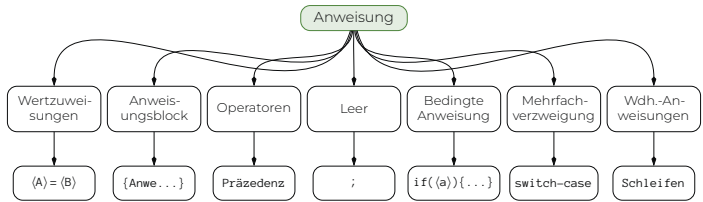
EIDI-KOMPAKT

Volle Version — 2.01.2022



Florian Sihler
Verteilte Systeme — Universität Ulm

Praktische Informatik
Ulm University

THEMENÜBERSICHT, I

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

THEMENÜBERSICHT, II

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 Ti*k*Z 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

1 ○ ○ ○ ○ ○ ○ ○

WAS IST EIN ALGORITHMUS?

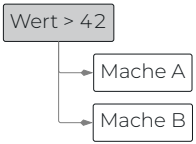
Definition 1: Algorithmus

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 tue:
2 |   Mache A;
3 sonst:
4 |   Mache B;
```

Pseudocode

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

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

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

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

Determiniertheit

Dieselben Eingabedaten erzeugen immer dieselben Ausgabedaten.

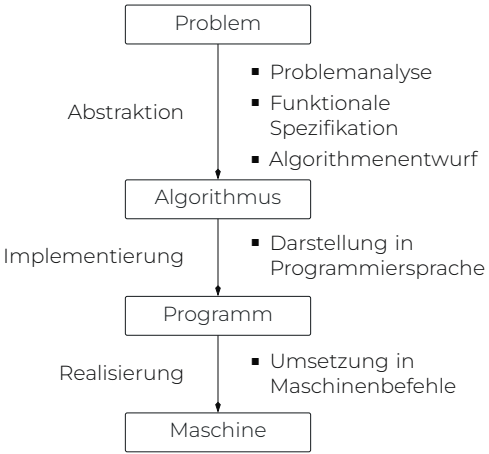
- Die Begriffe sind zu unterscheiden!
- Es können verschiedene Wege zum selben Ziel führen.
- (Sinnfreies) Beispiel: Der Algorithmus kann jedes Element eines Arrays quadrieren. Die Reihenfolge wählt er zufällig aus!

  › Dieser Algorithmus *determiniert*, ist aber *nicht-deterministisch*!

SCHEMA EINES ALGORITHMUS

- 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.

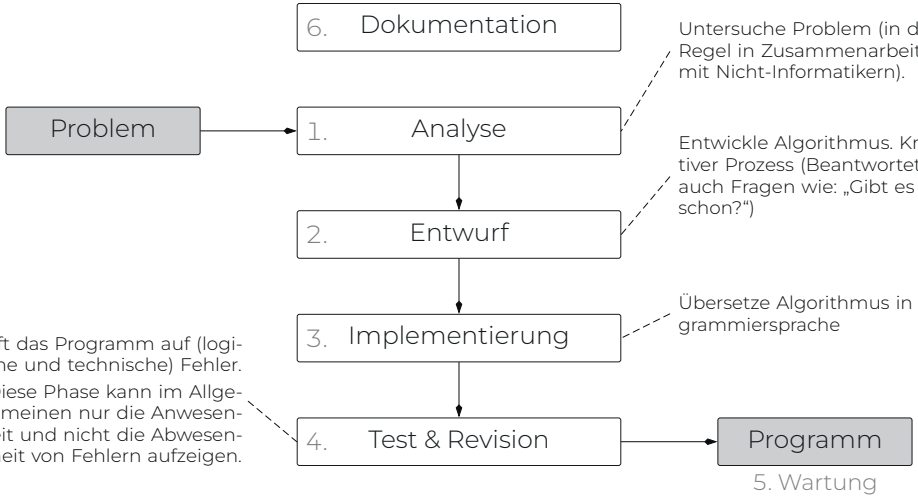
SCHEMA EINES ALGORITHMUS, II



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



ÜBUNGSAUFGABE

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:

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.

Aufgabe 2: Charakteristika (2 Minuten)

Ist der folgende Algorithmus für die Berechnung des Modulo a **mod** 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. So liefert er für ein negatives `b` kein korrektes Ergebnis, da hier `a -= b` zu einem Überlauf führt (bis dahin ist stets `a >= 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:

Algorithm 1: 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:

- 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:

i) nicht-deterministisch ist und determiniert.

ii) sowohl deterministisch ist, als auch determiniert.

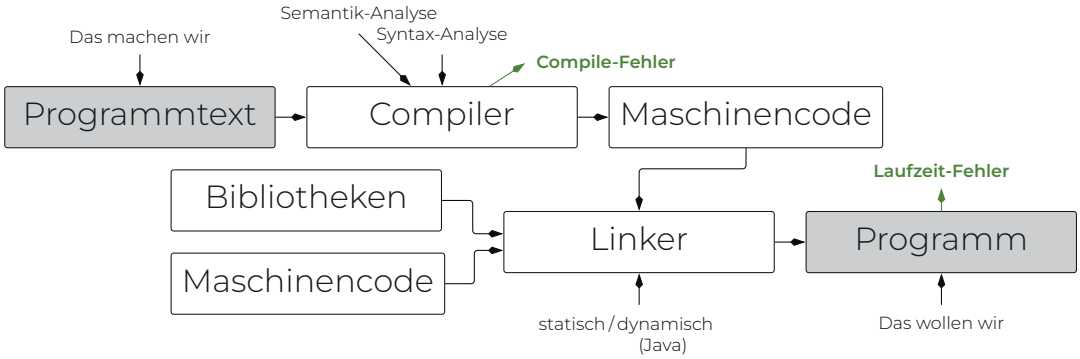
Lösung 4: Deterministische Determiniertheit

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, 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

- 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

```
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

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

  $$\texttt{byte} \prec \texttt{short} \prec \texttt{int} \prec \texttt{long} \prec \texttt{float} \prec \texttt{double}$$

  Sowie:

  $$\texttt{char} \prec \texttt{int}$$

- 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)

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

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

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

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

$$\left[a\texttt{++}, a\texttt{--}\right] \rightarrow \left[\texttt{!}a, -a, \texttt{++}a, \texttt{--}a\right] \rightarrow \left[*, /, \%\right] \rightarrow \left[a+b, a-b\right]$$
$$\rightarrow \left[==, >=, <, \ldots\right] \rightarrow \left[\texttt{\&\&}\right] \rightarrow \left[\texttt{||}\right]$$

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
```

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` → `false`) und `m > 1 && 2 != 3` betrachtet. Nun wird also `m > 1` (`2 > 1` → `true`) und `2 != 3` (`true`) ausgewertet. Damit evaluiert `true && true` zu `true` und somit auch `false || true` zu `true`.

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.

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.
- Auch das Klammern funktioniert wie gewohnt.
- Boolesche Operatoren funktionieren wie in der Aussagenlogik.

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.
- 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

- 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.
- Die allgemeine Syntax lautet:

```
⟨Zugriffsmodifikatoren⟩ ⟨Rückgabetyp⟩ ⟨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

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

- 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

- 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`	☑	☐	☑	☑	☐

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`.

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

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

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

```java
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).
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).
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.

ES GILT FESTZUHALTEN…

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.

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);
}
```

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!

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));
}
```

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)

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)`
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...)`

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`.

```
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 `char`s einfach zu ihrem Integerwert konvertiert werden, findet erst beim letzten Plus eine Konkatenation 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 Unterstich bei `-10` *ist in Java (zwischen Zahlen) komplett valide und wird in der Regel zum Trennen von* `1000`*er Blöcken verwendet.*

Lösung 12: Konkatenation (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 konkateniert, was die Ausgabe erzeugt.

ÜBUNGSAUFGABE

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

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)

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)

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

8. „`"x" + 3`“: `String` (x3)
   Analog zur Aufgabe zur String-Konkatenation wird hier `3` in eine Zeichenkette konvertiert.

Kontrollstrukturen und Arrays

FALLUNTERSCHEIDUNG 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:

  ⟨`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.

FALLUNTERSCHEIDUNG 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 `String`s.

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

- 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)

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.
- Für $i = 42$ unterscheiden sich die Ausgaben. Hier gibt do-while noch „42“ aus, während die while Schleife gar nicht erst betreten wird.

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

- 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)

- 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

- 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 $i = 19$.

