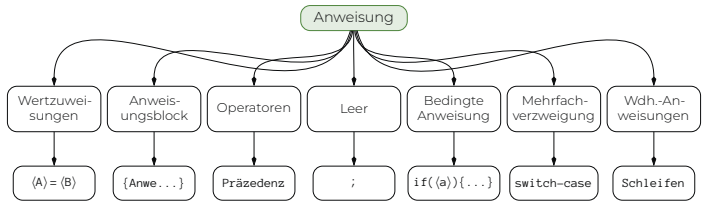
EIDI-KOMPAKT

Kurze Version — 2.01.2022



Florian Sihler
Verteilte Systeme — Universität Ulm

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?

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

WAS IST EIN ALGORITHMUS?

Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

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

Textuell

WAS IST EIN ALGORITHMUS?

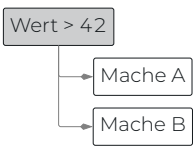
Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

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

Textuell



Grafisch

WAS IST EIN ALGORITHMUS?

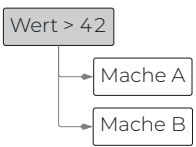
Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

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

Textuell



Grafisch

```
1 Ist Wert > 42 tue:
2 |   Mache A;
3 sonst:
4 |   Mache B;
```

Pseudocode

WAS IST EIN ALGORITHMUS?

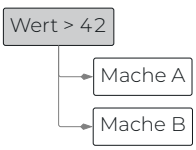
Definition 1: Algorithmus

Eine *eindeutige* Handlungsvorschrift zur Lösung eines Problems.

- Es existieren verschiedene Darstellungsformen:

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

Textuell



Grafisch

```
1 Ist Wert > 42 tue:
2 |   Mache A;
3 sonst:
4 |   Mache B;
```

Pseudocode

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

ALGORITHMUSEIGENSCHAFTEN

ALGORITHMUSEIGENSCHAFTEN

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

ALGORITHMUSEIGENSCHAFTEN

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

  Prozess: Die Ausführung der Schritte eines Algorithmus.

ALGORITHMUSEIGENSCHAFTEN

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

  Prozess: Die Ausführung der Schritte eines Algorithmus.

  Prozessor: Der Ausführende (Mensch, Computer, …).

ALGORITHMUSEIGENSCHAFTEN

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

  Prozess: Die Ausführung der Schritte eines Algorithmus.

  Prozessor: Der Ausführende (Mensch, Computer, …).

  Elementaroperation: Eine einzelne, eindeutige Handlung.

ALGORITHMUSEIGENSCHAFTEN

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

  Prozess: Die Ausführung der Schritte eines Algorithmus.

  Prozessor: Der Ausführende (Mensch, Computer, …).

  Elementaroperation: Eine einzelne, eindeutige Handlung.

- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:

ALGORITHMUSEIGENSCHAFTEN

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

  Prozess: Die Ausführung der Schritte eines Algorithmus.

  Prozessor: Der Ausführende (Mensch, Computer, …).

  Elementaroperation: Eine einzelne, eindeutige Handlung.

- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:

  Ausführbarkeit: Die Anleitung muss ausführ- und reproduzierbar sein.

ALGORITHMUSEIGENSCHAFTEN

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

  Prozess: Die Ausführung der Schritte eines Algorithmus.

  Prozessor: Der Ausführende (Mensch, Computer, …).

  Elementaroperation: Eine einzelne, eindeutige Handlung.

- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:

  Ausführbarkeit: Die Anleitung muss ausführ- und reproduzierbar sein.

  Endlichkeit: Er muss mit endlich viel Text beschreibbar sein.

ALGORITHMUSEIGENSCHAFTEN

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

  Prozess: Die Ausführung der Schritte eines Algorithmus.

  Prozessor: Der Ausführende (Mensch, Computer, …).

  Elementaroperation: Eine einzelne, eindeutige Handlung.

