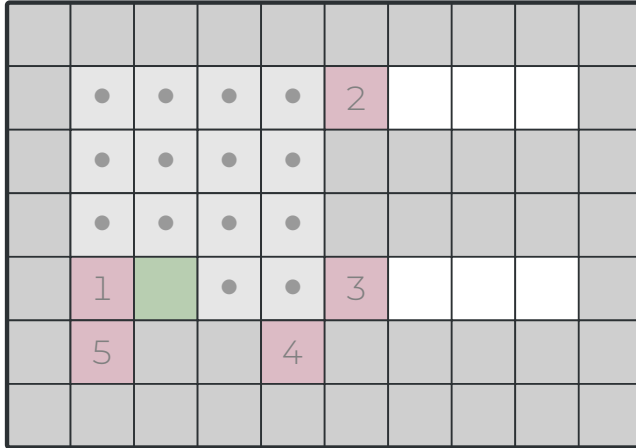
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

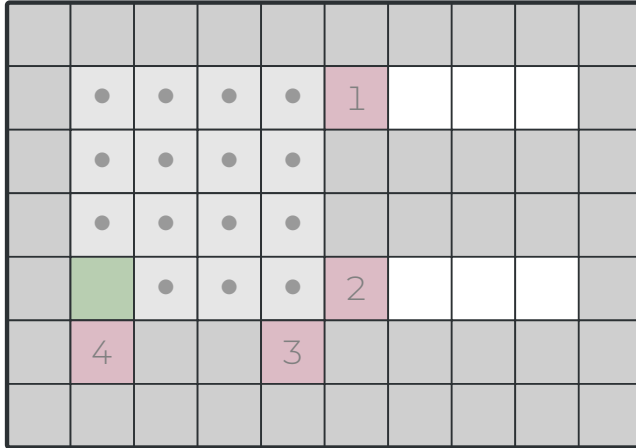
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

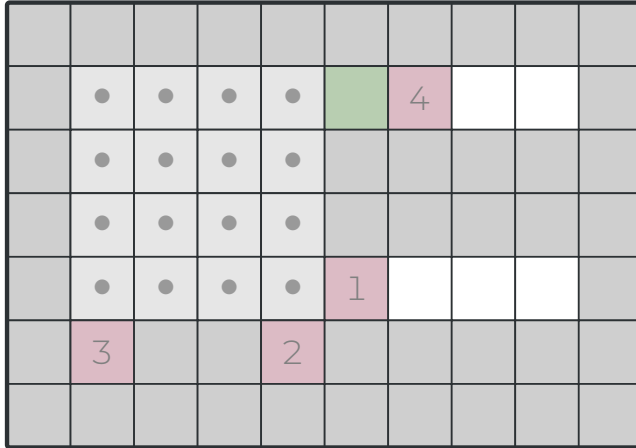
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

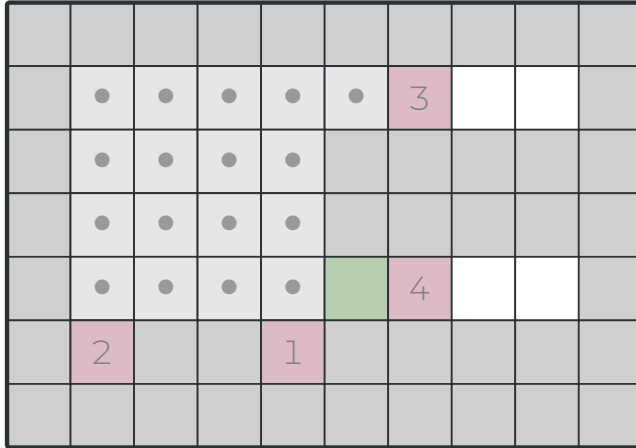
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

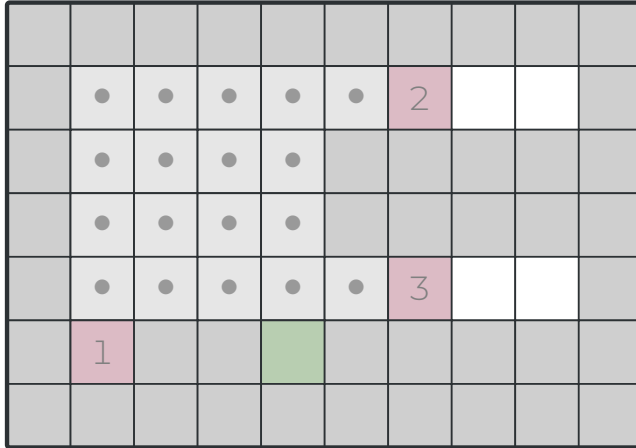
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schar

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

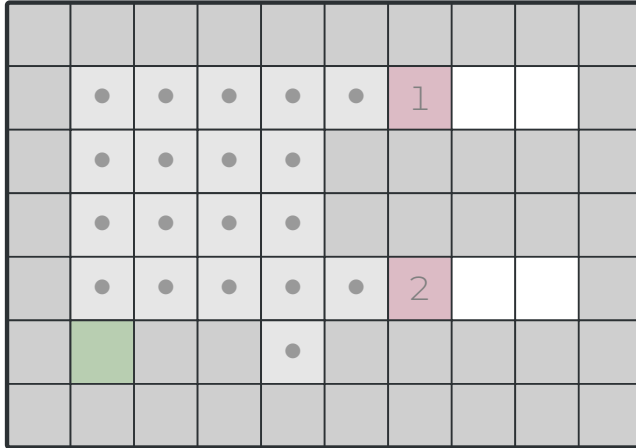
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

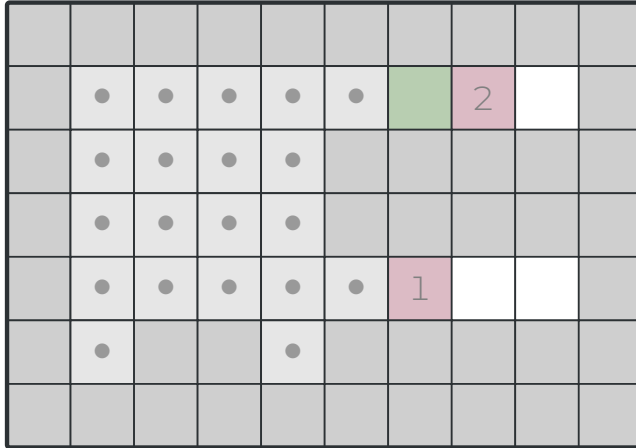
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

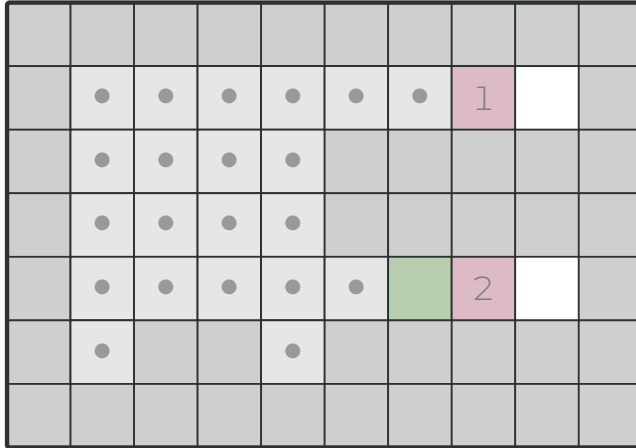
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

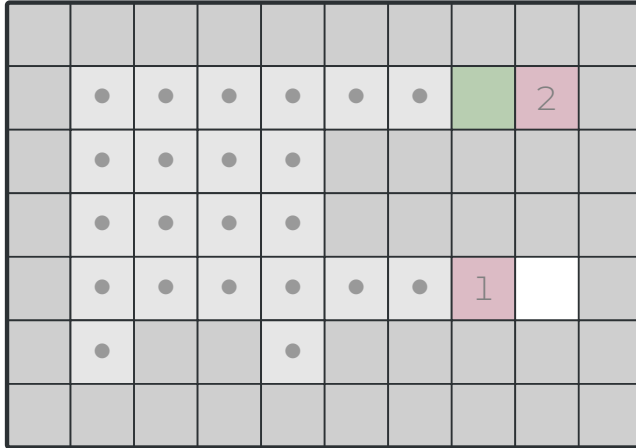
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schar

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

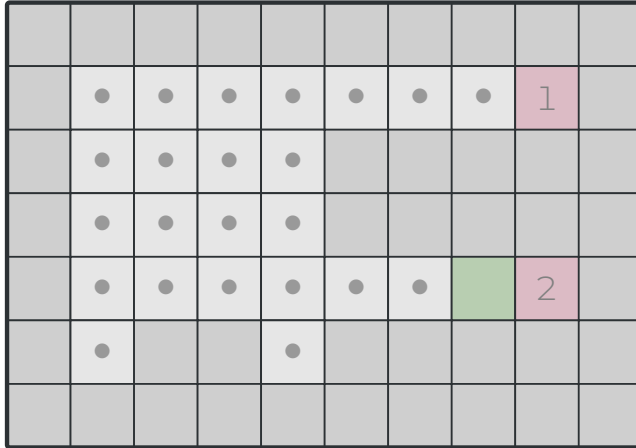
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

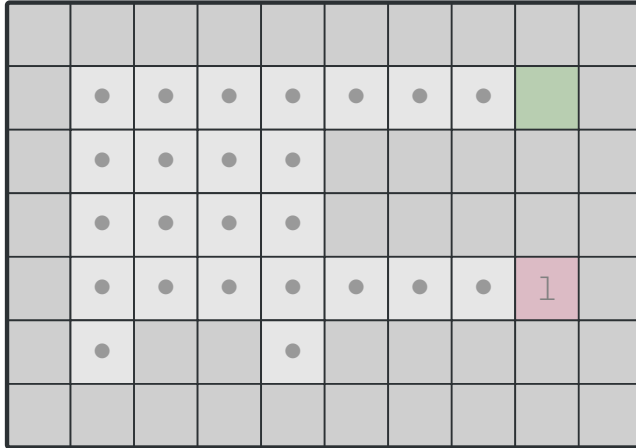
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

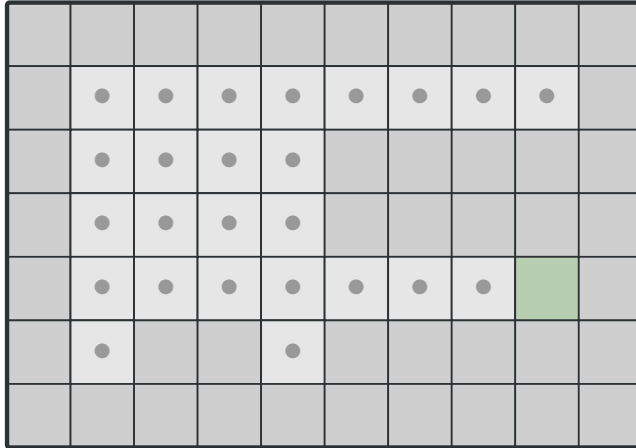
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

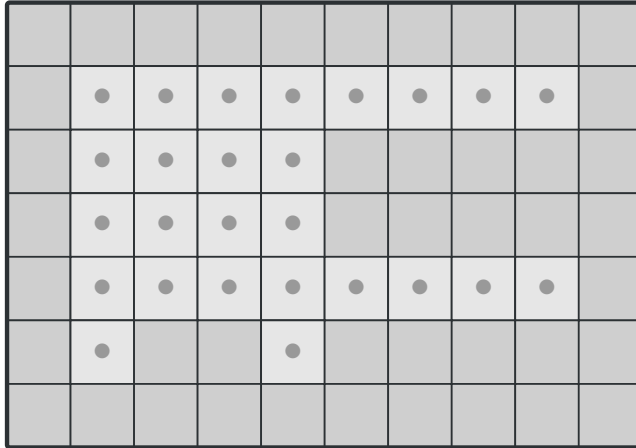
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



EIN BEISPIEL FÜR DIE BREITENSUCHE



TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Schier

13. Juli 2022
SP, Universität Ulm



ÜBUNGSaufGABE

Aufgabe 52: LinkedList - toString()

(4 Minuten)

Gegeben sei folgender Code. Es darf davon ausgegangen werden, dass `get(int)` korrekt implementiert ist.