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

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

- 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, 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 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

- 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
}
```

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} ⇒ {7, 15, 10}
{1, 2, 3} ⇒ {1, 3}
{31} ⇒ {31}
```

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

- 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

- 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

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

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] + "␣");
}
```

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, 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:

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, 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, das von 0 $\widehat{=}$ `a` bis 25 $\widehat{=}$ `z` die Häufigkeit der Buchstaben angibt. Groß- und Kleinschreibung, sowie andere Zeichen sollen dabei ignoriert werden.

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;
}
```

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

```
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;
}
```

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

```
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

- 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.

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. (⇒ Getter & Setter).

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* 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

Hier angegeben werden exemplarisch vier, jeweils mit kurzer Erklärung:

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

Informationsverdeckung: 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).

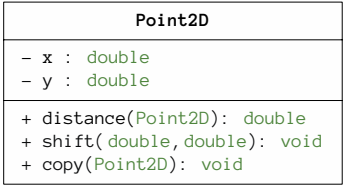
Komposition/Polymorphie: Klassen implementieren Schnittstellen, die sie von der Implementation abstrahieren, so dass diese sich problemlos austauschen lassen.

VOM KLASSENKONZEPT ZUM JAVACODE

- 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.



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

```java
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

- 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.
- 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

```
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.
- 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

- 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 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

- 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

- 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

- 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

- 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

```
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

- Die Funktion `wohinGehtEs` lässt sich nun kompakter fassen:

```
public static String wohinGehtEs(Richtung ziel){
    return ziel.getText();
}
```

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

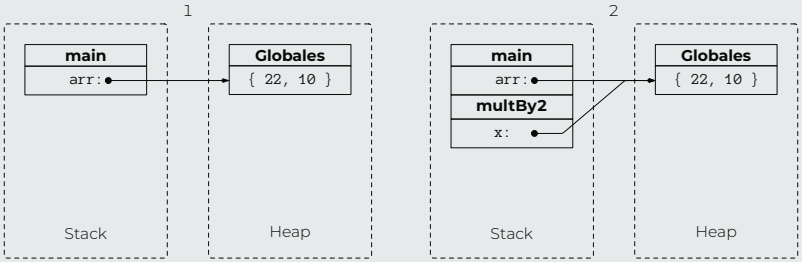
EINE LEICHTE AUFGABE ALS EINSTIEG

Aufgabe 26: Was liefert dieser Code? (1 Minute)

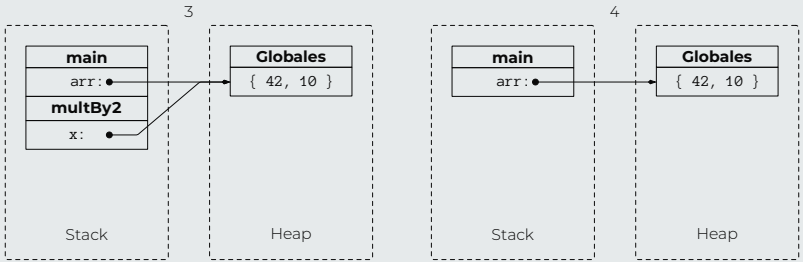
```java
public static void multBy2(int[] x){
    x[0] += x[1] * 2;
}

public static void main(String[] args) {
    int[] arr = { 22, 10 };
    System.out.println(arr[0]);
    multBy2(arr);
    System.out.println(arr[0]);
}
```

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)



Hinweis: eine derartige Grafik wird nicht gefordert werden, sie kann aber durchaus beim Verständnis der Thematik hilfreich sein.

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.

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)

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!*

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

```
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)

Ein Feld in einem gitterartigen Schienennetz kann entweder frei (≙ 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 „Schienenfelder“ der Fall).

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)

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)

```
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

○ ○ ○ ○ 5 ○ ○ ○

Florian Sihler, 2.01.2022

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

- 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.“

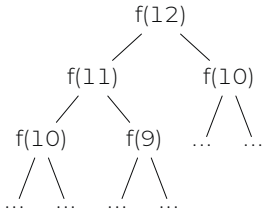
```
// → 1 2 3 4 5 6 ...
public void headDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Rekursion
    else headDecrement(i - 1);
    // Verarbeitung
    System.out.println(i);
}
```

```
// → 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

- 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

- 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 übersetzten.

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

```
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)

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:

$$\begin{aligned} f(15 + 2,\ 2 - 1,\ -2) &= f(17,\ 1,\ -2) \\ f(17 + 1,\ 1 - 1,\ -2) &= f(18,\ 0,\ -2) \\ f(18 + 0,\ 0 - 1,\ -2) &= f(18,\ -1,\ -2) \end{aligned}$$

Mit $b = -1 \not\geq 0$ betreten wir nun den „sonst“-Teil.

Lösung 31: Formel zu Rekursion (Fortsetzung)

Nun bei $f(18, -1, -2)$ berechnen wir $f(a-1, b-1, |c-1|)$:

$$\begin{aligned} f(18-1, -1-1, |-2-1|) &= f(17, -2, 3) \\ f(17-1, -2-1, |3-1|) &= f(16, -3, 2) \\ f(16-1, -3-1, |2-1|) &= f(15, -4, 1) \\ f(15-1, -4-1, |1-1|) &= f(14, -5, 0) \end{aligned}$$

Mit $c = 0$ terminiert die Funktion für $\max(14, -5) = 14$.

DAS PARADIGMA: DIVIDE AND CONQUER

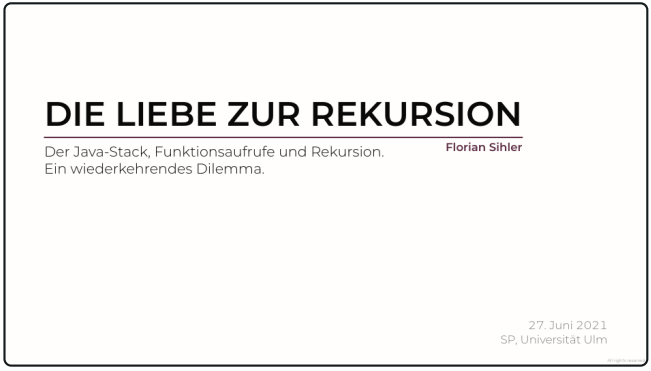
- Manche komplizierte Probleme, lassen sich durch Rekursion in immer kleinere Probleme aufspalten, die dann beherrschbarer sind.
- Dies wird uns bei den Sortieralgorithmen Merge- und Quicksort wieder begegnen.

DAS PARADIGMA: BACKTRACKING

- 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:* 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, …



Mehr zum Thema „Rekursion“ per Klick…

KOMPLEXITÄTSBETRACHTUNG

- 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

- 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

- 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`: $\frac{n(n-1)}{2}$

- 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

- 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

UML – GRUNDLAGEN

- 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:

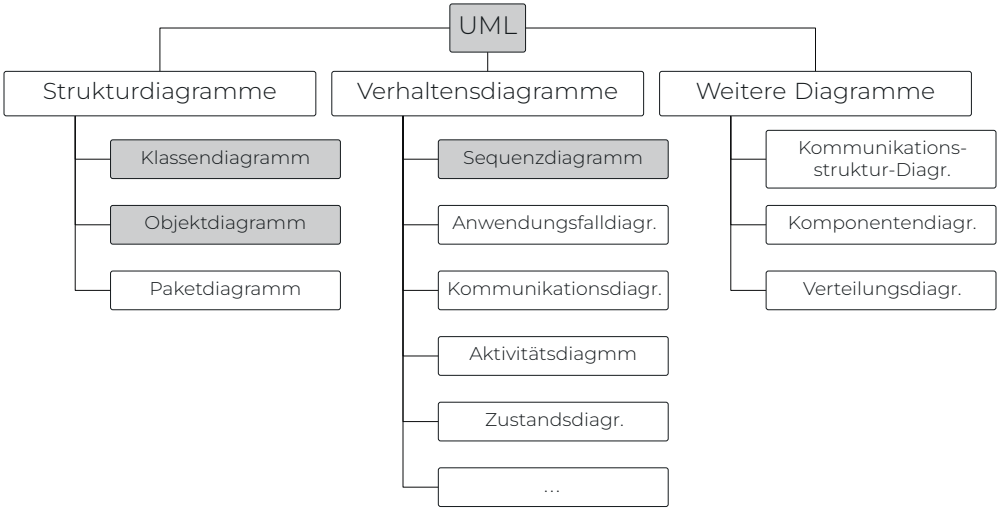
  KLASSENDIAGRAMME: modellieren die Beziehungen und Eigenschaften der beteiligten Klassen.

  OBJEKTDIAGRAMME: modellieren die Beziehungen und Ausprägungen (spezifischer) Objekte.