- Ein Algorithmus besitzt einige grundlegende **Eigenschaften**:

  Ausführbarkeit: Die Anleitung muss ausführ- und reproduzierbar sein.

  Endlichkeit: Er muss mit endlich viel Text beschreibbar sein.

  Verarbeitung: Ein Algorithmus erhält Eingabedaten E, hält lokale Daten D und erzeugt Ausgabedaten A.

EIGENSCHAFTEN FÜR DIE ANALYSE

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

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

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

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

Partielle Korrektheit

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

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

Partielle Korrektheit

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

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

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

Partielle Korrektheit

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

Totale Korrektheit

EIGENSCHAFTEN FÜR DIE ANALYSE

Termination

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

Partielle Korrektheit

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

Totale Korrektheit

Der Algorithmus terminiert und ist partiell korrekt.

EIGENSCHAFTEN FÜR DIE ANALYSE, II

EIGENSCHAFTEN FÜR DIE ANALYSE, II

Determinismus

EIGENSCHAFTEN FÜR DIE ANALYSE, II

Determinismus

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

EIGENSCHAFTEN FÜR DIE ANALYSE, II

Determinismus

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

Determiniertheit

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.

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!

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.

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:

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!

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*,

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

SCHEMA EINES ALGORITHMUS

SCHEMA EINES ALGORITHMUS

- Ein Algorithmus benötigt (formal):

SCHEMA EINES ALGORITHMUS

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

SCHEMA EINES ALGORITHMUS

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

SCHEMA EINES ALGORITHMUS

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

- Algorithmen fundieren oft auf einem Problem der realen Welt und bedienen sich Mechaniken, wie der Abstraktion, zur Lösung.