Aufgabe 52: LinkedList - toString()

(4 Minuten)

Gegeben sei folgender Code. Es darf davon ausgegangen werden, dass `get(int)` korrekt implementiert ist. Schreiben Sie eine Methode `toString()`, die die Elemente in korrekter Reihenfolge ausgibt.

Aufgabe 52: LinkedList - toString()

(4 Minuten)

Gegeben sei folgender Code. Es darf davon ausgegangen werden, dass `get(int)` korrekt implementiert ist. Schreiben Sie eine Methode `toString()`, die die Elemente in korrekter Reihenfolge ausgibt.

```
public class LinkedList {
    public static class Element {
        int value; Element next;
        public Element(int _v, Element _n) { value = _v; next = _n; }
    }
    Element head;
    public LinkedList() { /* ... */ }
    public Element get(int index) { /* ... */ }
}
```

Aufgabe 52: LinkedList - toString()

(4 Minuten)

Gegeben sei folgender Code. Es darf davon ausgegangen werden, dass `get(int)` korrekt implementiert ist. Schreiben Sie eine Methode `toString()`, die die Elemente in korrekter Reihenfolge ausgibt.

```
public class LinkedList {
    public static class Element {
        int value; Element next;
        public Element(int _v, Element _n) { value = _v; next = _n; }
    }
    Element head;
    public LinkedList() { /* ... */ }
    public Element get(int index) { /* ... */ }
}
```

Beispiel: 13, 12, -4, 9, (Das endständige Komma ist erlaubt)

Lösung 52: LinkedList - toString()

Lösung 52: LinkedList - toString()

```
public String toString() {  
    Element current = head;  
    String strList = "";  
  
}
```

Lösung 52: LinkedList - toString()

```
public String toString() {  
    Element current = head;  
    String strList = "";  
    while(current != null) {  
  
    }  
  
}
```

Lösung 52: LinkedList - toString()

```
public String toString() {
    Element current = head;
    String strList = "";
    while(current != null) {
        strList += current.value + ", ";
        current = current.next;
    }
}
```

Lösung 52: LinkedList - toString()

```
public String toString() {  
    Element current = head;  
    String strList = "";  
    while(current != null) {  
        strList += current.value + ", ";  
        current = current.next;  
    }  
    return strList;  
}
```

Lösung 52: LinkedList - toString()

```
public String toString() {
    Element current = head;
    String strList = "";
    while(current != null) {
        strList += current.value + ", ";
        current = current.next;
    }
    return strList;
}
```

Ist kein Komma am Ende erwünscht, so genügt ein Test der Form `if(current.next != null)` für das Anfügen des Kommas.

ÜBUNGSaufGABE

Aufgabe 53: Eine Liste umdrehen

(6 Minuten)

Gegeben sei folgender Code:

```
public class Element {  
    public final int value;  
    public Element next;  
    public Element(int _v, Element _n) {  
        value = _v; next = _n;  
    }  
}
```

Aufgabe 53: Eine Liste umdrehen

(6 Minuten)

Gegeben sei folgender Code:

```
public class Element {
    public final int value;
    public Element next;
    public Element(int _v, Element _n) {
        value = _v; next = _n;
    }
}
```

Schreiben Sie eine Methode `Element reverse(Element head)`, welche die übergebene Liste umkehrt und den neuen Kopf zurück gibt. Gehen Sie auch mit Randfällen, wie einer leeren Liste um. Das Erzeugen einer neuen Liste ist nicht gestattet.

Lösung 53: Eine Liste umdrehen

Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {
```

```
}
```

Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {  
    Element reversedHead = null;  
    Element current = head;  
  
}
```

Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {  
    Element reversedHead = null;  
    Element current = head;  
    while (current != null) {  
  
    }  
  
}
```

Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {  
    Element reversedHead = null;  
    Element current = head;  
    while (current != null) {  
        Element next = current.next;  
  
        }  
  
}
```


Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {  
    Element reversedHead = null;  
    Element current = head;  
    while (current != null) {  
        Element next = current.next;  
        current.next = reversedHead;  
        reversedHead = current;  
        current = next;  
    }  
}
```

Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {  
    Element reversedHead = null;  
    Element current = head;  
    while (current != null) {  
        Element next = current.next;  
        current.next = reversedHead;  
        reversedHead = current;  
        current = next;  
    }  
}
```

Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {  
    Element reversedHead = null;  
    Element current = head;  
    while (current != null) {  
        Element next = current.next;  
        current.next = reversedHead;  
        reversedHead = current;  
        current = next;  
    }  
    return reversedHead;  
}
```

Lösung 53: Eine Liste umdrehen

```
Element reverse(Element head) {  
    Element reversedHead = null;  
    Element current = head;  
    while (current != null) {  
        Element next = current.next;  
        current.next = reversedHead;  
        reversedHead = current;  
        current = next;  
    }  
    return reversedHead;  
}
```

Doch was passiert hier? Betrachten wir dies im Detail...

Lösung 53: Eine Liste umdrehen

(Fortsetzung)

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

Lösung 53: Eine Liste umdrehen

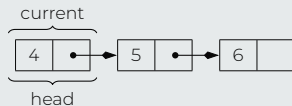
(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

Lösung 53: Eine Liste umdrehen

(Fortsetzung)

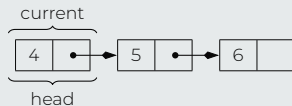
Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```


Lösung 53: Eine Liste umdrehen

(Fortsetzung)

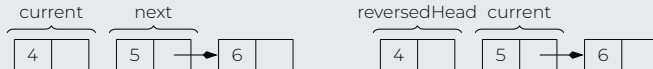
Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.



```

Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
    
```

1. Da `reversedHead = null`:



Lösung 53: Eine Liste umdrehen

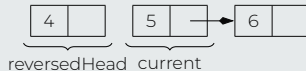
(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

Lösung 53: Eine Liste umdrehen

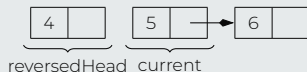
(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

Lösung 53: Eine Liste umdrehen

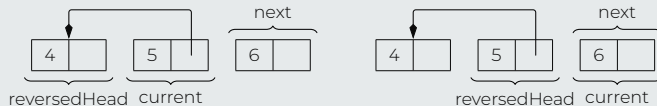
(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```

Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}

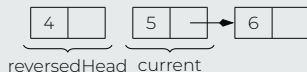
```

2. Nun zeigt `current.next` auf das vorherige Element:

Lösung 53: Eine Liste umdrehen

(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

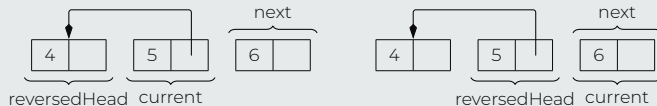


```

Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}

```

2. Nun zeigt `current.next` auf das vorherige Element:



Wie ein Schiffchen schieben wir also `reversedHead`, `current` und `next` nach rechts durch die Liste, um aus `current` \rightarrow `next` die Kante `current` \rightarrow `reversedHead` zu machen.

Lösung 53: Eine Liste umdrehen

(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```


Lösung 53: Eine Liste umdrehen

(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

Lösung 53: Eine Liste umdrehen

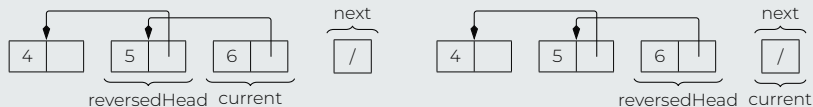
(Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.

```

Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}

```

3. `reversedHead` und `current` wandern durch die Liste:

ÜBUNGSaufGABE

Aufgabe 54: Theorie zu Stapel und Warteschlange (5 Minuten)

Bewerten (wahr/falsch) und Begründen Sie jeweils:

Aufgabe 54: Theorie zu Stapel und Warteschlange (5 Minuten)

Bewerten (wahr/falsch) und Begründen Sie jeweils:

1. Eine Warteschlange lässt sich auch dann implementieren, wenn nur die Datenstruktur *Stack* existiert.

Aufgabe 54: Theorie zu Stapel und Warteschlange (5 Minuten)

Bewerten (wahr/falsch) und Begründen Sie jeweils:

1. Eine Warteschlange lässt sich auch dann implementieren, wenn nur die Datenstruktur *Stack* existiert.
2. Stacks folgen dem FIFO-Prinzip.

Aufgabe 54: Theorie zu Stapel und Warteschlange (5 Minuten)

Bewerten (wahr/falsch) und Begründen Sie jeweils:

1. Eine Warteschlange lässt sich auch dann implementieren, wenn nur die Datenstruktur *Stack* existiert.
2. Stacks folgen dem FIFO-Prinzip.
3. Der Java-(Objekt-)Heap wird auch als (Datenstruktur) *Heap* verwaltet.

Lösung 54: Theorie zu Stapel und Warteschlange

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack:

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B .

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen.

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen. Ist B leer, so werden alle Elemente von A genommen und auf B abgelegt (und damit umgedreht).

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen. Ist B leer, so werden alle Elemente von A genommen und auf B abgelegt (und damit umgedreht).
2. Stacks sind FIFO:

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen. Ist B leer, so werden alle Elemente von A genommen und auf B abgelegt (und damit umgedreht).
2. Stacks sind FIFO: Dies ist falsch.

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen. Ist B leer, so werden alle Elemente von A genommen und auf B abgelegt (und damit umgedreht).
2. Stacks sind FIFO: Dies ist falsch. Stacks folgen dem LIFO (Last-In, First-Out) Prinzip (siehe: Stack).

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen. Ist B leer, so werden alle Elemente von A genommen und auf B abgelegt (und damit umgedreht).
2. Stacks sind FIFO: Dies ist falsch. Stacks folgen dem LIFO (Last-In, First-Out) Prinzip (siehe: Stack).
3. Java-Heap ist Heap:

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen. Ist B leer, so werden alle Elemente von A genommen und auf B abgelegt (und damit umgedreht).
2. Stacks sind FIFO: Dies ist falsch. Stacks folgen dem LIFO (Last-In, First-Out) Prinzip (siehe: Stack).
3. Java-Heap ist Heap: Dies ist falsch, der Heap auf dem (komplexe) Daten abgelegt werden,

Lösung 54: Theorie zu Stapel und Warteschlange

1. Queue durch Stack: Dies ist wahr. Hierzu benötigt man zwei Stacks, A und B . Neue Elemente werden auf A abgelegt und von B genommen. Ist B leer, so werden alle Elemente von A genommen und auf B abgelegt (und damit umgedreht).
2. Stacks sind FIFO: Dies ist falsch. Stacks folgen dem LIFO (Last-In, First-Out) Prinzip (siehe: Stack).
3. Java-Heap ist Heap: Dies ist falsch, der Heap auf dem (komplexe) Daten abgelegt werden, hat nur denselben Namen (siehe: `stackoverflow`).