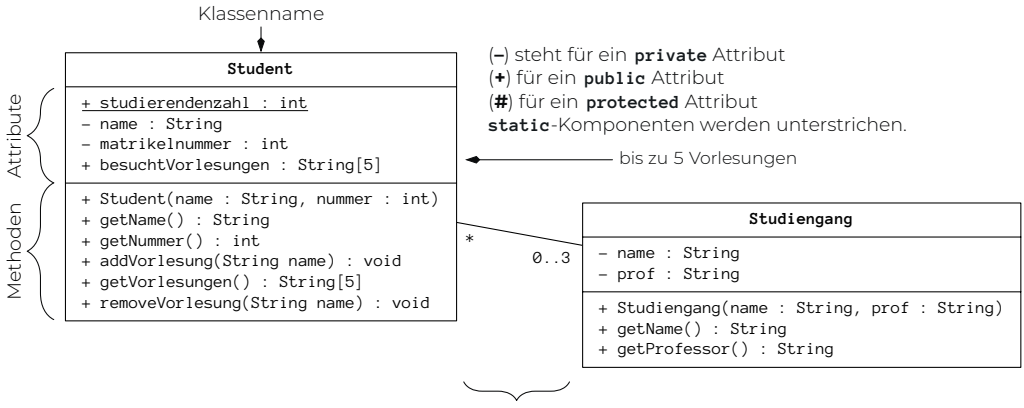
  SEQUENZDIAGRAMME: modellieren den Nachrichtenaustausch in einem Programm. Sie sind ereignisbasiert.

- UML wird hier (wie in der Vorlesung auch) nur oberflächlich betrachtet.

UML – EIN ÜBERBLICK



UML – KLASSENDIAGRAMME

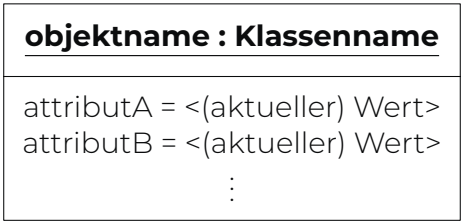


UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
  - gerichtete Assoziationen (A ⟶ B).
  - Abhängigkeiten (A ⇢ B).
  - Vererbungen (A ⟶▷ B).
  - Aggregationen (A ◇⟶ B). Markiert meist „Besitz“: „A besitzt ein B“.
  - Kompositionen (A ◆⟶ 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, …).

UML – OBJEKTDIAGRAMME

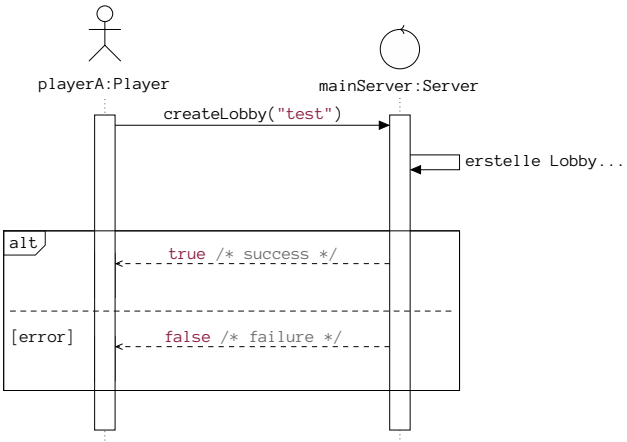
- Objektdiagramme ähneln Klassendiagrammen.
- Allerdings handelt es sich immer um explizite Ausprägungen einer Klasse (⇒ keine Methoden).



UML – SEQUENZDIAGRAMME

- Bilden den Nachrichtenaustausch ab. Das Senden und Empfangen wird auf Basis von Ereignissen ausgelöst, diese rufen wiederrum 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

- 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

- 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 $A \mathrel{\widehat{=}} 10$ bis $F \mathrel{\widehat{=}} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „ $\bmod b$“ generieren und die Zahl dann (ohne Rest) durch b teilen.
- Beispiel: $z = 10$ soll ins Dualsystem konvertiert werden:

$$\begin{aligned} 10 \div 2 &= 5 & 10 \bmod 2 &= 0 \ (\leftarrow \mathit{LSB}) \\ 5 \div 2 &= 2 & 5 \bmod 2 &= 1 \\ 2 \div 2 &= 1 & 2 \bmod 2 &= 0 \\ 1 \div 2 &= 0 & 1 \bmod 2 &= 1 \ (\leftarrow \mathit{MSB}) \end{aligned}$$

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:

$$AFFE_{(16)} = \overbrace{1010}^{A}\ \overbrace{1111}^{F}\ \overbrace{1111}^{F}\ \overbrace{1110}^{E}\ _{(2)}$$

Aufgabe 32: Zahlen konvertieren (3 Minuten)

Konvertieren Sie $42_{(8)}$ zur Basis $b = 16$, $b = 3$ und ins Binärsystem.

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: $42_{(8)} = \overbrace{100}^{4}\overbrace{010}^{2}{}_{(2)}$

B = 16: $100010_{(2)} = \overbrace{2}^{0010}\overbrace{2}^{0010}{}_{(16)}$ (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:

$$
\begin{aligned}
34 \div 3 &= 11 & 34 &\bmod 3 = 1 \ (\leftarrow \mathit{LSB}) \\
11 \div 3 &= 3 & 11 &\bmod 3 = 2 \\
3 \div 3 &= 1 & 3 &\bmod 3 = 0 \\
1 \div 3 &= 0 & 1 &\bmod 3 = 1 \ (\leftarrow \mathit{MSB})
\end{aligned}
$$

Damit ergibt sich: $42_{(8)} = 100010_{(2)} = 22_{(16)} = 34_{(10)} = 1021_{(3)}$

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

```
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:

```
int2bin(42) // → "101010"
```

Hinweis: Um eine Zahl in einen String zu konvertieren, kann Konkatenation oder `Integer.toString(int)` verwendet werden.

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 positiven 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 $z_i \in \{1, \ldots, 9\}$ unterstützen.

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 (`number / 10`), fügen diese „von vorne nach hinten“ an die neue Zahl an (`current * 10 + number % 10`).

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 1<u>00</u>55001 gehen hier verloren.

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;
}
```

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)

Sei die Länge von `a` durch n notiert (`a.length()` $\widehat{=}\, n$)

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.

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. $\mathcal{O}(4^n) - \mathcal{O}(2^n)$
3. $\mathcal{O}(\frac{n}{12}) + \mathcal{O}(n^3)$
4. $\mathcal{O}(\frac{n!}{n})$
5. $\mathcal{O}(14n + 12)$

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. $\mathcal{O}(\frac{n}{12}) + \mathcal{O}(n^3) = \mathcal{O}(n^3)$
2. $\mathcal{O}(4^n) - \mathcal{O}(2^n) = \mathcal{O}(4^n - 2^n) = \mathcal{O}(4^n)$
4. $\mathcal{O}(\frac{n!}{n}) = \mathcal{O}((n-1)!) = \mathcal{O}(n!)$

Eine derartige Kategorisierung kann auch (noch) genauer präzisiert 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:

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:** $a * wert$;

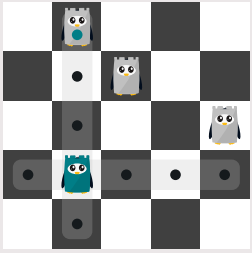
Lösung 38: Von Pseudocode zu Java

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.

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.