SCHEMA EINES ALGORITHMUS

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

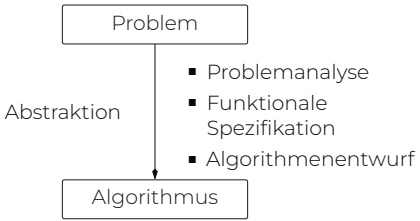
SCHEMA EINES ALGORITHMUS

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

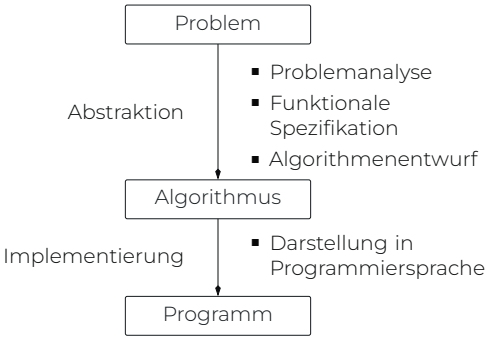
SCHEMA EINES ALGORITHMUS, II

SCHEMA EINES ALGORITHMUS, II

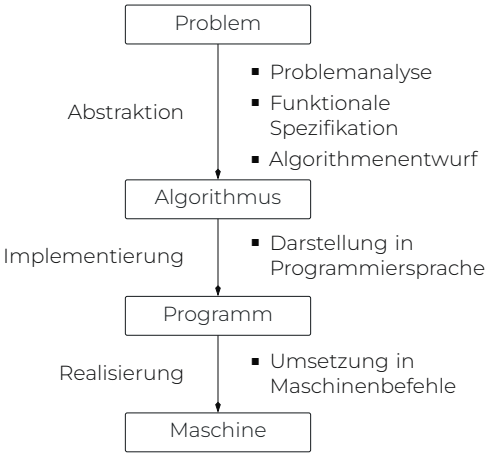
SCHEMA EINES ALGORITHMUS, II



SCHEMA EINES ALGORITHMUS, II



SCHEMA EINES ALGORITHMUS, II



WIE EIN PROGRAMM ENTSTEHT

WIE EIN PROGRAMM ENTSTEHT

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

WIE EIN PROGRAMM ENTSTEHT

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

WIE EIN PROGRAMM ENTSTEHT

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