ÜBUNGSaufGABE

Aufgabe 55: Traversierung eines Baumes

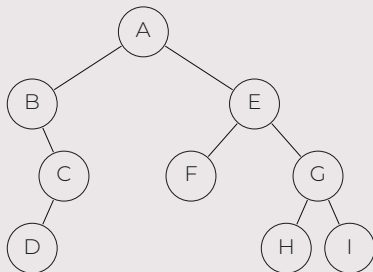
(4 Minuten)

Gegeben sei der folgende Binärbaum. Geben Sie sowohl die Besuchsreihenfolge des Level-, als auch des In- und des Post-Order Durchlaufs an.

Aufgabe 55: Traversierung eines Baumes

(4 Minuten)

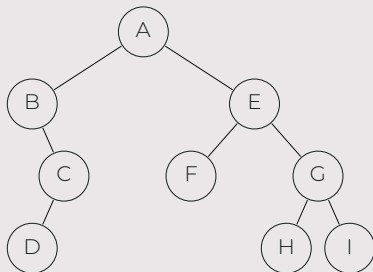
Gegeben sei der folgende Binärbaum. Geben Sie sowohl die Besuchsreihenfolge des Level-, als auch des In- und des Post-Order Durchlaufs an.



Aufgabe 55: Traversierung eines Baumes

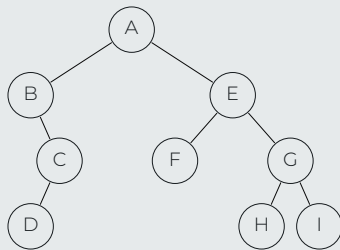
(4 Minuten)

Gegeben sei der folgende Binärbaum. Geben Sie sowohl die Besuchsreihenfolge des Level-, als auch des In- und des Post-Order Durchlaufs an.

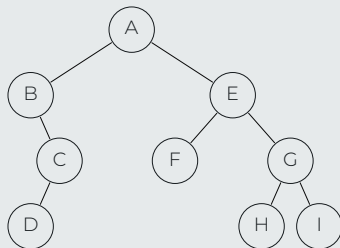


Hinweis, die Pre-Order Traversierung wäre: A, B, C, D, E, F, G, H, I.

Lösung 55: Traversierung eines Baumes

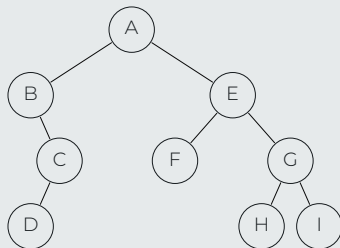


Lösung 55: Traversierung eines Baumes



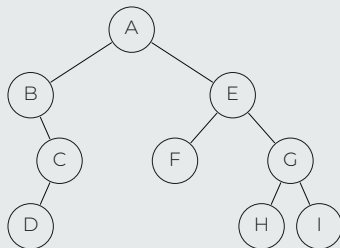
1. Breitendurchlauf (Level-Order): *A, B, E, C, F, G, D, H, I.*

Lösung 55: Traversierung eines Baumes



1. Breitendurchlauf (Level-Order): $A, B, E, C, F, G, D, H, I$.
2. In-Order: $B, D, C, A, F, E, H, G, I$.

Lösung 55: Traversierung eines Baumes



1. Breitendurchlauf (Level-Order): $A, B, E, C, F, G, D, H, I$.
2. In-Order: $B, D, C, A, F, E, H, G, I$.
3. Post-Order: $D, C, B, F, H, I, G, E, A$.

ÜBUNGSaufGABE

Aufgabe 56: Baum durch Traversierung

(4 Minuten)

Aufgabe 56: Baum durch Traversierung

(4 Minuten)

Gegeben seien die beiden folgenden Traversierungen des selben Binärbaums:

Aufgabe 56: Baum durch Traversierung

(4 Minuten)

Gegeben seien die beiden folgenden Traversierungen des selben Binärbaums:

- Pre-Order: 9, 7, 14, 12, 0, 11, 8
- In-Order: 14, 12, 7, 0, 9, 11, 8

Aufgabe 56: Baum durch Traversierung

(4 Minuten)

Gegeben seien die beiden folgenden Traversierungen des selben Binärbaums:

- Pre-Order: 9, 7, 14, 12, 0, 11, 8
- In-Order: 14, 12, 7, 0, 9, 11, 8

Stellen Sie einen Binärbaum grafisch dar, welcher diese Ausgaben erzeugt. Achten Sie darauf deutlich zu zeichnen, ob es sich jeweils um linke oder rechte Kindknoten handelt.

Lösung 56: Traversierung eines Baumes

Lösung 56: Traversierung eines Baumes

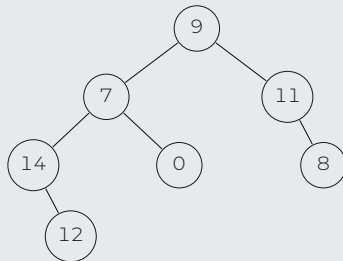
1. Pre-Order: 9, 7, 14, 12, 0, 11, 8

Lösung 56: Traversierung eines Baumes

1. Pre-Order: 9, 7, 14, 12, 0, 11, 8
2. In-Order: 14, 12, 7, 0, 9, 11, 8

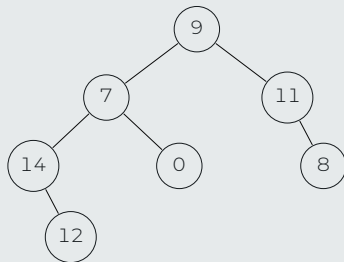
Lösung 56: Traversierung eines Baumes

1. Pre-Order: 9, 7, 14, 12, 0, 11, 8
2. In-Order: 14, 12, 7, 0, 9, 11, 8



Lösung 56: Traversierung eines Baumes

1. Pre-Order: 9, 7, 14, 12, 0, 11, 8
2. In-Order: 14, 12, 7, 0, 9, 11, 8



Wir sehen, dass 9 die Wurzel ist (erstes Element in Pre-Order), die Suche der 9 in In-Order liefert die Aufteilung auf links und rechts...

ÜBUNGSAUFGABE

Aufgabe 57: Traversierung eines Baumes - Java (4 Minuten)

Die folgende Klasse definiere die Knoten eines Binärbaums. Schreiben Sie eine rekursive Methode `inorder(Node n)`, die eine In-Order Traversierung ab dem Knoten n vornimmt.

Aufgabe 57: Traversierung eines Baumes - Java (4 Minuten)

Die folgende Klasse definiere die Knoten eines Binärbaums. Schreiben Sie eine rekursive Methode `inorder(Node n)`, die eine In-Order Traversierung ab dem Knoten n vornimmt. Der Wert des Knoten soll (jeweils) durch `System.out.println` ausgegeben werden.

Aufgabe 57: Traversierung eines Baumes - Java

(4 Minuten)

Die folgende Klasse definiere die Knoten eines Binärbaums. Schreiben Sie eine rekursive Methode `inorder(Node n)`, die eine In-Order Traversierung ab dem Knoten n vornimmt. Der Wert des Knoten soll (jeweils) durch `System.out.println` ausgegeben werden.

```
public class Node {  
    public int value;  
    public Node left, right;  
}
```


Lösung 57: Traversierung eines Baumes - Java

```
void inorder(Node n) {  
    if (n == null) return;  
  
    inorder(n.left);  
    System.out.println(n.value);  
    inorder(n.right);  
}
```

ÜBUNGSAUFGABE

Aufgabe 58: Traversierung eines Baumes, 2 - Java (6 Minuten)

`Node` definiere die Knoten eines Binärbaums, `Stack` sei eine korrekt implementierte Stack-Klasse (deren Konstruktor einen leeren Stack initialisiert). Schreiben Sie eine **iterative** Methode `preorderIt(Node n)`, die eine Pre-Order Traversierung ab dem Knoten n vornimmt. Diese soll dabei den linken vor dem rechten Teilbaum bearbeiten.

Aufgabe 58: Traversierung eines Baumes, 2 - Java (6 Minuten)

`Node` definiere die Knoten eines Binärbaums, `Stack` sei eine korrekt implementierte Stack-Klasse (deren Konstruktor einen leeren Stack initialisiert). Schreiben Sie eine **iterative** Methode `preorderIt(Node n)`, die eine Pre-Order Traversierung ab dem Knoten n vornimmt. Diese soll dabei den linken vor dem rechten Teilbaum bearbeiten. Der Wert des Knoten soll (jeweils) durch `System.out.println` ausgegeben werden.

Aufgabe 58: Traversierung eines Baumes, 2 - Java (6 Minuten)

`Node` definiere die Knoten eines Binärbaums, `Stack` sei eine korrekt implementierte Stack-Klasse (deren Konstruktor einen leeren Stack initialisiert). Schreiben Sie eine **iterative** Methode `preorderIt(Node n)`, die eine Pre-Order Traversierung ab dem Knoten n vornimmt. Diese soll dabei den linken vor dem rechten Teilbaum bearbeiten. Der Wert des Knoten soll (jeweils) durch `System.out.println` ausgegeben werden.

```
public class Node {
    public int value;
    public Node left;
    public Node right;
}
```

```
public class Stack {
    public Stack() { /*...*/ }
    public boolean isEmpty() { /*...*/ }
    public void push(Node n) { /*...*/ }
    public Node pop() { /*...*/ }
}
```


Lösung 58: Traversierung eines Baumes, 2 - Java

Lösung 58: Traversierung eines Baumes, 2 - Java

```
void preorderIt(Node n) {
```

```
}
```


Lösung 58: Traversierung eines Baumes, 2 - Java

```
void preorderIt(Node n) {  
    Stack s = new Stack();  
    s.push(n);  
    while(!s.isEmpty()) {  
  
        }  
    }  
}
```

Lösung 58: Traversierung eines Baumes, 2 - Java

```
void preorderIt(Node n) {  
    Stack s = new Stack();  
    s.push(n);  
    while(!s.isEmpty()) {  
        Node current = s.pop();  
  
        }  
}
```


Lösung 58: Traversierung eines Baumes, 2 - Java

```
void preorderIt(Node n) {  
    Stack s = new Stack();  
    s.push(n);  
    while(!s.isEmpty()) {  
        Node current = s.pop();  
        if(current == null)  
            continue;  
    }  
}
```

Lösung 58: Traversierung eines Baumes, 2 - Java

```
void preorderIt(Node n) {  
    Stack s = new Stack();  
    s.push(n);  
    while(!s.isEmpty()) {  
        Node current = s.pop();  
        if(current == null)  
            continue;  
        System.out.println(current.value);  
    }  
}
```

Lösung 58: Traversierung eines Baumes, 2 - Java

```
void preorderIt(Node n) {  
    Stack s = new Stack();  
    s.push(n);  
    while(!s.isEmpty()) {  
        Node current = s.pop();  
        if(current == null)  
            continue;  
        System.out.println(current.value);  
        s.push(current.right);  
    }  
}
```

Lösung 58: Traversierung eines Baumes, 2 - Java

```
void preorderIt(Node n) {  
    Stack s = new Stack();  
    s.push(n);  
    while(!s.isEmpty()) {  
        Node current = s.pop();  
        if(current == null)  
            continue;  
        System.out.println(current.value);  
        s.push(current.right);  
        s.push(current.left); // erst links  
    }  
}
```

ÜBUNGSaufGABE

Aufgabe 59: Breitensuche - Pseudocode

(6 Minuten)

Schreiben Sie Pseudocode, der den folgenden Vertrag durch eine Breitensuche erfüllt:

Aufgabe 59: Breitensuche - Pseudocode

(6 Minuten)

Schreiben Sie Pseudocode, der den folgenden Vertrag durch eine Breitensuche erfüllt:

In: Adjazenzmatrix a für $n \in \mathbb{N}$ Knoten mit $n > 0$.