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);
}
```

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

6

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:

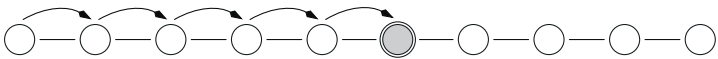
  EINFACH: Durchlaufen Suchraum auf Basis einer Datenstruktur.

  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

- 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 $\mathcal{O}(n)$.



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 gegebene Schlüssel durchsucht und den (ersten) Index des Schlüssels zurückliefert. Wird der Schlüssel nicht gefunden, so soll die Methode einen negativen Wert zurückliefern.

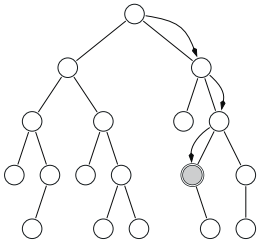
LINEARE SUCHE – LÖSUNG

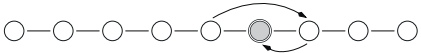
Lösung 40: Lineare Suche

```
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"
}
```

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.





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 gegebene Schlüssel durchsucht und den Index des Schlüssels zurückliefert. Wird der Schlüssel nicht gefunden, so soll die Methode einen beliebigen negativen Integer zurückliefern.

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

- 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

- 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.

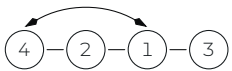
  Divide-and-Conquer: Daten werden aufgeteilt, in Teilen sortiert und dann aus sortierten Teilmengen zusammengefügt. In der Regel rekursiv implementiert.

  Halde: Sortieren mit dafür geeigneten Datenstrukturen wie dem *Heap*. In der Regel rekursiv implementiert.

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).

1. 

2. 

3. 

4. 

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.

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

```
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;
    }
}
```

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.

Schreiben Sie eine Methode `void insertionSort(int[])`, die das übergebene Array mittels Insertionsort aufsteigend sortiert. Dabei darf das Array *nur* durch die übergeben `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;
        while(pos > 0 && arr[pos] <= arr[pos - 1]) {
            swap(arr, pos, pos - 1);
            pos -= 1;
        }
    }
}
```

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;
    }
}
```

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

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 übergeben `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:

```
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;
    }
}
```

Aufgabe 45: Shakersort visualisieren (5 Minuten)

Gegeben sei das folgende Tupel:

$$(13, -15, 9, 8, 23, 8, 0)$$

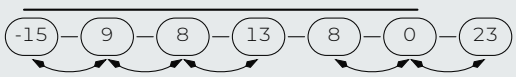
Visualisieren Sie die einzelnen Durchläufe des Shaker-Sort, indem Sie für jeden auf- und für jeden absteigenden Durchlauf nacheinander alle getätigten Vertauschungen und den (aus Sicht des Algorithmus) noch unsortierten Teil des Tupels angeben.

Eine beispielhafte erste Ausgabe für das Tupel $(5, 2, 3, 7, 8, 9, 3)$:

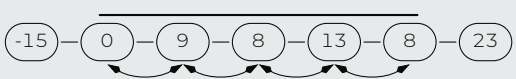
SHAKERSORT VISUALISIEREN – LÖSUNG

Lösung 45: Shakersort visualisieren

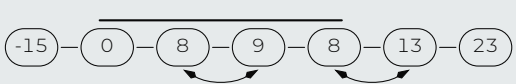
0. 13 — -15 — 9 — 8 — 23 — 8 — 0

1. 

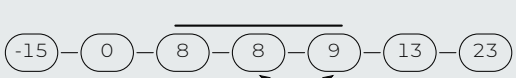
(up)

2. 

(down)

3. 

(up)

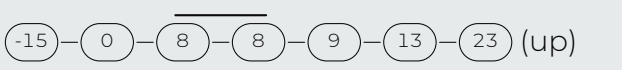
4. 

(down)

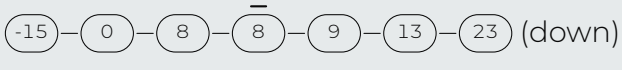
Lösung 45: Shakersort visualisieren (Fortsetzung)

4. 

(down)

5. 

(up)

6. 

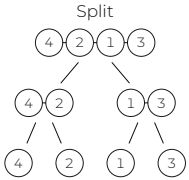
(down)

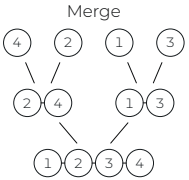
Dieser Schritt kann theoretisch weggelassen werden, wenn man die optimierte Version von Shakersort benutzt. Diese bricht ab, sobald in einem Durchlauf nicht mehr getauscht wird.

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)

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 `cut(l, i, j)` mit folgendem Vertrag gegeben:

Input: Liste $l = (l_1, \ldots, l_n)$, $i,j \in \{1, \ldots, n\}$
Output: Teilliste $(l_i, \ldots, l_j)$, 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 + l`$_i$“ erzeugt eine *neue* Liste $(x_1, \ldots, x_n, l_i)$. Ebenso „`x + l`“ mit $(x_1, \ldots, x_n, l_1, \ldots, l_n)$.

Lösung 46: Mergesort Pseudocode

Algorithm 2: mergesort(t)

Input: Liste $l = (l_1, \ldots, l_n)$
Output: Aufsteigend sortierte Liste l.

```
   // Ein- oder Nullelementig, bereits sortiert:
1  if n ≤ 1 then return l;
2  left := cut(l, 1, ⌊n/2⌋);
3  right := cut(l, ⌊n/2⌋, n);
   // Rekursion
4  left := mergesort(left);
5  right := mergesort(right);
6  return merge(left, right); // Auf der nächsten Folie
```

Lösung 46: Mergesort Pseudocode (Fortsetzung)

Algorithm 3: 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 left_1 ≤ right_1 then
4  |  |  res := res + left_1; left = cut(left, 2, n_left);
5  |  else
6  |  |  res := res + right_1; right = cut(right, 2, n_right);
7  |  end
8  end
9  res := (res + left) + right;  // Es gilt left = () ∨ right = ()
10 return res;
```

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.

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.

Tipp: 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

```java
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);
}
```

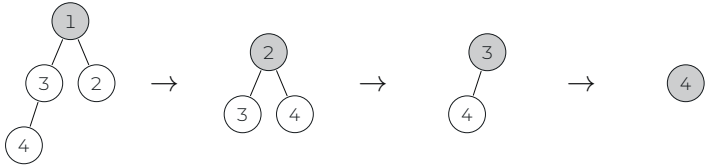
Lösung 47: Quicksort implementieren (Fortsetzung)

```
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;
}
```

Definition 14: Heapsort (rekursiv)

Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente. Sukzessives Entfernen des Wurzelelements (aktuelles Minimum), zur Wiederherstellung der Heap-Eigenschaft.

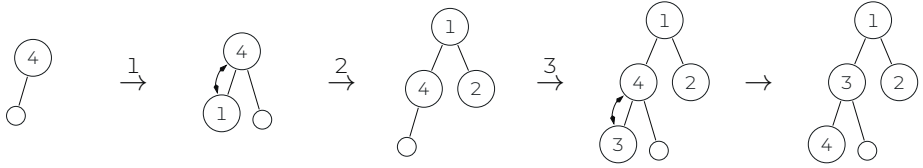
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Die genaue Funktionsweise eines Heaps veranschaulichen wir gleich separat.

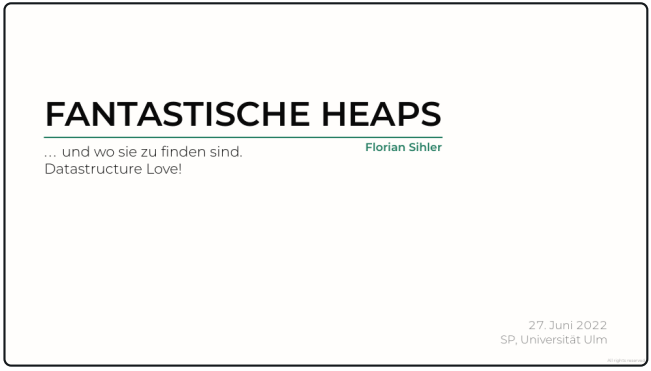


Definition 15: Heap

Binärbaum, mit: Eltern $\leq$ Kinder (*Min-Heap*, für *Max-Heap*: $\geq$).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch ○). 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.





Mehr zu „Heaps“ per Klick…

SORTIERVERFAHREN: ZUSAMMENFASSUNG

	Laufzeit				
Verfahren	best	average	worst	Speicher	Ansatz
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

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	$\Longrightarrow$	24	**I**nga	*Hans sicher vor Inga.*
27	Hansine		27	Hansine	
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).

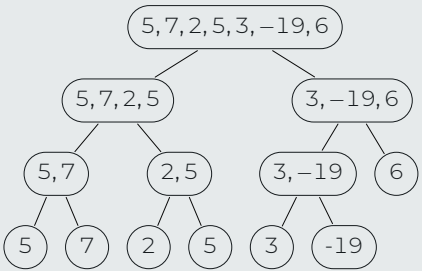
ÜBUNGSAUFGABE

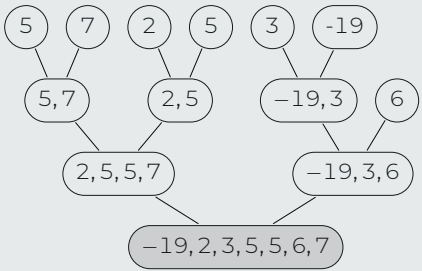
Aufgabe 49: Sortieralgorithmen zeichnen, I (5 Minuten)

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:





Aufgabe 50: Sortieralgorithmen zeichnen, II (5 Minuten)

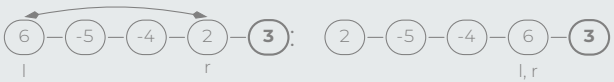
Veranschaulichen Sie den Sortierprozess der Zahlenfolge $3, -5, -4, 2, 6$ unter der Verwendung von Quicksort grafisch. Nutzen Sie das in-place Verfahren durch Vertauschung aus der Vorlesung und sortieren Sie aufsteigend. Geben Sie den Zustand der Folge nach jeder Partitionierung aus und markieren Sie die Partitionen.