WIE EIN PROGRAMM ENTSTEHT

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

WIE EIN PROGRAMM ENTSTEHT

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

WIE EIN PROGRAMM ENTSTEHT

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

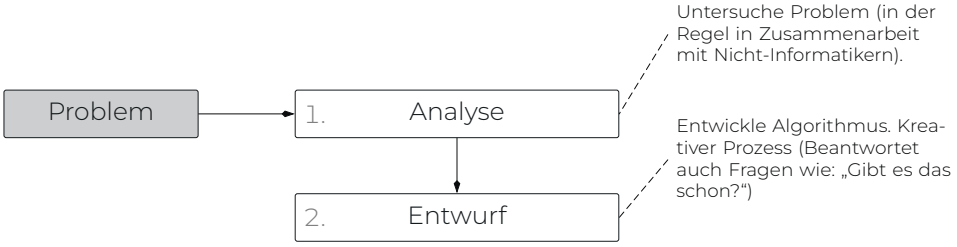
WIE EIN PROGRAMM ENTSTEHT, II

WIE EIN PROGRAMM ENTSTEHT, II

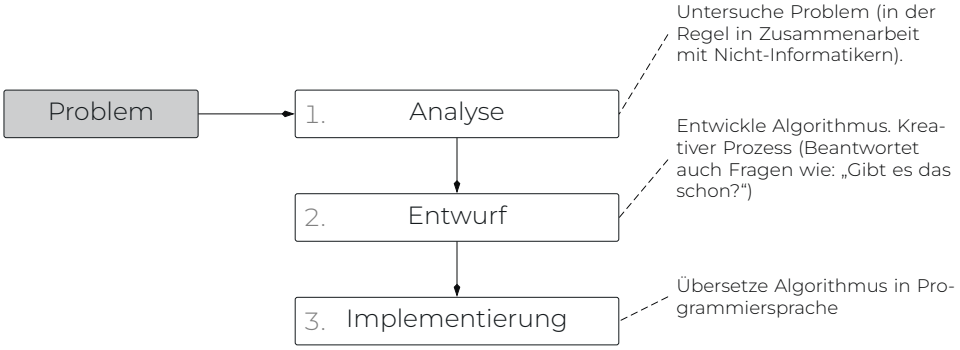
Problem

WIE EIN PROGRAMM ENTSTEHT, II

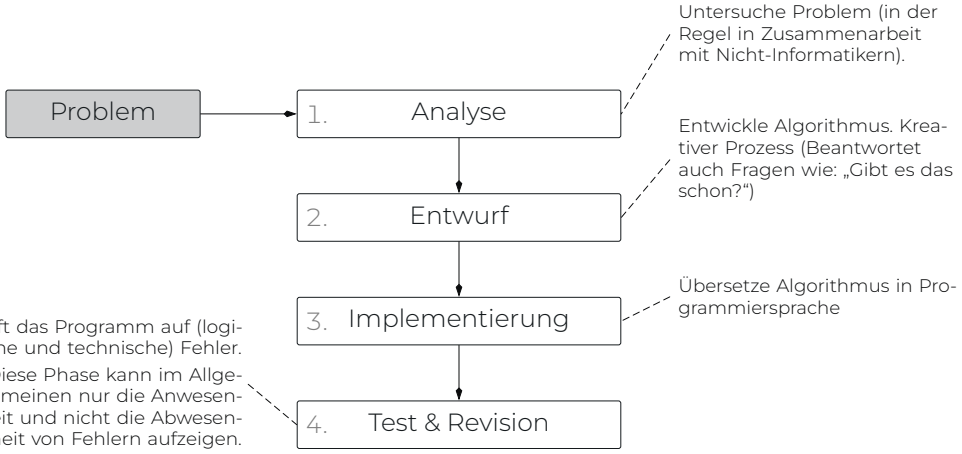
WIE EIN PROGRAMM ENTSTEHT, II



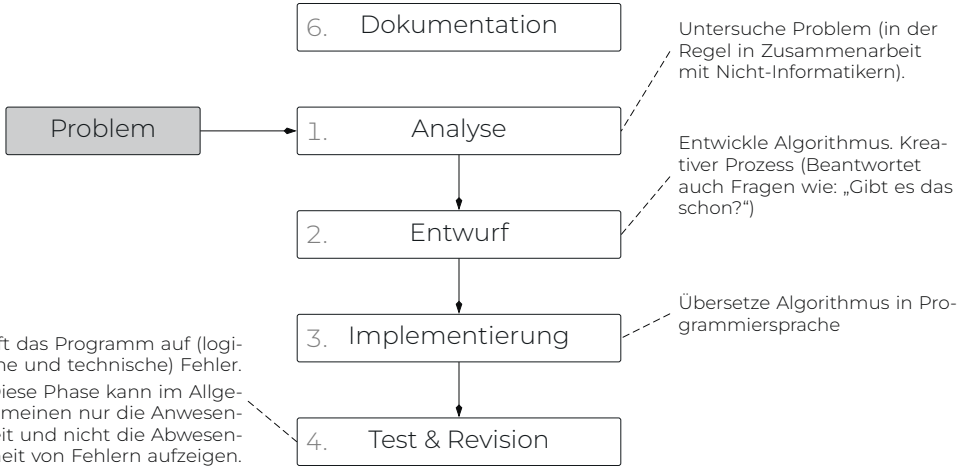
WIE EIN PROGRAMM ENTSTEHT, II



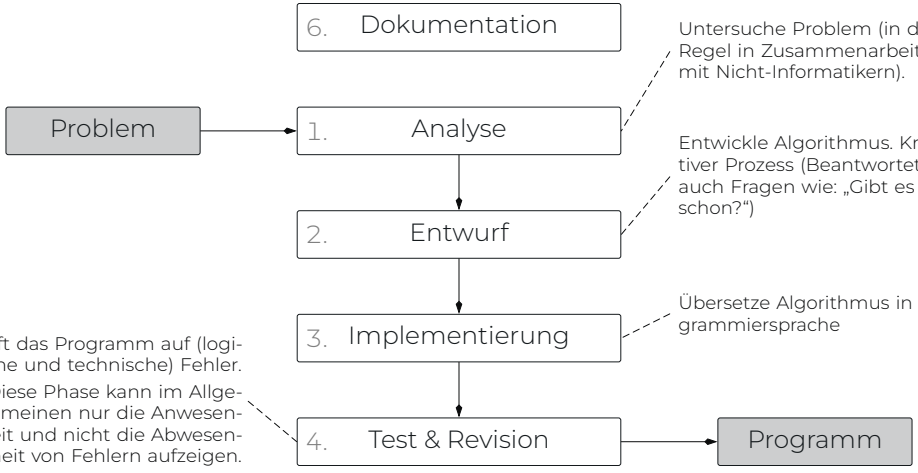
WIE EIN PROGRAMM ENTSTEHT, II



WIE EIN PROGRAMM ENTSTEHT, II