In: Startknoten $s \in \{0, \dots, n - 1\}$ und gesuchter Knoten $g \in \{0, \dots, n - 1\}$.

Out: „Ja“, wenn g von s aus in a erreichbar ist. Sonst „Nein“.

Aufgabe 59: Breitensuche - Pseudocode

(6 Minuten)

Schreiben Sie Pseudocode, der den folgenden Vertrag durch eine Breitensuche erfüllt:

In: Adjazenzmatrix a für $n \in \mathbb{N}$ Knoten mit $n > 0$.

In: Startknoten $s \in \{0, \dots, n - 1\}$ und gesuchter Knoten $g \in \{0, \dots, n - 1\}$.

Out: „Ja“, wenn g von s aus in a erreichbar ist. Sonst „Nein“.

Sie können Mengen, sowie Warteschlangen verwenden:

- `Queue.create()` erzeugt eine neue Warteschlange.
- `enqueue(<Wert>)` fügt einen Wert hinten an.
- `dequeue()` liefert und entfernt den vordersten Wert.
- `size()` liefert Ihnen die Größe der Warteschlange.

Lösung 59: Breitensuche - Pseudocode

Lösung 59: Breitensuche - Pseudocode

Lösung 59: Breitensuche - Pseudocode

```
1 queue ← Queue.create();  
2 queue.enqueue(s);           // Startknoten hinzufügen
```

Lösung 59: Breitensuche - Pseudocode

```
1 queue ← Queue.create();  
2 queue.enqueue(s); // Startknoten hinzufügen  
3 visited ← {s}; // Menge markiert bereits beachtete
```

Lösung 59: Breitensuche - Pseudocode

```
1 queue ← Queue.create();  
2 queue.enqueue(s); // Startknoten hinzufügen  
3 visited ← {s}; // Menge markiert bereits beachtete  
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
```



Lösung 59: Breitensuche - Pseudocode

```
1 queue ← Queue.create();
2 queue.enqueue(s); // Startknoten hinzufügen
3 visited ← {s}; // Menge markiert bereits beachtete
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
5   current ← queue.dequeue(); // Betrachte nächsten Knoten
```

Lösung 59: Breitensuche - Pseudocode

```
1 queue ← Queue.create();
2 queue.enqueue(s); // Startknoten hinzufügen
3 visited ← {s}; // Menge markiert bereits beachtete
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
5     current ← queue.dequeue(); // Betrachte nächsten Knoten
6     if current = g then return „Ja“; // Ist er das Ziel?
```


Lösung 59: Breitensuche - Pseudocode

```
1 queue ← Queue.create();
2 queue.enqueue(s); // Startknoten hinzufügen
3 visited ← {s}; // Menge markiert bereits beachtete
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
5     current ← queue.dequeue(); // Betrachte nächsten Knoten
6     if current = g then return „Ja“; // Ist er das Ziel?
    // Durchsuche Adjazenzmatrix nach allen erreichbaren Knoten
7     for i = 0 to n - 1 do
```

Lösung 59: Breitensuche - Pseudocode

```
1 queue ← Queue.create();
2 queue.enqueue(s); // Startknoten hinzufügen
3 visited ← {s}; // Menge markiert bereits beachtete
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
5     current ← queue.dequeue(); // Betrachte nächsten Knoten
6     if current = g then return „Ja“; // Ist er das Ziel?
    // Durchsuche Adjazenzmatrix nach allen erreichbaren Knoten
7     for i = 0 to n - 1 do
8         if acurrent,i > 0 then // Es gibt eine Kante von current nach i
```

Lösung 59: Breitensuche - Pseudocode

```

1 queue ← Queue.create();
2 queue.enqueue(s); // Startknoten hinzufügen
3 visited ← {s}; // Menge markiert bereits beachtete
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
5     current ← queue.dequeue(); // Betrachte nächsten Knoten
6     if current = g then return „Ja“; // Ist er das Ziel?
    // Durchsuche Adjazenzmatrix nach allen erreichbaren Knoten
7     for i = 0 to n - 1 do
8         if acurrent,i > 0 then // Es gibt eine Kante von current nach i
9             if i ∉ visited then // Noch nicht besucht

```

Lösung 59: Breitensuche - Pseudocode

```

1 queue ← Queue.create();
2 queue.enqueue(s); // Startknoten hinzufügen
3 visited ← {s}; // Menge markiert bereits beachtete
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
5     current ← queue.dequeue(); // Betrachte nächsten Knoten
6     if current = g then return „Ja“; // Ist er das Ziel?
    // Durchsuche Adjazenzmatrix nach allen erreichbaren Knoten
7     for i = 0 to n - 1 do
8         if acurrent,i > 0 then // Es gibt eine Kante von current nach i
9             if i ∉ visited then // Noch nicht besucht
10                queue.enqueue(i);
11                visited ← visited ∪ {i}; // Füge zu besuchten hinzu.

```

Lösung 59: Breitensuche - Pseudocode

```

1 queue ← Queue.create();
2 queue.enqueue(s); // Startknoten hinzufügen
3 visited ← {s}; // Menge markiert bereits beachtete
4 while queue.size() > 0 do // Es gibt noch besuchbare Punkte
5     current ← queue.dequeue(); // Betrachte nächsten Knoten
6     if current = g then return „Ja“; // Ist er das Ziel?
    // Durchsuche Adjazenzmatrix nach allen erreichbaren Knoten
7     for i = 0 to n - 1 do
8         if acurrent,i > 0 then // Es gibt eine Kante von current nach i
9             if i ∉ visited then // Noch nicht besucht
10                queue.enqueue(i);
11                visited ← visited ∪ {i}; // Füge zu besuchten hinzu.
12 return „Nein“;

```

ÜBUNGSAUFGABE

Aufgabe 60: Breitensuche - Java

(6 Minuten)

Übersetzen Sie den Pseudocode (frei nach Aufgabe 59) in Java Code.

Aufgabe 60: Breitensuche - Java

(6 Minuten)

Übersetzen Sie den Pseudocode (frei nach Aufgabe 59) in Java Code.

```
1 queue ← new empty Queue;
2 queue.enqueue(s);
3 visited ← {s};
4 Solange queue.size() > 0 tue:
5   | cur ← queue.dequeue();
6   | Wenn cur = g dann: Rückgabe: wahr;
7   | Für i = 0 bis n - 1 tue:
8     | Wenn acur,i > 0 & i ∉ visited dann:
9       | queue.enqueue(i);
10      | visited ← visited ∪ {i};
11 Rückgabe: falsch;
```


Aufgabe 60: Breitensuche - Java

(6 Minuten)

Übersetzen Sie den Pseudocode (frei nach Aufgabe 59) in Java Code.

Gegeben seien die folgenden, korrekt implementierten Klassen:

```
1 queue ← new empty Queue;
2 queue.enqueue(s);
3 visited ← {s};
4 Solange queue.size() > 0 tue:
5     cur ← queue.dequeue();
6     Wenn cur = g dann: Rückgabe: wahr;
7     Für i = 0 bis n - 1 tue:
8         Wenn  $a_{cur,i} > 0$  &  $i \notin visited$  dann:
9             queue.enqueue(i);
10            visited ← visited ∪ {i};
11 Rückgabe: falsch;
```

Aufgabe 60: Breitensuche - Java

(6 Minuten)

Übersetzen Sie den Pseudocode (frei nach Aufgabe 59) in Java Code.

```
1 queue ← new empty Queue;
2 queue.enqueue(s);
3 visited ← {s};
4 Solange queue.size() > 0 tue:
5     cur ← queue.dequeue();
6     Wenn cur = g dann: Rückgabe: wahr;
7     Für i = 0 bis n - 1 tue:
8         Wenn acur,i > 0 & i ∉ visited dann:
9             queue.enqueue(i);
10            visited ← visited ∪ {i};
11 Rückgabe: falsch;
```

Gegeben seien die folgenden, korrekt implementierten Klassen:

```
class Set {
    void add(int value) {...}
    boolean contains(int value) {...}
}
class Queue {
    void enqueue(int value) {...}
    int dequeue() {...}
    int size() {...}
}
```

Schreiben Sie eine Java-Methode `boolean bfs(int[][] a, int s, int g)`. Sie dürfen annehmen, dass a korrekt ist und s , sowie g , gültig sind.

Lösung 60: Breitensuche - Java

Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {  
    Queue queue = new Queue(); queue.enqueue(s);  
    Set visited = new Set(); visited.add(s);  
    while(queue.size() > 0) {  
  
    }  
}
```


Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {  
    Queue queue = new Queue(); queue.enqueue(s);  
    Set visited = new Set(); visited.add(s);  
    while(queue.size() > 0) {  
        int cur = queue.dequeue();  
  
        }  
}
```

Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {  
    Queue queue = new Queue(); queue.enqueue(s);  
    Set visited = new Set(); visited.add(s);  
    while(queue.size() > 0) {  
        int cur = queue.dequeue();  
        if(cur == g) return true;  
  
    }  
}
```

Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {  
    Queue queue = new Queue(); queue.enqueue(s);  
    Set visited = new Set(); visited.add(s);  
    while(queue.size() > 0) {  
        int cur = queue.dequeue();  
        if(cur == g) return true;  
        for(int i = 0; i < a.length; i++) { // oder i <= a.length - 1  
  
        }  
    }  
}
```

Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {
    Queue queue = new Queue(); queue.enqueue(s);
    Set visited = new Set(); visited.add(s);
    while(queue.size() > 0) {
        int cur = queue.dequeue();
        if(cur == g) return true;
        for(int i = 0; i < a.length; i++) { // oder i <= a.length - 1
            if(a[cur][i] > 0 && !visited.contains(i)) {

                }
            }
        }
    }
}
```

Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {
    Queue queue = new Queue(); queue.enqueue(s);
    Set visited = new Set(); visited.add(s);
    while(queue.size() > 0) {
        int cur = queue.dequeue();
        if(cur == g) return true;
        for(int i = 0; i < a.length; i++) { // oder i <= a.length - 1
            if(a[cur][i] > 0 && !visited.contains(i)) {
                queue.enqueue(i);
            }
        }
    }
}
```

Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {
    Queue queue = new Queue(); queue.enqueue(s);
    Set visited = new Set(); visited.add(s);
    while(queue.size() > 0) {
        int cur = queue.dequeue();
        if(cur == g) return true;
        for(int i = 0; i < a.length; i++) { // oder i <= a.length - 1
            if(a[cur][i] > 0 && !visited.contains(i)) {
                queue.enqueue(i);
                visited.add(i);
            }
        }
    }
}
```

Lösung 60: Breitensuche - Java

```
boolean bfs(int[][] a, int s, int g) {
    Queue queue = new Queue(); queue.enqueue(s);
    Set visited = new Set(); visited.add(s);
    while(queue.size() > 0) {
        int cur = queue.dequeue();
        if(cur == g) return true;
        for(int i = 0; i < a.length; i++) { // oder i <= a.length - 1
            if(a[cur][i] > 0 && !visited.contains(i)) {
                queue.enqueue(i);
                visited.add(i);
            }
        }
    }
    return false;
}
```

Java-Advanced



DAS KONZEPT DER VERERBUNG

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
 - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
 - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
 - als Erweiterung: die Methoden werden modifiziert.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
 - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
 - als Erweiterung: die Methoden werden modifiziert.
- Häufig wird das Konzept der *Erweiterung* verfolgt.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
 - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
 - als Erweiterung: die Methoden werden modifiziert.
- Häufig wird das Konzept der *Erweiterung* verfolgt.
- Zwischen vererbenden Klassen gilt eine „is-a“ Relation.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
 - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
 - als Erweiterung: die Methoden werden modifiziert.
- Häufig wird das Konzept der *Erweiterung* verfolgt.
- Zwischen vererbenden Klassen gilt eine „is-a“ Relation. So gilt beispielsweise „Student *is-a* Mensch *is-a* Lebewesen“.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **extends** realisiert.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **extends** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **extends** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.
- In Java kann eine Klasse nur von maximal einer anderen Erben.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **extends** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.
- In Java kann eine Klasse nur von maximal einer anderen Erben.
- Auf Attribute der Elternklasse, die **private** sind, können wir in erbenden Kindklassen nicht zugreifen.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **extends** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.
- In Java kann eine Klasse nur von maximal einer anderen Erben.
- Auf Attribute der Elternklasse, die **private** sind, können wir in erbenden Kindklassen nicht zugreifen.
- Der Konstruktor sowie Methoden der Superklasse können (soweit sichtbar) durch **super** erreicht werden.

ÜBUNGSAUFGABE

Aufgabe 61: Fehler finden - Vererbung

(5 Minuten)

Erklären Sie drei Kompilierzeit-Fehler. Die Klassen befinden sich jeweils in Dateien mit dem korrekten Namen und in dem *selben* Verzeichnis:

Aufgabe 61: Fehler finden - Vererbung

(5 Minuten)

Erklären Sie drei Kompilierzeit-Fehler. Die Klassen befinden sich jeweils in Dateien mit dem korrekten Namen und in dem *selben* Verzeichnis:

```
public class A {
    A(int foo) { System.out.println("Called_with_" + foo); }
    public static int have() { return 42; }
    protected boolean fun() { return true; }
    private String at() { return "Fridays"; }
    public String at(String place) { return place; }
}

public class B extends A {
    public static int have() { return 0; }
    private boolean fun() { return false; }
    public String at() { return "my_place"; }
    protected String at(String place) { return super.at(); }
}
```


Lösung 61: Fehler finden - Vererbung

Dies sind alle Fehler, welche vom Java-Compiler erkannt werden:

1. B benötigt einen Konstruktor, da es für die Superklasse nur `A(int)` gibt.

Lösung 61: Fehler finden - Vererbung

Dies sind alle Fehler, welche vom Java-Compiler erkannt werden:

1. B benötigt einen Konstruktor, da es für die Superklasse nur `A(int)` gibt. Zum Beispiel: `„B() { super(0); }“`.

Hinweis: `B::at()` ist kein Problem, da es die Sichtbarkeit von **private** auf **public** erhöht („erweitert“).

Lösung 61: Fehler finden - Vererbung

Dies sind alle Fehler, welche vom Java-Compiler erkannt werden:

1. B benötigt einen Konstruktor, da es für die Superklasse nur `A(int)` gibt. Zum Beispiel: „`B() { super(0); }`“.
2. `B::fun()` ist illegal, es schränkt die Sichtbarkeit von `A::fun()` von **protected** auf **private** ein. Dies widerspricht dem Erweiterungskonzept.

Hinweis: `B::at()` ist kein Problem, da es die Sichtbarkeit von **private** auf **public** erhöht („erweitert“).

Lösung 61: Fehler finden - Vererbung

Dies sind alle Fehler, welche vom Java-Compiler erkannt werden:

1. B benötigt einen Konstruktor, da es für die Superklasse nur `A(int)` gibt. Zum Beispiel: „`B() { super(0); }`“.
2. `B::fun()` ist illegal, es schränkt die Sichtbarkeit von `A::fun()` von **protected** auf **private** ein. Dies widerspricht dem Erweiterungskonzept.
3. `B::at(String)` ist aus dem selben Grund illegal. Es schränkt die Sichtbarkeit von **public** auf **protected** ein!

Hinweis: `B::at()` ist kein Problem, da es die Sichtbarkeit von **private** auf **public** erhöht („erweitert“).

Lösung 61: Fehler finden - Vererbung

Dies sind alle Fehler, welche vom Java-Compiler erkannt werden:

1. B benötigt einen Konstruktor, da es für die Superklasse nur `A(int)` gibt. Zum Beispiel: „`B() { super(0); }`“.
2. `B::fun()` ist illegal, es schränkt die Sichtbarkeit von `A::fun()` von **protected** auf **private** ein. Dies widerspricht dem Erweiterungskonzept.
3. `B::at(String)` ist aus dem selben Grund illegal. Es schränkt die Sichtbarkeit von **public** auf **protected** ein!
4. Ebenso darf `B::at(String)` die Methode `super.at()`; gar nicht aufrufen, da `A::at()` die Sichtbarkeit **private** besitzt.

Hinweis: `B::at()` ist kein Problem, da es die Sichtbarkeit von **private** auf **public** erhöht („erweitert“).

EIN BEISPIEL IN JAVA

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

```
class Lebewesen {  
    int alter;  
    float grosse;  
  
    Lebewesen(int a, float g) {  
        alter = a;  
        grosse = g;  
    }  
}
```

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

```
class Lebewesen {  
    int alter;  
    float grosse;  
  
    Lebewesen(int a, float g) {  
        alter = a;  
        grosse = g;  
    }  
}
```

```
class Mensch extends Lebewesen {  
    String name;  
  
    Mensch(int a, float g, String n) {  
        super(a, g);  
        name = n;  
    }  
}
```

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

```
class Student extends Mensch {  
    long matrikelnummer;  
  
    Student(int a, float g, String n,  
            long m) {  
        super(a, g, n);  
        matrikelnummer = m;  
    }  
}
```

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

```
class Student extends Mensch {
    long matrikelnummer;

    Student(int a, float g, String n,
            long m) {
        super(a, g, n);
        matrikelnummer = m;
    }
}
```

```
class Dozent extends Mensch {
    String vorlesung;

    Dozent(int a, float g, String n,
           String v) {
        super(a, g, n);
        vorlesung = v;
    }
}
```

INSTANCEOF UND GETCLASS()

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie.

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

Die Klasse `Student` *erbt* von `Mensch`.

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

- Ob `herbert` wirklich ein `Student` ist,

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

- Ob `herbert` wirklich ein `Student` ist, kann man mit `instanceof` prüfen:

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

- Ob `herbert` wirklich ein `Student` ist, kann man mit `instanceof` prüfen:

```
herbert instanceof Student // → true
```

```
herbert instanceof Mensch // → true
```

```
herbert instanceof Lebewesen // → true
```

```
herbert instanceof Dozent // → false
```

Dabei liefert `null instanceof X` immer `false`, für beliebige Klassen `X`.

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.
- Methoden der Superklasse lassen sich überladen (gleicher Name, andere Signatur),

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.
- Methoden der Superklasse lassen sich überladen (gleicher Name, andere Signatur), als auch überschreiben (gleiche Signatur)

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.
- Methoden der Superklasse lassen sich überladen (gleicher Name, andere Signatur), als auch überschreiben (gleiche Signatur)
- Vererbung sollte nur dann verwendet werden,

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.
- Methoden der Superklasse lassen sich überladen (gleicher Name, andere Signatur), als auch überschreiben (gleiche Signatur)
- Vererbung sollte nur dann verwendet werden, wenn es sich wirklich um eine Erweiterung handelt und nicht wenn die Klasse „nur“ die Eigenschaften (logisch) besitzt.

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.
- Methoden der Superklasse lassen sich überladen (gleicher Name, andere Signatur), als auch überschreiben (gleiche Signatur)
- Vererbung sollte nur dann verwendet werden, wenn es sich wirklich um eine Erweiterung handelt und nicht wenn die Klasse „nur“ die Eigenschaften (logisch) besitzt.

Beispiel: **Auto** sollte nicht von **Motor** erben, da es zwar einen besitzt, aber kein Motor ist.

ÜBERSCHREIBEN

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {  
    public int foo(int i) { return i + 1; }  
}
```

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {  
    public int foo(int i) { return i + 1; }  
}  
public class B extends A {  
    public int foo(int i) { return i * i; }  
}
```

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {  
    public int foo(int i) { return i + 1; }  
}  
public class B extends A {  
    public int foo(int i) { return i * i; }  
}
```

- Hier überschreibt `B::foo(int)` die Methode `A::foo(int)`.