Dabei soll stets das erste Element als Pivot gewählt werden.

Lösung 50: Sortieralgorithmen zeichnen, II

Ich führe zusätzlich auf, was mit den `l`- und `r`-Markern geschieht:

1. **3** – -5 – -4 – 2 – 6 Tausch ans Ende: 6 – -5 – -4 – 2 – **3**

   Tausche *l* und Pivot: 2 – -5 – -4 – **3** – 6

2. Links: **2** – -5 – -4 (3) – (6) Tausch ans Ende: -4 – -5 – **2** (3) – (6)

   Tausche *l* und Pivot: -4 – -5 – **2** (3) – (6)

Lösung 50: Sortieralgorithmen zeichnen, II (Fortsetzung)

3. Links: -4 – -5 2 3 – 6 Tausch ans Ende: -5 – -4 2 3 – 6

   -5 – -4 2 3 – 6
   r l

   Tausche l und Pivot: -5 – -4 2 3 – 6

4. Rechts in -4 – -5 – 2 3 – 6 ist leer. Vertauschungen ändern nichts.

5. Rechts in 2 – -5 – -4 – 3 – 6 besteht mit 6 nur aus einem Element. Vertauschungen ändern nichts.

Finales Ergebnis: -5 – -4 – 2 – 3 – 6

Dynamische Datenstrukturen

○ ○ ○ ○ ○ ○ 7 ○

Florian Sihler, 2.01.2022

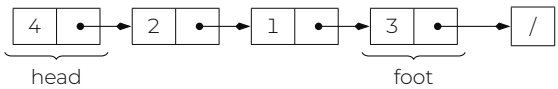
LISTEN – DYNAMISCHE ARRAYS

- 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 Löschoperationen 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

- 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*.

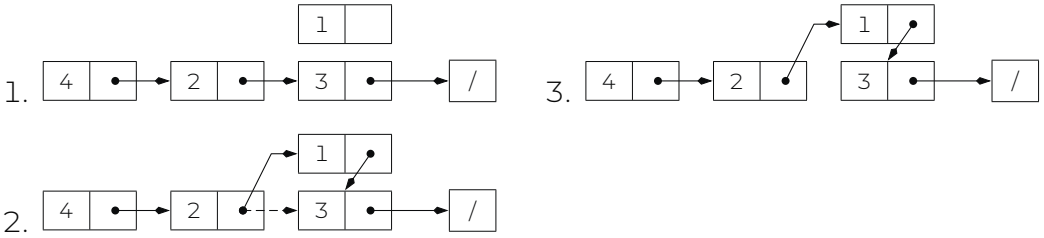
```
class Element {
    public int value;
    public Element next;
    // ...
}
```



- Doppelt verkettete Listen verbrauchen mehr Speicher und sind komplizierter (da zwei Verweise).

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketten 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:

```
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:

```
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)

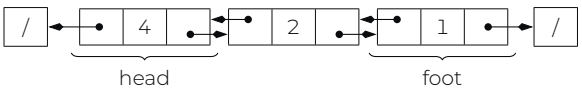
Wir suchen das Ende der Liste:

```
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

- 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

- 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

- 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.

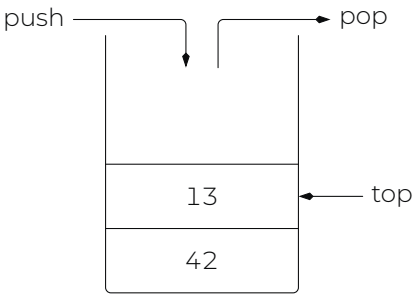
DATENSTRUKTUR: QUEUE

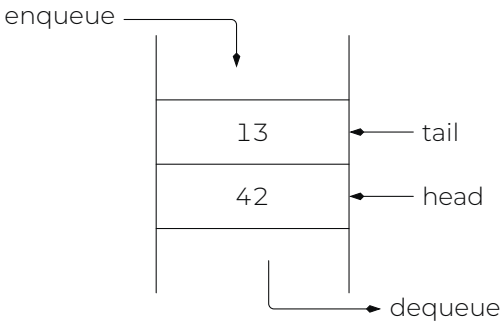
- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).
- Elemente können nur hinten in der Schlange eingereiht und nur vorne von der Schlange entfernen werden.
- Queues können zum Puffern verwendet werden (analog zum Schalter).
- Es existieren Derivate, wie die *double-ended queue* (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

- 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





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.

BÄUME: MATHEMATISCH

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.

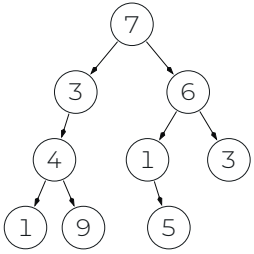
- Die Kanten können ungerichtet oder gerichtet („gewurzelter Baum“) sein.
- Ein Baum lässt sich auch als rekursive Datenstruktur auffassen.
- Hier ist jeder Knoten gleichzeitig die „Wurzel“ eines Teilbaumes der entweder leer sein ($\widehat{=}$ Blatt) oder noch weitere Knoten enthalten kann ($\widehat{=}$ 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.
- 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

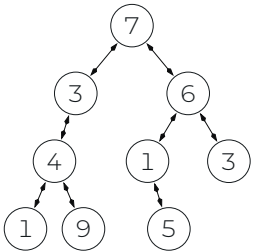
```
class Node {
    public int value;
    public Node left;
    public Node right;
    // ...
}
```



- 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.

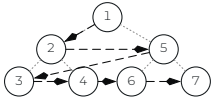
BÄUME: IMPLEMENTATION, II

```
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



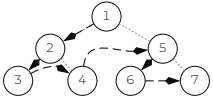
Breitendurchlauf

Ebenen

Tiefendurchlauf

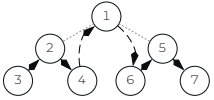
Pre-Order

root, left, right



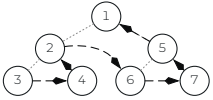
In-Order

left, root, right



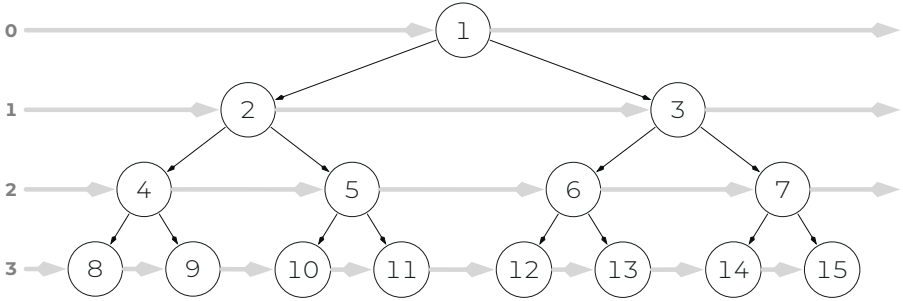
Post-Order

left, right, root



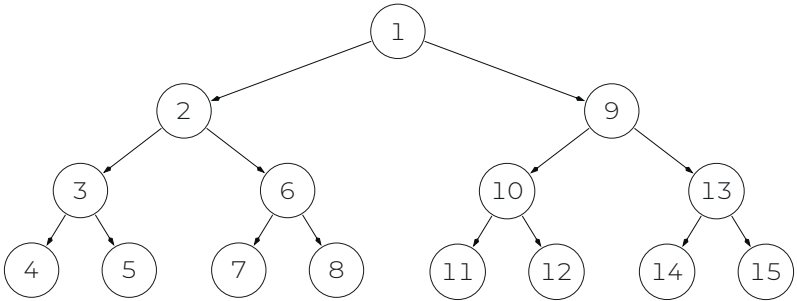
TRAVERSIERUNG: LEVEL-ORDER

- 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*).



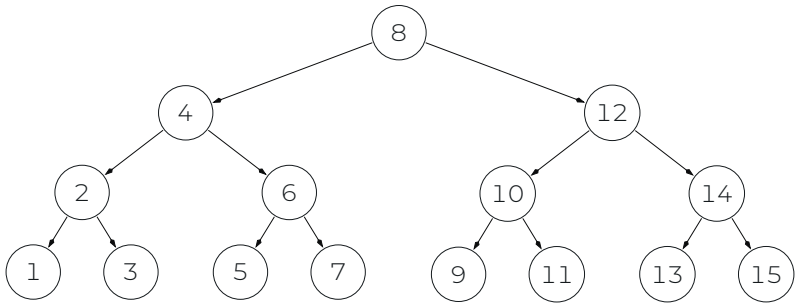
TRAVERSIERUNG: PRE-ORDER

- 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.