WIE EIN PROGRAMM ENTSTEHT, II



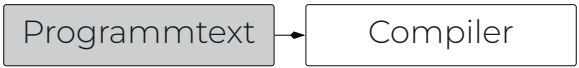
Java-Basics

VOM TEXT ZUM PROGRAMM

VOM TEXT ZUM PROGRAMM

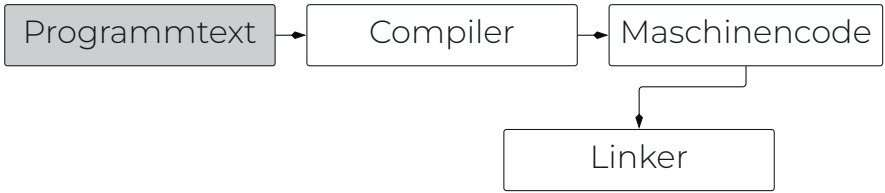
Programmtext

VOM TEXT ZUM PROGRAMM

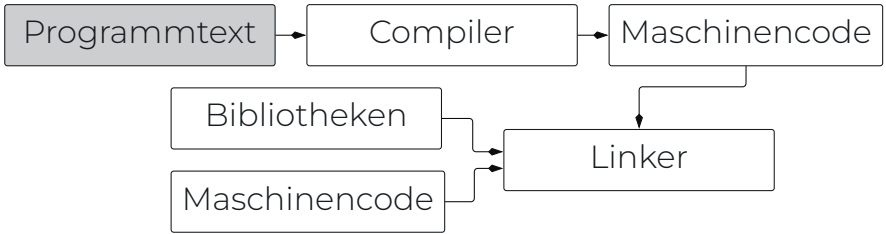


VOM TEXT ZUM PROGRAMM

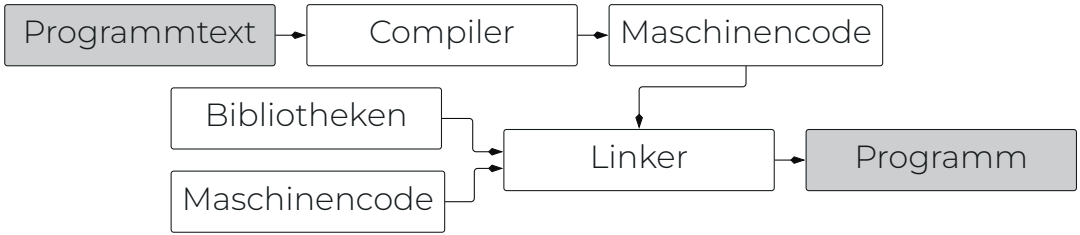
VOM TEXT ZUM PROGRAMM



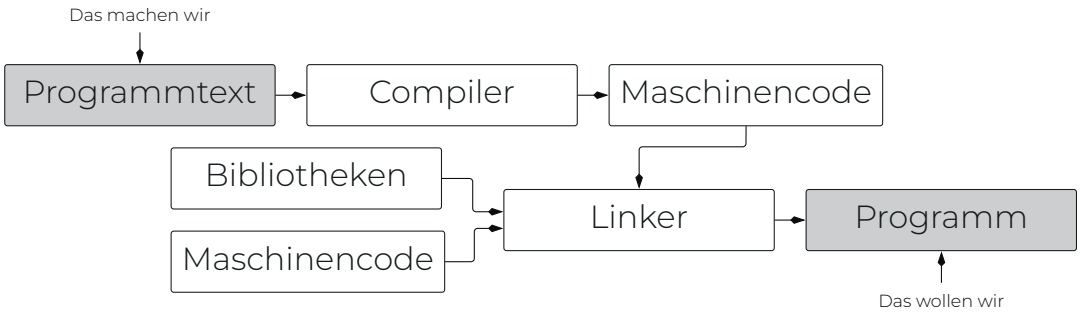
VOM TEXT ZUM PROGRAMM



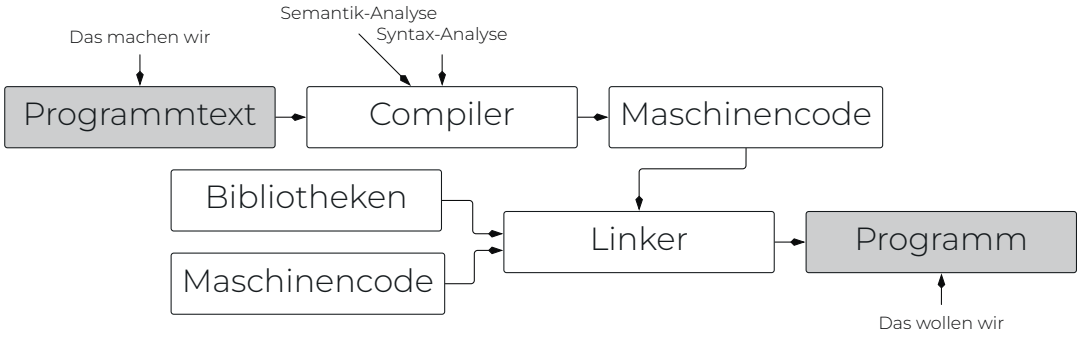
VOM TEXT ZUM PROGRAMM



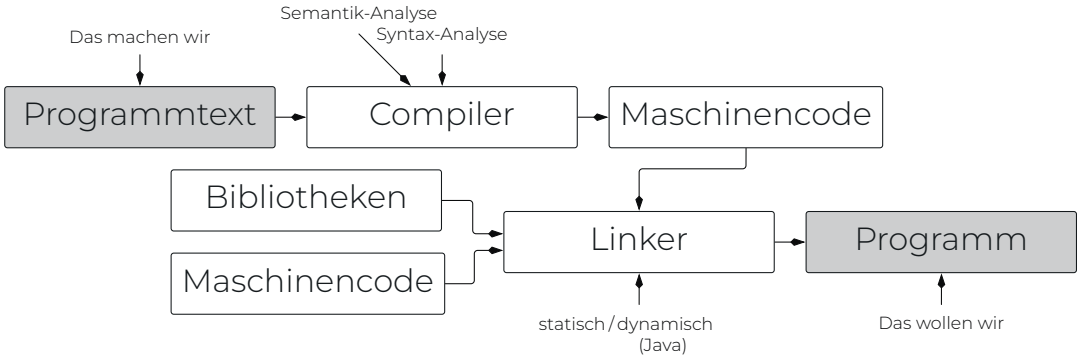
VOM TEXT ZUM PROGRAMM



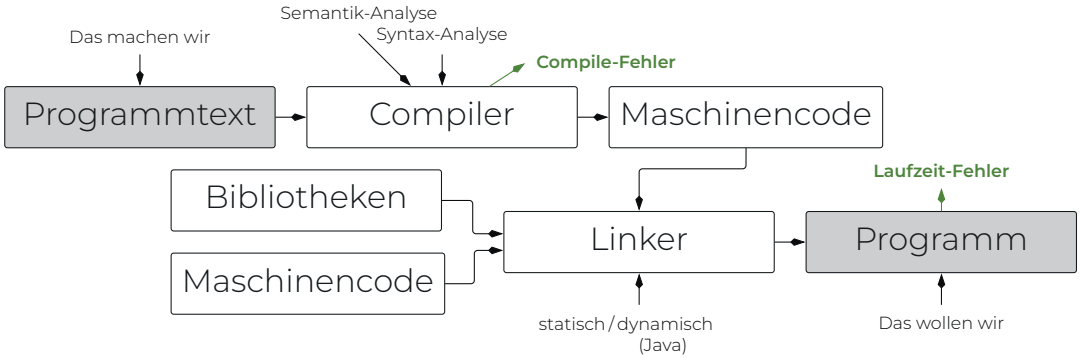
VOM TEXT ZUM PROGRAMM



VOM TEXT ZUM PROGRAMM



VOM TEXT ZUM PROGRAMM



VOM TEXT ZUM PROGRAMM, II

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

VOM TEXT ZUM PROGRAMM, II

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

GÜLTIGE JAVA-BEZEICHNER

GÜLTIGE JAVA-BEZEICHNER

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

GÜLTIGE JAVA-BEZEICHNER

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

GÜLTIGE JAVA-BEZEICHNER

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

GÜLTIGE JAVA-BEZEICHNER

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

VARIABLEN UND WERTZUWEISUNGEN

VARIABLEN UND WERTZUWEISUNGEN

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

VARIABLEN UND WERTZUWEISUNGEN

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

- Java ist eine *streng typisierte* Sprache.