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {  
    public int foo(int i) { return i + 1; }  
}  
public class B extends A {  
    public int foo(int i) { return i * i; }  
}
```

- Hier überschreibt `B::foo(int)` die Methode `A::foo(int)`.
- `new B().foo(3)` produziert damit `9`. In `B` steht `A::foo(int)` über `super` zur Verfügung.

ABSTRAKTE KLASSEN

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, ...

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, ... die diese Operationen implementieren.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, ... die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind,

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, ... die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **abstract** bezeichnen.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, ... die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **abstract** bezeichnen.
- Von **abstract** Klassen darf kein Objekt erstellt werden. Von ihnen kann aber geerbt werden.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, ... die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **abstract** bezeichnen.
- Von **abstract** Klassen darf kein Objekt erstellt werden.
Von ihnen kann aber geerbt werden.
- Eine abstrakte Klasse kann abstrakte Methoden (ohne Rumpf!) definieren.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, ... die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **abstract** bezeichnen.
- Von **abstract** Klassen darf kein Objekt erstellt werden. Von ihnen kann aber geerbt werden.
- Eine abstrakte Klasse kann abstrakte Methoden (ohne Rumpf!) definieren. Diese *müssen* von einer erbenden, nicht-abstrakten Klasse implementiert werden.

ABSTRAKTE KLASSEN, EIN BEISPIEL

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {
```

```
}
```


ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {  
  
    // Für 'super()'  
    public AbstractList() { }  
  
}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {  
  
    // Für 'super()'  
    public AbstractList() { }  
  
    abstract void add(Element e);  
  
}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {  
  
    // Für 'super()'  
    public AbstractList() { }  
  
    abstract void add(Element e);  
  
    int size() {  
        // ...  
    }  
  
}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {  
  
    // Für 'super()'  
    public AbstractList() { }  
  
    abstract void add(Element e);  
  
    int size() {  
        // ...  
    }  
    // ...  
}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
class ArrayList extends AbstractList {
```

```
}
```


ABSTRAKTE KLASSEN, EIN BEISPIEL

```
class ArrayList extends AbstractList {  
    Element[] elems;  
  
    public ArrayList() {  
  
    }  
  
}
```


ABSTRAKTE KLASSEN, EIN BEISPIEL

```
class ArrayList extends AbstractList {  
    Element[] elems;  
  
    public ArrayList() {  
        super();  
        elems = new Element[0];  
    }  
  
}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
class ArrayList extends AbstractList {
    Element[] elems;

    public ArrayList() {
        super();
        elems = new Element[0];
    }

    public void add(Element e) {
        // ...
    }
}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
class ArrayList extends AbstractList {  
    Element[] elems;  
  
    public ArrayList() {  
        super();  
        elems = new Element[0];  
    }  
  
    public void add(Element e) {  
        // ...  
    }  
    // ...  
}
```

STATISCHE BINDUNG

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:
 - STATISCH: Nehmen wir an, wir haben zwei Klassen **A** und **B**, wobei **B** von **A** erbt.

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:
 - STATISCH: Nehmen wir an, wir haben zwei Klassen **A** und **B**, wobei **B** von **A** erbt. Beide deklarieren die Variable **i**, in **A** wird sie auf 32, in **B** auf 42 gesetzt.

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:
 - STATISCH: Nehmen wir an, wir haben zwei Klassen **A** und **B**, wobei **B** von **A** erbt. Beide deklarieren die Variable **i**, in **A** wird sie auf 32, in **B** auf 42 gesetzt. Es ergibt sich:

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

STATISCH: Nehmen wir an, wir haben zwei Klassen **A** und **B**, wobei **B** von **A** erbt. Beide deklarieren die Variable **i**, in **A** wird sie auf 32, in **B** auf 42 gesetzt. Es ergibt sich:

```
A a = new A();  
B b = new B();  
A c = new B();
```

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

STATISCH: Nehmen wir an, wir haben zwei Klassen **A** und **B**, wobei **B** von **A** erbt. Beide deklarieren die Variable **i**, in **A** wird sie auf 32, in **B** auf 42 gesetzt. Es ergibt sich:

```
A a = new A();  
B b = new B();  
A c = new B();  
System.out.println(a.i); // → 32
```

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

STATISCH: Nehmen wir an, wir haben zwei Klassen **A** und **B**, wobei **B** von **A** erbt. Beide deklarieren die Variable **i**, in **A** wird sie auf 32, in **B** auf 42 gesetzt. Es ergibt sich:

```
A a = new A();  
B b = new B();  
A c = new B();  
System.out.println(a.i); // → 32  
System.out.println(b.i); // → 42
```

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

STATISCH: Nehmen wir an, wir haben zwei Klassen **A** und **B**, wobei **B** von **A** erbt. Beide deklarieren die Variable **i**, in **A** wird sie auf 32, in **B** auf 42 gesetzt. Es ergibt sich:

```
A a = new A();  
B b = new B();  
A c = new B();  
System.out.println(a.i); // → 32  
System.out.println(b.i); // → 42  
System.out.println(c.i); // → 32
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:
DYNAMISCH: Angenommen die Klasse **Mensch** hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt.

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:
DYNAMISCH: Angenommen die Klasse **Mensch** hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. **Student** überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus.

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:
DYNAMISCH: Angenommen die Klasse **Mensch** hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. **Student** überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:
DYNAMISCH: Angenommen die Klasse **Mensch** hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. **Student** überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);  
Student b = new Student(/*...*/);  
Mensch c = new Student(/*...*/);
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCH: Angenommen die Klasse **Mensch** hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. **Student** überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);  
Student b = new Student(/*...*/);  
Mensch c = new Student(/*...*/);  
a.hallo(); // → Hallo ich bin ein Mensch
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCH: Angenommen die Klasse **Mensch** hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. **Student** überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);  
Student b = new Student(/*...*/);  
Mensch c = new Student(/*...*/);  
a.hallo(); // → Hallo ich bin ein Mensch  
b.hallo(); // → Hallo ich bin ein Student
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCH: Angenommen die Klasse **Mensch** hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. **Student** überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);  
Student b = new Student(/*...*/);  
Mensch c = new Student(/*...*/);  
a.hallo(); // → Hallo ich bin ein Mensch  
b.hallo(); // → Hallo ich bin ein Student  
c.hallo(); // → Hallo ich bin ein Student
```

WENN VERERBUNG NICHT REICHT

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren,

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **implements** implementieren.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **implements** implementieren.
- Ein solches **interface** fordert gewisse Methoden vom Implementor.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **implements** implementieren.
- Ein solches **interface** fordert gewisse Methoden vom Implementor.
- Weiter kann ein **interface** seit Java 8 durch **default** Standardimplementationen für die Methoden liefern.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **implements** implementieren.
- Ein solches **interface** fordert gewisse Methoden vom Implementor.
- Weiter kann ein **interface** seit Java 8 durch **default** Standardimplementationen für die Methoden liefern.
- Ein **interface** kann *keine* Attribute definieren, diese werden automatisch zu Konstanten.

WENN VERERBUNG NICHT REICHT

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.
- Da *Generics* kein Teil der Vorlesung sind, wurde das Interface *Sortable* definiert:

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.
- Da *Generics* kein Teil der Vorlesung sind, wurde das Interface *Sortable* definiert:

```
public interface Sortable {  
    boolean le(Sortable o); // '<='  
    boolean lt(Sortable o); // '<'  
    boolean eq(Sortable o); // '=='  
}
```

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.
- Da *Generics* kein Teil der Vorlesung sind, wurde das Interface `Sortable` definiert:

```
public interface Sortable {  
    boolean le(Sortable o); // '<='  
    boolean lt(Sortable o); // '<'  
    boolean eq(Sortable o); // '=='  
}
```

- Dies zeigt auch, wie man Interface-Bezeichnet wie (abstrakte) Klassen als Typanforderung liefern kann. (Polymorphieeee)

INTERFACE BEISPIEL

```
public class Element implements Sortable /*, interfaceB, ... */ {
    int value;
    Element next;

    // Nur als Beispiel:
    public boolean le(Sortable o) {
        return this.value <= ((Element)o).value;
    }

    // ...
}
```

WEITERE KOMMENTARE

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **instanceof** erfüllt.

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **instanceof** erfüllt.
- Ein Interface wie `Sortable` wird in Java durch `Comparable` mit nur einer einzelnen Funktion `compareTo()` gelöst.

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **instanceof** erfüllt.
- Ein Interface wie `Sortable` wird in Java durch `Comparable` mit nur einer einzelnen Funktion `compareTo()` gelöst.
- Interfaces sollten dann verwendet werden wenn es um eine Funktionalität geht (wie `Drawable`, `Moveable`, `Consumeable`, ...)

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **instanceof** erfüllt.
- Ein Interface wie `Sortable` wird in Java durch `Comparable` mit nur einer einzelnen Funktion `compareTo()` gelöst.
- Interfaces sollten dann verwendet werden wenn es um eine Funktionalität geht (wie `Drawable`, `Moveable`, `Consumeable`, ...)
- Das Implementieren eines Interfaces kennzeichnet UML durch eine gestrichelte, das Erweitern einer Klasse durch eine durchgezogene Linie.

ARTEN VON FEHLERN

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:
SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:
 - SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.
 - LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:
 - SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.
 - LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.
- Wie schwer ein Fehler ist, hängt von der Situation ab.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:
 - SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.
 - LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.
- Wie schwer ein Fehler ist, hängt von der Situation ab.
- Die Fehlerbehandlung erfolgt meist durch die aufrufende Methode.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:
 - SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.
 - LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.
- Wie schwer ein Fehler ist, hängt von der Situation ab.
- Die Fehlerbehandlung erfolgt meist durch die aufrufende Methode.
- Fehler sind in Java Objekte von Klasse,

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:
 - SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.
 - LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.
- Wie schwer ein Fehler ist, hängt von der Situation ab.
- Die Fehlerbehandlung erfolgt meist durch die aufrufende Methode.
- Fehler sind in Java Objekte von Klasse, die von `Exception` erben.

ARTEN VON FEHLERN

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **throw** werfen.

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **throw** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **null**, ...)

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **throw** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **null**, ...)
- Wir werfen eine explizite Ausnahme (allgemein: ein **Throwable**):

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **throw** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **null**, ...)
- Wir werfen eine explizite Ausnahme (allgemein: ein **Throwable**):
throw new IndexOutOfBoundsException();

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **throw** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **null**, ...)
- Wir werfen eine explizite Ausnahme (allgemein: ein **Throwable**):
throw new IndexOutOfBoundsException();
- Wird eine Ausnahme nicht behandelt, bricht das Programm ab.

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **throw** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **null**, ...)
- Wir werfen eine explizite Ausnahme (allgemein: ein **Throwable**):
throw new IndexOutOfBoundsException();
- Wird eine Ausnahme nicht behandelt, bricht das Programm ab.
- Java-Errors, wie ein **InternalError** lassen sich nicht sinnvoll behandeln!

BEHANDELN VON FEHLERN

BEHANDELN VON FEHLERN

- Um einen Fehler zu behandeln, bietet Java das **try-catch**-Konstrukt

```
try {  
    // Anweisungen, Methodenaufrufe, ...  
} catch(⟨FehlerKlasseA⟩ ⟨Bezeichner⟩) {  
    // Handle Fehler der Klasse ⟨FehlerKlasseA⟩  
} catch(⟨FehlerKlasseB⟩ ⟨Bezeichner⟩) {  
    // Handle Fehler der Klasse ⟨FehlerKlasseB⟩  
} finally {  
    // Anweisungen die immer ausgeführt werden.  
}
```

BEHANDELN VON FEHLERN

- Um einen Fehler zu behandeln, bietet Java das **try-catch**-Konstrukt

```
try {  
    // Anweisungen, Methodenaufrufe, ...  
} catch(⟨FehlerKlasseA⟩ ⟨Bezeichner⟩) {  
    // Handle Fehler der Klasse ⟨FehlerKlasseA⟩  
} catch(⟨FehlerKlasseB⟩ ⟨Bezeichner⟩) {  
    // Handle Fehler der Klasse ⟨FehlerKlasseB⟩  
} finally {  
    // Anweisungen die immer ausgeführt werden.  
}
```

- Das **finally** ist optional, ebenso kann nur ein **catch**-Block auftreten.

CHECKED- & UNCHECKED EXCEPTIONS

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.
- *Checked* Exceptions *müssen* behandelt werden:

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.
- *Checked* Exceptions *müssen* behandelt werden:
 - Entweder direkt durch ein **try-catch** (welches die Exception behandelt).

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.
- *Checked* Exceptions *müssen* behandelt werden:
 - Entweder direkt durch ein **try-catch** (welches die Exception behandelt).
 - Oder durch eine Weiterreichung per **throws** bei der Methoden-Deklaration:

```
public void foo() throws Exception/*, ExceptionB, ...*/ {  
    throw new Exception();  
}
```

EXCEPTIONS WEITERWERFEN

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **try-catch** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **try-catch** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **try-catch**-Konstrukte verschachteln.

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **try-catch** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **try-catch**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {  
    try {  
        throw new Exception();  
    } catch (Exception ex) {  
  
    }  
}
```

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **try-catch** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **try-catch**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {  
    try {  
        throw new Exception();  
    } catch (Exception ex) {  
  
    }  
}
```

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **try-catch** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **try-catch**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {  
    try {  
        throw new Exception();  
    } catch (Exception ex) {  
        // do stuff ..  
    }  
}
```

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **try-catch** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **try-catch**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {  
    try {  
        throw new Exception();  
    } catch (Exception ex) {  
        // do stuff ..  
        throw ex; // oder sowas wie: throw new Exception(ex)  
    }  
}
```


BEHANDELN VON FEHLERN

BEHANDELN VON FEHLERN

- Bei mehreren **catch**-Blöcken wird von oben nach unten ein passender gesucht.

BEHANDELN VON FEHLERN

- Bei mehreren **catch**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von `Exception`),

BEHANDELN VON FEHLERN

- Bei mehreren **catch**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von `Exception`), sollten die „allgemeineren“ weiter unten stehen.

BEHANDELN VON FEHLERN

- Bei mehreren **catch**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von `Exception`), sollten die „allgemeineren“ weiter unten stehen.
- Beispiel:

BEHANDELN VON FEHLERN

- Bei mehreren **catch**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von **Exception**), sollten die „allgemeineren“ weiter unten stehen.
- Beispiel:

```
try {  
    System.out.println(42 / 0);  
} catch (ArithmeticException ex) {  
    // Behandlung  
    System.err.println("Division_durch_0!");  
    ex.printStackTrace();  
}
```

ÜBUNGSAUFGABE

Aufgabe 62: Fehler finden - Exceptions

(5 Minuten)

Erklären Sie alle Verletzungen der Java-Syntax (die Import sind korrekt).
`IOException` und `FileNotFoundException` sind checked-Exceptions.
`ArithmeticException` ist eine unchecked-Exception.

Aufgabe 62: Fehler finden - Exceptions

(5 Minuten)

Erklären Sie alle Verletzungen der Java-Syntax (die Import sind korrekt).
`IOException` und `FileNotFoundException` sind checked-Exceptions.
`ArithmeticException` ist eine unchecked-Exception.

```
public class A {
    public static void main(String[] args) throws FileNotFoundException {
        try { doStuff(); } catch (Exception ex) {}
        try { doElse(); } catch (RuntimeException ex) {}
        throw new IOException("ABC");
        try {
            throw new ArithmeticException();
        } catch(IOException ignored) {}
    }
    public static void doStuff() { throw new FileNotFoundException(); }
    public static void doElse() { throw RuntimeException; }
}
```


Lösung 62: Fehler finden - Exceptions

1. **throw** `RuntimeException` liefert einen Symbolfehler: es muss ein Objekt geworfen werden.

Lösung 62: Fehler finden - Exceptions

1. **throw** `RuntimeException` liefert einen Symbolfehler: es muss ein Objekt geworfen werden.
Korrekt wäre **throw new** `RuntimeException()`;

Lösung 62: Fehler finden - Exceptions

1. **throw** `RuntimeException` liefert einen Symbolfehler: es muss ein Objekt geworfen werden.
Korrekt wäre **throw new** `RuntimeException()`;
2. **catch**(`IOException ignored`) für das dritte try-catch ist illegal: Der try-Block wirft keine `IOException`.

Lösung 62: Fehler finden - Exceptions

1. **throw** `RuntimeException` liefert einen Symbolfehler: es muss ein Objekt geworfen werden.
Korrekt wäre **throw new** `RuntimeException()`;
2. **catch**(`IOException ignored`) für das dritte try-catch ist illegal: Der try-Block wirft keine `IOException`.
Korrekt, wenn auch unsinnig, wäre das Abfangen der `ArithmeticException`.

Lösung 62: Fehler finden - Exceptions

1. **throw** `RuntimeException` liefert einen Symbolfehler: es muss ein Objekt geworfen werden.
Korrekt wäre **throw new** `RuntimeException()`;
2. **catch**(`IOException ignored`) für das dritte try-catch ist illegal: Der try-Block wirft keine `IOException`.
Korrekt, wenn auch unsinnig, wäre das Abfangen der `ArithmeticException`.
3. Der Code nach der „ABC“-`IOException` ist illegal, die Anweisungen werden nie ausgeführt.

Lösung 62: Fehler finden - Exceptions

1. **throw** `RuntimeException` liefert einen Symbolfehler: es muss ein Objekt geworfen werden.
Korrekt wäre **throw new** `RuntimeException()`;
2. **catch**(`IOException ignored`) für das dritte try-catch ist illegal: Der try-Block wirft keine `IOException`.
Korrekt, wenn auch unsinnig, wäre das Abfangen der `ArithmeticException`.
3. Der Code nach der „ABC“-`IOException` ist illegal, die Anweisungen werden nie ausgeführt.
Dafür müsste der Fehler abgefangen werden.

Lösung 62: Fehler finden - Exceptions

(Fortsetzung)

4. Die `IOException`, die in der `main` geworfen wird, muss markiert oder abgefangen werden.

Lösung 62: Fehler finden - Exceptions

(Fortsetzung)

4. Die `IOException`, die in der `main` geworfen wird, muss markiert oder abgefangen werden.
5. In `doStuff()` muss die `FileNotFoundException` entweder markiert werden (mittels **throws**) oder abgefangen (mittels **try**).

Lösung 62: Fehler finden - Exceptions

(Fortsetzung)

4. Die `IOException`, die in der `main` geworfen wird, muss markiert oder abgefangen werden.
5. In `doStuff()` muss die `FileNotFoundException` entweder markiert werden (mittels **throws**) oder abgefangen (mittels **try**). Dies gilt für alle Exceptions, bei denen es sich nicht um Runtime-Exceptions handelt.

Hinweis: Mit **throws** lassen sich auch mehr Exceptions markieren. Für Java ist **throws** `FileNotFoundException` kein Fehler.

ÜBUNGSAUFGABE

Aufgabe 63: Statische und Dynamische Bindung (4 Minuten)

Erklären Sie kurz den Unterschied zwischen statischer und dynamischer Bindung und wann Java was verwendet.

Lösung 63: Statische und Dynamische Bindung

Die *statische* Bindung verwendet Java bei Attributen. Sie wird zur Kompilierzeit aufgelöst und sorgt dafür, dass in Subklassen überschriebene Variablen stets der verwendeten Klasse zugeordnet sind. In `A a = new B();` würde sich `a.x` also auf das `x` der Klasse `A` beziehen. In `B a = new B();` auf das der Klasse `B`.

Die *dynamische* Bindung verwendet Java bei Methoden. Sie wird zur Laufzeit aufgelöst und bedeutet, dass im Falle des Überschreibens von Methoden einer Subklasse die Klasse des Objekts entscheidet welche verwendet wird und nicht die der Variable die darauf zeigt.

Lösung 63: Statische und Dynamische Bindung

Die *statische* Bindung verwendet Java bei Attributen. Sie wird zur Kompilierzeit aufgelöst und sorgt dafür, dass in Subklassen überschriebene Variablen stets der verwendeten Klasse zugeordnet sind. In `A a = new B();` würde sich `a.x` also auf das `x` der Klasse `A` beziehen. In `B a = new B();` auf das der Klasse `B`.

Die *dynamische* Bindung verwendet Java bei Methoden. Sie wird zur Laufzeit aufgelöst und bedeutet, dass im Falle des Überschreibens von Methoden einer Subklasse die Klasse des Objekts entscheidet welche verwendet wird und nicht die der Variable die darauf zeigt.

Hier hilft auch Aufgabe 64.

ÜBUNGSAUFGABE

Aufgabe 64: Was liefert dieser Code?

(4 Minuten)

```
public class A {
    int a = 0;
    void x(int a) { this.a = 2 - a; }

    static class B extends A {
        int a = 1;
        void x(int a) { this.a = 2 * a; }
    }

    public static void main(String[] args) {
        A a = new A(); a.x(4);
        B b = new B(); b.x(4);
        A c = new B(); c.x(4);
        System.out.format("%d_%d_%d", a.a, b.a, c.a);
    }
}
```


Lösung 64: Was liefert dieser Code?

Er liefert „-2 8 0“. Warum?

-2: Es wird `A::x` aufgerufen, damit ist `a`: $2 - 4 = -2$.

8: Es wird `B::x` aufgerufen, damit ist `a`: $2 \cdot 4 = 8$.

0: Da Attribute statisch gebunden werden, verändert `B::x` *nicht* das `a` in `A` und dieses bleibt damit beim initialen Wert: 0.

ÜBUNGSAUFGABE

Aufgabe 65: Vererbungshierarchien

(6 Minuten)

Aufgabe 65: Vererbungshierarchien

(6 Minuten)

Konstruieren Sie eine *abstrakte* Klasse `Entity`, die die gemeinsamen Eigenschaften folgender Klassen eint, die Sichtbarkeiten und Modifikatoren bestmöglich rekreiert, sich aber in einem anderen Ordner befindet.

Aufgabe 65: Vererbungshierarchien

(6 Minuten)

Konstruieren Sie eine *abstrakte* Klasse `Entity`, die die gemeinsamen Eigenschaften folgender Klassen eint, die Sichtbarkeiten und Modifikatoren bestmöglich rekreiert, sich aber in einem anderen Ordner befindet. Schreiben Sie die Klassen so um, dass sie nun `Entity` verwenden.

Aufgabe 65: Vererbungshierarchien

(6 Minuten)

Konstruieren Sie eine *abstrakte* Klasse `Entity`, die die gemeinsamen Eigenschaften folgender Klassen eint, die Sichtbarkeiten und Modifikatoren bestmöglich rekreiert, sich aber in einem anderen Ordner befindet. Schreiben Sie die Klassen so um, dass sie nun `Entity` verwenden.

```
class Enemy {
    private final long id; protected int hp, ap, x, y;
    public Enemy(long id) { this.id = id; }
    public void move(int x, int y) {
        this.x = x; this.y = y;
    }
    Enemy whoIsEvil() {
        System.out.println("I_LaaaamL("+ id +)");
        return this;
    }
}
```

Aufgabe 65: Vererbungshierarchien

(6 Minuten)

Konstruieren Sie eine *abstrakte* Klasse `Entity`, die die gemeinsamen Eigenschaften folgender Klassen eint, die Sichtbarkeiten und Modifikatoren bestmöglich rekreiert, sich aber in einem anderen Ordner befindet. Schreiben Sie die Klassen so um, dass sie nun `Entity` verwenden.