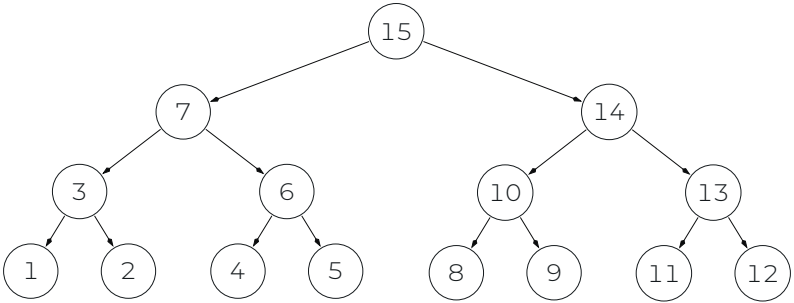
TRAVERSIERUNG: IN-ORDER

- 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

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit der Post-Order Strategie werden erst rekursiv beide Teilbäume und dann der Knoten selbst bearbeitet.



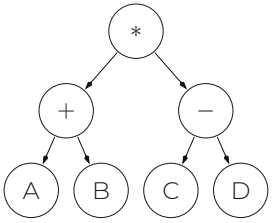
Mehr zu „Traversierungen“ per Klick…

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 ⇒ binäre Suche.)
- Vergleiche hierzu auch die Datenstruktur Heap.
- Die Verfahren lassen sich auch auf Bäume mit höherem Verzweigungsgrad anwenden!

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 Postifx-Notation: $AB + CD - *$.

- Im Gegensatz zu Bäumen, können Knoten in Graphen auch mehrere Vorgänger haben.

Definition 18: Ungerichter Graph

Ist ein Tupel $G = (V, E)$ aus Knoten V und Kanten $E \subseteq \binom{V}{2}$.

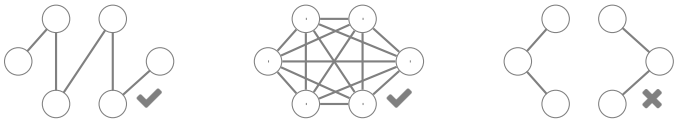
- Die Graphdefinition gleicht der aus „Formale Grundlagen“. Sie wird hier zusammengefasst.

Definition 19: Gerichteter Graph

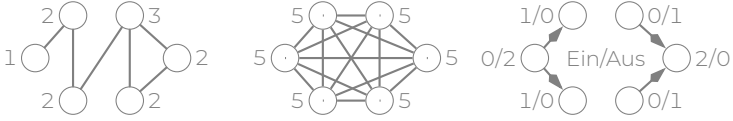
Ein gerichteter Graph definiert sich ähnlich zum ungerichteten, definiert die Kanten aber als Tupel $E \subseteq V \times V$.

BEGRIFFE IN EINEM GRAPHEN

ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.

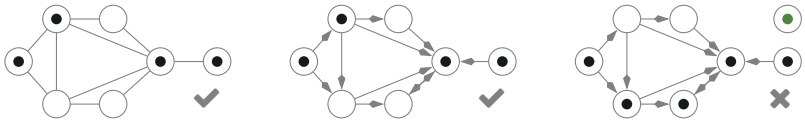


GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.

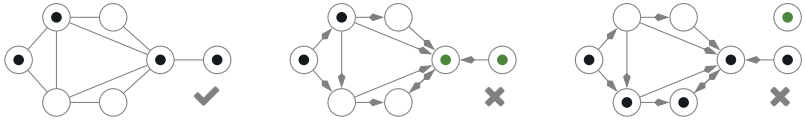


BEGRIFFE IN EINEM GRAPHEN, II

Weg: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.

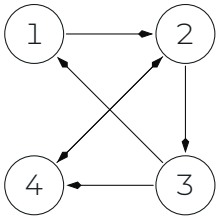


Pfad: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.



VARIANTEN VON GRAPHEN

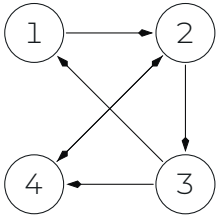
- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \to M$ einen Wert aus einer Menge M zu. (Beispielsweise $M = \mathbb{N}$)
- Wir können einen Graphen durch eine *Adjazenzmatrix* beschreiben. Hierbei geben die jeweiligen Zellen $m_{i,j}$, ob eine Verbindung vom Knoten i zum Knoten j besteht ($m_{i,j} > 0$ für eine Kante).



$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

ADJAZENZLISTEN

- Adjazenzlisten sind eine andere Variante Graphen zu repräsentieren.
- Hier hält jeder Knoten eine (verkettete) Liste an seinen benachbarten Knoten.



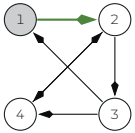
1. $\rightarrow 2$
2. $\rightarrow 3 \rightarrow 4$
3. $\rightarrow 1 \rightarrow 4$
4. $\rightarrow 2$

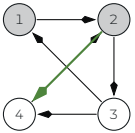
ALGORITHMEN AUF GRAPHEN

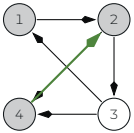
- Eine gängiges Problem ist es herauszufinden ob ein Graph Zyklen enthält und/oder ob er zusammenhängend ist.
- Wegfindungsproblem wie das kürzeste Wege Probleme (Routenfindung, …), Traveling Salesman Problem.
- Finden des minimalen, aufspannenden Baum.
- Exemplarisch betrachtet die Vorlesung die Breiten- und die Tiefensuche.

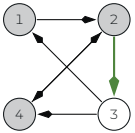
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

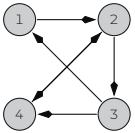
- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor zurück und probiert die Ansätze dort erneut.

1. 

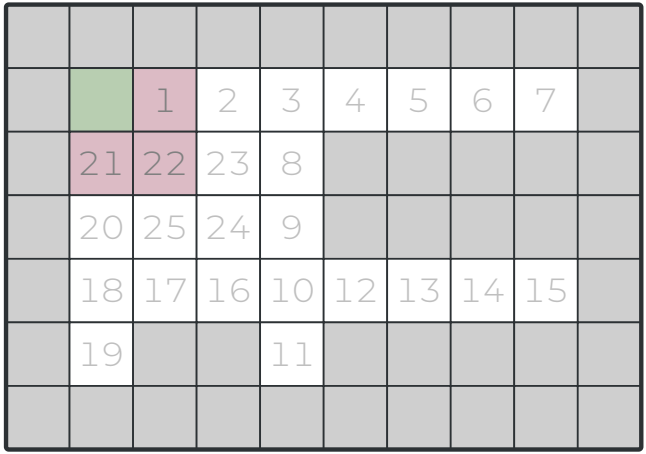
2. 

3. 

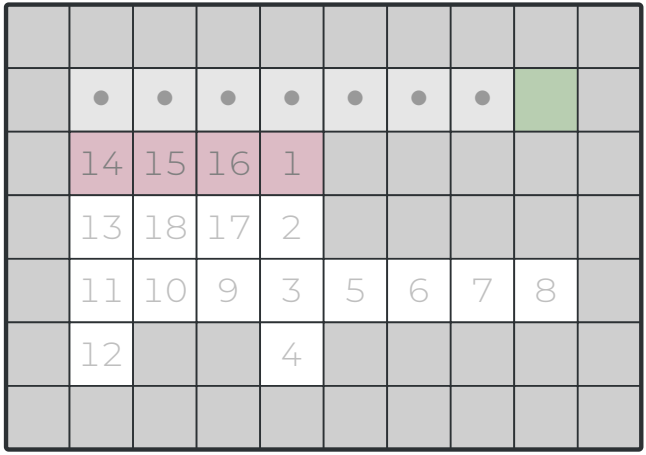
4. 

5. 