VARIABLEN UND WERTZUWEISUNGEN

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

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

VARIABLEN UND WERTZUWEISUNGEN

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

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

VARIABLEN UND WERTZUWEISUNGEN

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

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

  PRIMITIVE: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`

  Hinweis: Stack, *call-by-value*

VARIABLEN UND WERTZUWEISUNGEN

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

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

  PRIMITIVE: `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `boolean`
  Hinweis: Stack, *call-by-value*

  KOMPLEXE: `String`, `Random`, …
  Hinweis: Heap, „*call-by-reference*“

KONVERTIERUNG PRIMITIVER DATENTYPEN

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren.

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

KONVERTIERUNG PRIMITIVER DATENTYPEN

- Java kann manche Datentypen implizit Konvertieren. Es gilt:

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

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:

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:

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

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

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).

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.

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.**

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 $(\langle \texttt{Datentyp} \rangle)\langle \texttt{Ausdruck} \rangle$ eine explizite Konvertierung auch in umgekehrte Richtung erfolgen.

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`

KONSTANTEN UND HINWEISE

KONSTANTEN UND HINWEISE

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

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

KONSTANTEN UND HINWEISE

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

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

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

KONSTANTEN UND HINWEISE

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

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

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

KONSTANTEN UND HINWEISE

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

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

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

KONSTANTEN UND HINWEISE

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

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

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

- Zeichen (`char`) werden in Java im UTF-16 Format gespeichert.

KONSTANTEN UND HINWEISE

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

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

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

- Zeichen (`char`) werden in Java im UTF-16 Format gespeichert. Für die unteren 7-Bit ist es identisch zur ASCII-Kodierung.

KONSTANTEN UND HINWEISE

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

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

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

- Zeichen (`char`) werden in Java im UTF-16 Format gespeichert. Für die unteren 7-Bit ist es identisch zur ASCII-Kodierung.

- Der „Datentyp“ `void` gibt bei Methoden an,

KONSTANTEN UND HINWEISE

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

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

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

- Zeichen (`char`) werden in Java im UTF-16 Format gespeichert. Für die unteren 7-Bit ist es identisch zur ASCII-Kodierung.

- Der „Datentyp“ `void` gibt bei Methoden an, dass diese nichts zurückgeben!

PRÄZEDENZREGELN

PRÄZEDENZREGELN

- Operationen werden in Java in einer gewissen Reihenfolge ausgeführt.

PRÄZEDENZREGELN

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

PRÄZEDENZREGELN

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

$$\left[a\texttt{++}, a\texttt{--}\right]$$

PRÄZEDENZREGELN

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

$$\left[a\texttt{++}, a\texttt{--}\right] \rightarrow \left[\texttt{!}a, -a, \texttt{++}a, \texttt{--}a\right]$$

PRÄZEDENZREGELN

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

$$\left[a\texttt{++}, a\texttt{--}\right] \rightarrow \left[\texttt{!}a, -a, \texttt{++}a, \texttt{--}a\right] \rightarrow \left[*, /, \%\right]$$

PRÄZEDENZREGELN

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

$$\left[a\texttt{++}, a\texttt{--}\right] \rightarrow \left[\texttt{!}a, -a, \texttt{++}a, \texttt{--}a\right] \rightarrow \left[*, /, \texttt{\%}\right] \rightarrow \left[a + b, a - b\right]$$

PRÄZEDENZREGELN

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

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

PRÄZEDENZREGELN

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

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

PRÄZEDENZREGELN

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

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

PRÄZEDENZREGELN – KOMMENTAR

PRÄZEDENZREGELN – KOMMENTAR

- Java wendet die mathematischen Rechenregeln wie bekannt an.

PRÄZEDENZREGELN – KOMMENTAR

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

PRÄZEDENZREGELN – KOMMENTAR

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

STANDARDWERTE

STANDARDWERTE

- Java weist bestimmten Variablen initiale Werte zu!