```
class Enemy {
    private final long id; protected int hp, ap, x, y;
    public Enemy(long id) { this.id = id; }
    public void move(int x, int y) {
        this.x = x; this.y = y;
    }
    Enemy whoIsEvil() {
        System.out.println("I_LaaaamL("+ id +)");
        return this;
    }
}

class Companion {
    private final long id; protected int x, y;
    public Companion(long id) { this.id = id; }
    public void move(int x, int y) {
        this.y = y; this.x = x;
    }
}
```

Aufgabe 65: Vererbungshierarchien

(6 Minuten)

Konstruieren Sie eine *abstrakte* Klasse `Entity`, die die gemeinsamen Eigenschaften folgender Klassen eint, die Sichtbarkeiten und Modifikatoren bestmöglich rekreiert, sich aber in einem anderen Ordner befindet. Schreiben Sie die Klassen so um, dass sie nun `Entity` verwenden.

```
class Enemy {
    private final long id; protected int hp, ap, x, y;
    public Enemy(long id) { this.id = id; }
    public void move(int x, int y) {
        this.x = x; this.y = y;
    }
    Enemy whoIsEvil() {
        System.out.println("I_Laaaam_("+ id +)");
        return this;
    }
}

class Companion {
    private final long id; protected int x, y;
    public Companion(long id) { this.id = id; }
    public void move(int x, int y) {
        this.y = y; this.x = x;
    }
}
```

```
class NPC {
    private final long id; protected int x, y;
    public final String visibleName;
    public NPC(String name, long id) {
        this.visibleName = name; this.id = id;
    }

    public void move(int x, int y) {
        this.x = x; this.y = y;
        annoyingInterruption();
    }

    void annoyingInterruption() {
        System.out.println("Movin'_"+ pos +)");
    }
}
```


Lösung 65: Vererbungshierarchien

Lösung 65: Vererbungshierarchien

```
abstract class Entity {  
    protected final long id;  
    protected int x, y; // oder private mit Getter  
    public Entity(long id) { this.id = id; }  
    public void move(int x, int y) {  
        this.x = x; this.y = y;  
    }  
}
```

Lösung 65: Vererbungshierarchien

```
abstract class Entity {
    protected final long id;
    protected int x, y; // oder private mit Getter
    public Entity(long id) { this.id = id; }
    public void move(int x, int y) {
        this.x = x; this.y = y;
    }
}

class Enemy extends Entity {
    protected int hp, ap;
    public Enemy(long id) { super(id); }

    Enemy whoIsEvil() {
        System.out.println("I_aaaam_("+ id +")");
        return this;
    }
}
```


Lösung 65: Vererbungshierarchien

```
abstract class Entity {
    protected final long id;
    protected int x, y; // oder private mit Getter
    public Entity(long id) { this.id = id; }
    public void move(int x, int y) {
        this.x = x; this.y = y;
    }
}

class Enemy extends Entity {
    protected int hp, ap;
    public Enemy(long id) { super(id); }

    Enemy whoIsEvil() {
        System.out.println("I_Laaaam_(" + id + ")");
        return this;
    }
}

class Companion extends Entity {
    public Companion(long id) { super(id); }
}
```

Lösung 65: Vererbungshierarchien

```
abstract class Entity {
    protected final long id;
    protected int x, y; // oder private mit Getter
    public Entity(long id) { this.id = id; }
    public void move(int x, int y) {
        this.x = x; this.y = y;
    }
}
```

```
class Enemy extends Entity {
    protected int hp, ap;
    public Enemy(long id) { super(id); }

    Enemy whoIsEvil() {
        System.out.println("I_Laaaam_"+ id + "");
        return this;
    }
}
```

```
class Companion extends Entity {
    public Companion(long id) { super(id); }
}
```

```
class NPC extends Entity {
    public final String visibleName;
    public NPC(String name, long id) {
        super(id); this.visibleName = name;
    }
    public void move(int x, int y) {
        super.move(x, y);
        annoyingInterruption();
    }
    void annoyingInterruption() {
        System.out.println("Movin'_"+ pos + "");
    }
}
```

ÜBUNGSAUFGABE

Aufgabe 66: Fehler finden, V

(4 Minuten)

Finden und korrigieren Sie alle (syntaktischen und semantischen) Fehler:

Aufgabe 66: Fehler finden, V

(4 Minuten)

Finden und korrigieren Sie alle (syntaktischen und semantischen) Fehler:

```
public class A {
    private static final class B extends A {
        public B() {
            super(this);
        }
    }

    public A(A other){}
    public abstract class C extends A.B {
        protected abstract boolean _$get$(String);
    }
}
```


Lösung 66: Fehler finden, V

```
public class A {  
    private static class B extends A {  
        public B() {  
            super(null);  
        }  
    }  
  
    public A(A other){}  
    public abstract class C extends A.B {  
        protected abstract boolean _$get$(String a);  
    }  
}
```

Lösung 66: Fehler finden, V

(Fortsetzung)

1. Das **final** ist an sich nicht falsch, dann kann aber **C** nicht mehr von **B** erben.
2. Der Bezeichner **this** darf nicht innerhalb eines Konstruktoraufrufs verwendet werden.
3. In der Methode `_get` genügt nicht die Signatur,

Lösung 66: Fehler finden, V

(Fortsetzung)

1. Das **final** ist an sich nicht falsch, dann kann aber **C** nicht mehr von **B** erben.
2. Der Bezeichner **this** darf nicht innerhalb eines Konstruktoraufrufs verwendet werden.
3. In der Methode `_get` genügt nicht die Signatur, Java erwartet (nicht-bindende) Variablenbezeichner.

Lösung 66: Fehler finden, V

(Fortsetzung)

1. Das **final** ist an sich nicht falsch, dann kann aber **C** nicht mehr von **B** erben.
2. Der Bezeichner **this** darf nicht innerhalb eines Konstruktoraufrufs verwendet werden.
3. In der Methode `_get` genügt nicht die Signatur, Java erwartet (nicht-bindende) Variablenbezeichner.

Hinweis:

Lösung 66: Fehler finden, V

(Fortsetzung)

1. Das **final** ist an sich nicht falsch, dann kann aber **C** nicht mehr von **B** erben.
2. Der Bezeichner **this** darf nicht innerhalb eines Konstruktoraufrufs verwendet werden.
3. In der Methode `_get` genügt nicht die Signatur, Java erwartet (nicht-bindende) Variablenbezeichner.

Hinweis: Die Dollarzeichen sind kein Fehler,

Lösung 66: Fehler finden, V

(Fortsetzung)

1. Das **final** ist an sich nicht falsch, dann kann aber **C** nicht mehr von **B** erben.
2. Der Bezeichner **this** darf nicht innerhalb eines Konstruktoraufrufs verwendet werden.
3. In der Methode `_get` genügt nicht die Signatur, Java erwartet (nicht-bindende) Variablenbezeichner.

Hinweis: Die Dollarzeichen sind kein Fehler, aber es ist Konvention sie nicht zu verwenden (Java verwendet sie intern für zum Beispiel generierten Code).

ÜBUNGSAUFGABE

Aufgabe 67: Fehler finden, VI

(4 Minuten)

Finden und erklären Sie alle (Kompilier-)Fehler:

Aufgabe 67: Fehler finden, VI

(4 Minuten)

Finden und erklären Sie alle (Kompilier-)Fehler:

```
public class MyL1tt13C14ss extends MyL1tt13C14ss.A {
    private class A {
        public int x;
        A(int x) { this.x = x; }
    }

    public void MyEp1cM3th0d {
        System.out::println(5 == 2 + + x);
    }
}
```


Lösung 67: Fehler finden, VI

```
public class MyL1tt13C14ss extends MyL1tt13C14ss.A {  
    private class A {  
        public int x;  
        A(int x) { this.x = x; }  
    }  
  
    public void MyEp1cM3th0d {  
        System.out.println(5 == 2 + + x);  
    }  
}
```

Lösung 67: Fehler finden, VI

(Fortsetzung)

1. `MyL1tt13C14ss extends MyL1tt13C14ss.A` ist eine Variante des zyklischen Vererbungs-Problems und in Java nicht gestattet.
2. Die Methode `MyEp1cM3th0d` benötigt die Klammern zur Kennzeichnung der Parameteranzahl.
3. Der Ausdruck `System.out::println` ist eine Methoden-Referenz. Für den Aufruf benötigen wir `System.out.println`.
4. `=_ =` ist kein gültiger Java-Ausdruck! Nur `==` oder `=`.

Lösung 67: Fehler finden, VI

(Fortsetzung)

1. `MyL1tt13C14ss extends MyL1tt13C14ss.A` ist eine Variante des zyklischen Vererbungs-Problems und in Java nicht gestattet.
2. Die Methode `MyEp1cM3th0d` benötigt die Klammern zur Kennzeichnung der Parameteranzahl.
3. Der Ausdruck `System.out::println` ist eine Methoden-Referenz. Für den Aufruf benötigen wir `System.out.println`.
4. `=_ =` ist kein gültiger Java-Ausdruck! Nur `==` oder `=`.

Dabei ist `2 + + x` kein Problem! es wird als die Addition von `2` zu `+x` (analog zu `+1`) verstanden.

Lösung 67: Fehler finden, VI

(Fortsetzung)

1. `MyL1tt13C14ss extends MyL1tt13C14ss.A` ist eine Variante des zyklischen Vererbungs-Problems und in Java nicht gestattet.
2. Die Methode `MyEp1cM3th0d` benötigt die Klammern zur Kennzeichnung der Parameteranzahl.
3. Der Ausdruck `System.out::println` ist eine Methoden-Referenz. Für den Aufruf benötigen wir `System.out.println`.
4. `=_ =` ist kein gültiger Java-Ausdruck! Nur `==` oder `=`.

Dabei ist `2 + + x` kein Problem! es wird als die Addition von `2` zu `+x` (analog zu `+1`) verstanden.

Übrigens: `MyL1tt13C14ss.A` statisch zu machen, hilft nicht. Java Liebe!

WEITERES — DIE WEIHNACHTSWIEDERHOLUNG

Se greit **Eid!** recap

Hat jemand schon die Pingu-Gang informiert?

Florian Sthler · 22.12.2021



Eine „Weihnachtswiederholung“ per Klick...

EIN WORT ZUM ABSCHLUSS

Wie bereits gesagt erhebt dieses Dokument keinen Anspruch auf Vollständigkeit und sieht sich vermutlich auch weiteren Aktualisierungen unterworfen. Eine aktuelle Variante sollte es stets hier geben: <https://github.com/EagleoutIce/eidi-pseudo-rep>.

Bei Anregungen oder Verbesserungsvorschlägen einfach melden!

Zum Ende dann wohl noch ein bisschen Meta-Gequatsche. Dieses Dokument ergründet sich auf Basis wundervoller 6649 Zeilen und summa summarum 314595 Zeichen einiger Zeit und Arbeit, die in es geflossen ist. Ich hoffe, die war es auch wert!

Florian Sihler, florian.sihler@uni-ulm.de

„Viel Spaß beim Lernen und natürlich viel Erfolg!“

- [1] Florian Sihler. *L^AT_EX*-Package, *tikzpingus*. 2021.
- [2] Florian Sihler. *Die Datenstruktur Heap*. 2021.
- [3] Florian Sihler. *Rekursion, Java-Stack & -Heap*. 2021.
- [4] Florian Sihler. *Traversierungsvarianten*. 2021.

Florian Sihler

Ulm, den 13. September 2022