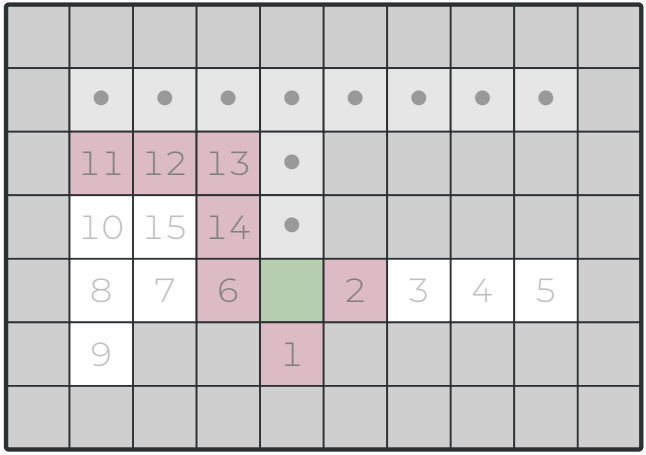
EIN BEISPIEL FÜR DIE TIEFENSUCHE



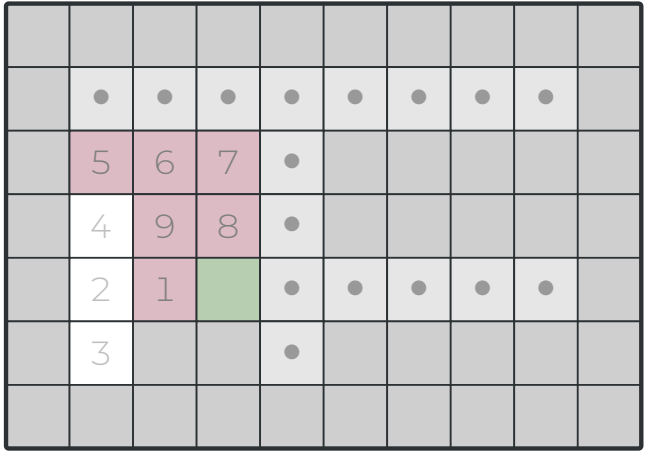
EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



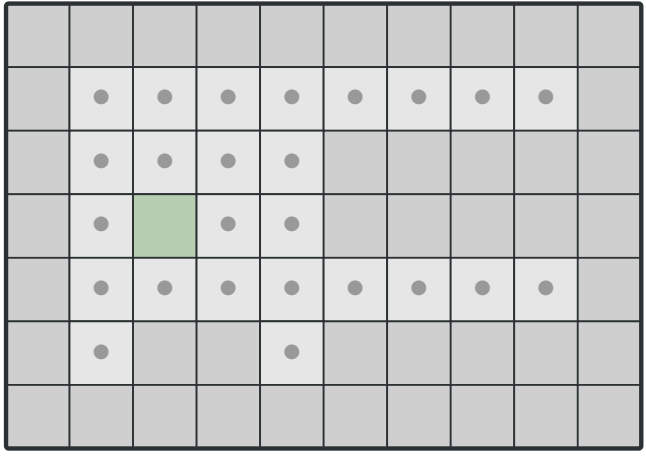
TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE TIEFENSUCHE



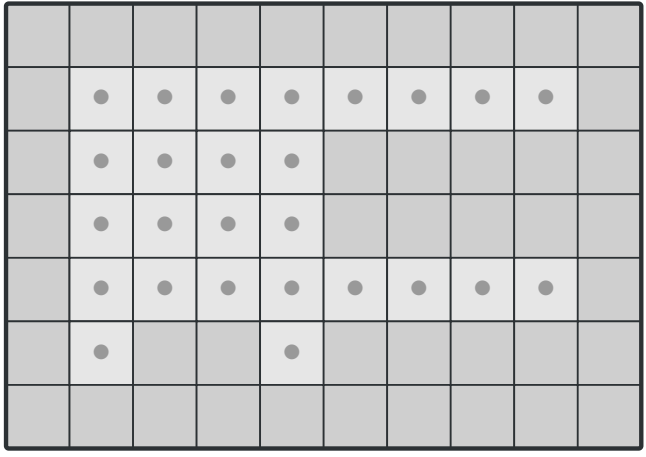
TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Sihler

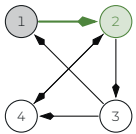
13. Jul 2022
SP, Universität Ulm

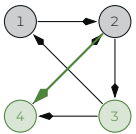
EIN BEISPIEL FÜR DIE TIEFENSUCHE

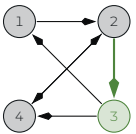


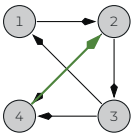
GRAPHEN DURCHSUCHEN: BREITENSUCHE

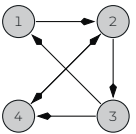
- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.

1. 

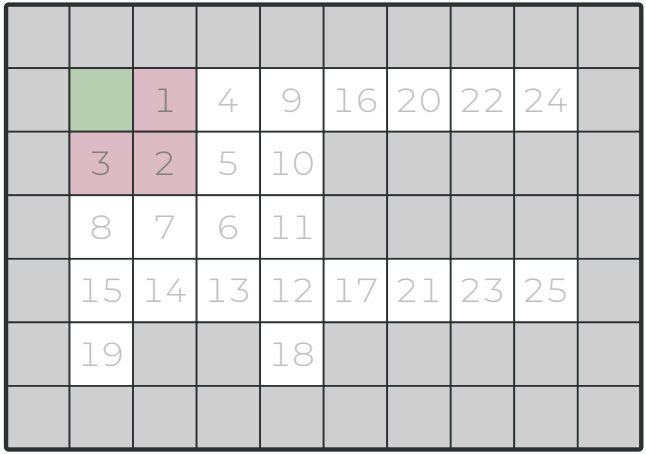
2. 

3. 

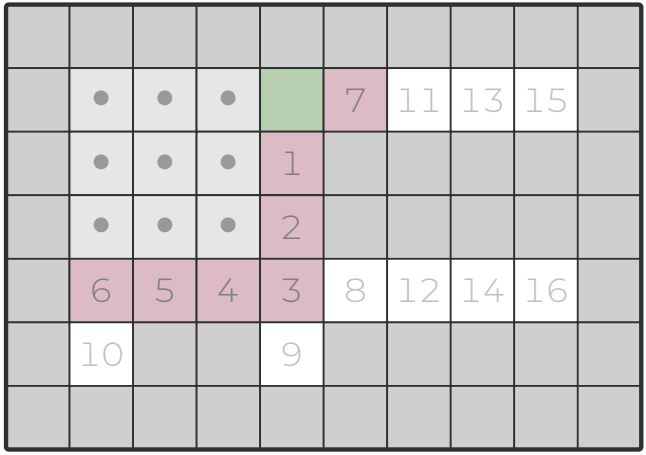
4. 

5. 

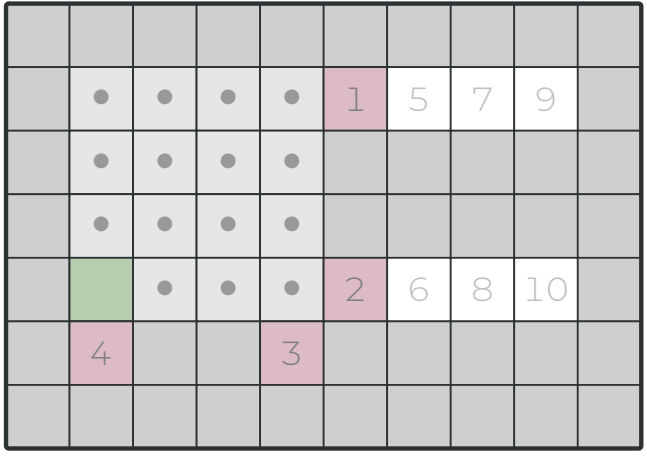
EIN BEISPIEL FÜR DIE BREITENSUCHE



EIN BEISPIEL FÜR DIE BREITENSUCHE



EIN BEISPIEL FÜR DIE BREITENSUCHE



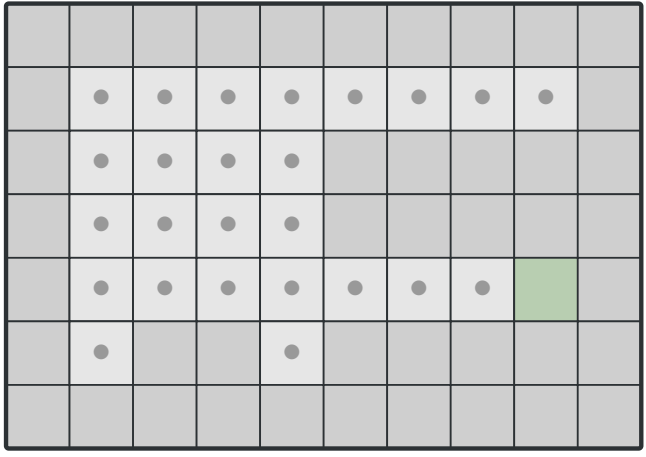
TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



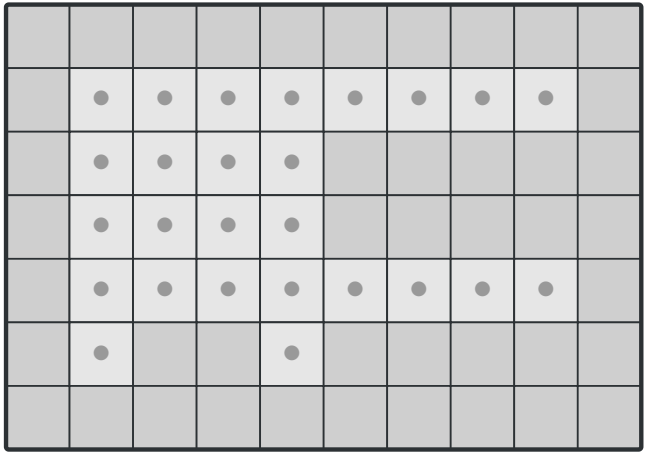
TRAVERSIERUNGSVARIANTEN

Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.

Florian Sihler

13. Jul 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 Sihler

13. Jul 2022
SP, Universität Ulm

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()

```
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;
   }
}
```

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.

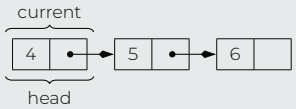
Lösung 53: Eine Liste umdrehen

```
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…

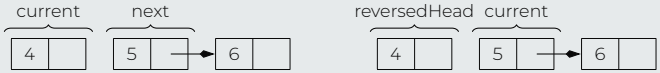
Lösung 53: Eine Liste umdrehen (Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste 4 → 5 → 6.



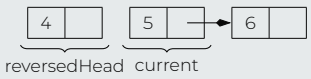

```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

1. Da `reversedHead = null`:



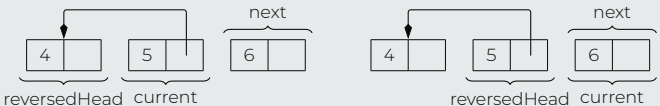
Lösung 53: Eine Liste umdrehen (Fortsetzung)

Betrachten wir die Vorgehensweise für die Liste $4 \to 5 \to 6$.




```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

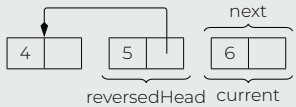
2. Nun zeigt `current.next` auf das vorherige Element:



Wie ein Schiffchen schieben wir also `reversedHead`, `current` *und* `next` *nach rechts durch die Liste, um aus* `current` → `next` *die Kante* `current` → `reversedHead` *zu machen.*

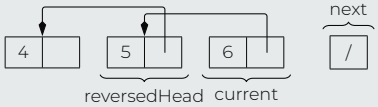
Lösung 53: Eine Liste umdrehen (Fortsetzung)

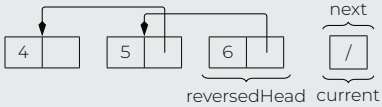
Betrachten wir die Vorgehensweise für die Liste $4 \rightarrow 5 \rightarrow 6$.




```
Element reversedHead = null;
Element current = head;
while (current != null) {
    Element next = current.next;
    current.next = reversedHead;
    reversedHead = current;
    current = next;
}
```

3. **`reversedHead`** und **`current`** wandern durch die Liste:





Aufgabe 54: Theorie zu Stapel und Warteschlange (5 Minuten)

Bewerten (wahr/falsch) und Begründen Sie jeweils:

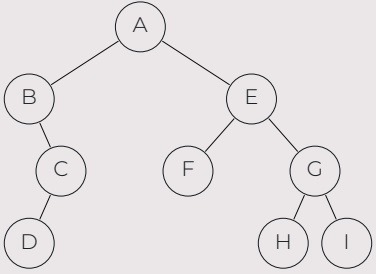
1. Eine Warteschlange lässt sich auch dann implementieren, wenn nur die Datenstruktur *Stack* existiert.
2. Stacks folgen dem FIFO-Prinzip.
3. Der Java-(Objekt-)Heap wird auch als (Datenstruktur) *Heap* verwaltet.

Lösung 54: Theorie zu Stapel und Warteschlange

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`).

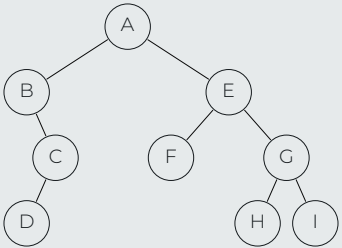
Aufgabe 55: Traversierung eines Baumes (4 Minuten)

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



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: $D, C, B, F, H, I, G, E, A$.

Aufgabe 56: Baum durch Traversierung (4 Minuten)

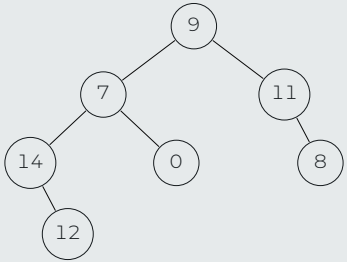
Gegeben seien die beiden folgenden Traversierung des selben Binärbaums:

- 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

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…

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

```java
void inorder(Node n) {
    if (n == null) return;

    inorder(n.left);
    System.out.println(n.value);
    inorder(n.right);
}
```

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

```
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
   }
}
```

Aufgabe 59: Breitensuche - Pseudocode (6 Minuten)

Schreiben Sie Pseudocode, der den folgenden Vertrag durch eine Breitensuche erfüllt:

In: Ajazenzmatrix a für $n \in \mathbb{N}$ Knoten mit $n > 0$.
In: Startknoten $s \in \{0, \ldots, n-1\}$ und gesuchter Knoten $g \in \{0, \ldots, n-1\}$.
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

```
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 a_{current,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“;
```

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 a_{cur,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

```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

- 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.
- 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.

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. 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

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

```java
class Lebewesen {
    int alter;
    float grosse;

    Lebewesen(int a, float g) {
        alter = a;
        groesse = g;
    }
}
```

```java
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.

```java
class Student extends Mensch {
   long matrikelnummer;

   Student(int a, float g, String n,
         long m) {
      super(a, g, n);
      matrikelnummer = m;
   }
}
```

```java
class Dozent extends Mensch {
   String vorlesung;

   Dozent(int a, float g, String n,
         String v) {
      super(a, g, n);
      vorlesung = v;
   }
}
```

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`.

KOMMENTARE

- 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

- 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

- 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

```
abstract class AbstractList {

    // Für 'super()'
    public AbstractList() { }

    abstract void add(Element e);

    int size() {
        // ...
    }
    // ...
}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
class ArrayList extends AbstractList {
    Element[] elems;

    public ArrayList() {
        super();
        elems = new Element[0];
    }

    public void add(Element e) {
        // ...
    }
    // ...
}
```

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:

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

- 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

- 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

- 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 werdenn 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

- 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

- 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

- Um einen Fehler zu behandeln, bietet Java das **try-catch**-Konstrukt

```
try {
    // Anweisungen, Methodenaufrufe, ...
} catch(⟨FehlerKlasseA⟩ ⟨Bezeichner⟩) {
    // Behandle Fehler der Klasse ⟨FehlerKlasseA⟩
} catch(⟨FehlerKlasseB⟩ ⟨Bezeichner⟩) {
    // Behandle Fehler der Klasse ⟨FehlerKlasseB⟩
} finally {
    // Anweisungen die immer ausgeführt werden.
}
```

- Das **finally** ist optional, ebenso kann nur ein **catch**-Block auftreten.

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

- 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

- 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();
}
```

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.
   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.
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.

Hier hilft auch Aufgabe 64.

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.

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␣aaaam␣("+ 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

```java
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;
   }
}
```

```java
class Enemy extends Entity {
   protected int hp, ap;
   public Enemy(long id) { super(id); }

   Enemy whoIsEvil() {
      System.out.println("I␣aaaam␣("+ id +")");
      return this;
   }
}

class Companion extends Entity {
  public Companion(long id) { super(id); }
}
```

```java
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 +")");
   }
}
```

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, 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).

Aufgabe 67: Fehler finden, VI (4 Minuten)

Finden und erklären Sie alle (Kompilier-)Fehler:

```
public class MyL1ttl3Cl4ss extends MyL1ttl3Cl4ss.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 MyL1ttl3Cl4ss extends MyL1ttl3Cl4ss.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. `MyL1ttl3Cl4ss extends MyL1ttl3Cl4ss.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: `MyL1ttl3Cl4ss.A` statisch zu machen, hilft nicht. Java Liebe!

WEITERES — DIE WEIHNACHTSWIEDERHOLUNG

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. *LaTeX-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