STANDARDWERTE

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

STANDARDWERTE

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

STANDARDWERTE

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

STANDARDWERTE

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

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`**.

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“.

STANDARDWERTE

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

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

STANDARDWERTE

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

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

- `k = '\u0000'`.

STANDARDWERTE

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

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

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

STANDARDWERTE

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

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

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

STANDARDWERTE

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

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

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

KOMPLEXE DATENTYPEN

KOMPLEXE DATENTYPEN

- Arrays sind komplexe Datentypen.

KOMPLEXE DATENTYPEN

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

KOMPLEXE DATENTYPEN

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

KOMPLEXE DATENTYPEN

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

KOMPLEXE DATENTYPEN

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

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

KOMPLEXE DATENTYPEN

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

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

  speichert Java für `rnd` die Referenz,

KOMPLEXE DATENTYPEN

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

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

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

KOMPLEXE DATENTYPEN

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

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

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

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

KOMPLEXE DATENTYPEN

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

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

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

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

STRINGS

STRINGS

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

STRINGS

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

STRINGS

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

STRINGS

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

  `s1`.equals(s2): Prüft, ob die Zeichenketten *gleich* sind.

  `s1`.length(): Liefert die Länge einer Zeichenkette.

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.

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.

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

FUNKTIONEN

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

FUNKTIONEN

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

FUNKTIONEN

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

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

FUNKTIONEN

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

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

- Beispiel:

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 /* ... */ ;
}
```

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:

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,

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`**,

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

SIGNATUR & ÜBERLADUNG

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

SIGNATUR & ÜBERLADUNG

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

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.

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.

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.

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

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)`, …),

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.

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

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:

PRIVATE UND PUBLIC

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

PRIVATE UND PUBLIC

- Java erlaubt vier verschiedene Zugriffsmodifikatoren:

  PRIVATE: Zugriff ist nur innerhalb der Klasse erlaubt.

  PROTECTED: Ist sichtbar im gesamten Paket sowie in allen Unterklassen (Vererbung).

PRIVATE UND PUBLIC

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

PRIVATE UND PUBLIC

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

PRIVATE UND PUBLIC

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

PRIVATE UND PUBLIC

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

GÜLTIGKEITSBEREICHE – SCOPES

GÜLTIGKEITSBEREICHE – SCOPES

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

GÜLTIGKEITSBEREICHE – SCOPES

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

```
public class Pinguin { // 1.




}
```

GÜLTIGKEITSBEREICHE – SCOPES

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

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




}
```

GÜLTIGKEITSBEREICHE – SCOPES

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

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


}
```

GÜLTIGKEITSBEREICHE – SCOPES

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

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

}
```

GÜLTIGKEITSBEREICHE – SCOPES

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

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

GÜLTIGKEITSBEREICHE – SCOPES

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

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

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

GÜLTIGKEITSBEREICHE – SCOPES

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

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

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

	1	2	3	4	5
`Pinguin` `Tiger`					

	1	2	3	4	5
`Auto` `Felspingu`					

GÜLTIGKEITSBEREICHE – SCOPES

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

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

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

	1	2	3	4	5
`Pinguin` `Tiger`	☑	☑	☑	☑	☑

	1	2	3	4	5
`Auto` `Felspingu`					

GÜLTIGKEITSBEREICHE – SCOPES

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

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

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

	1	2	3	4	5
`Pinguin`	☑	☑	☑	☑	☑
`Tiger`	☑	☐	☑	☑	☑

	1	2	3	4	5
`Auto`					
`Felspingu`					

GÜLTIGKEITSBEREICHE – SCOPES

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

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

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

	1	2	3	4	5
`Pinguin`	☑	☑	☑	☑	☑
`Tiger`	☑	☐	☑	☑	☑

	1	2	3	4	5
`Auto`	☑	☐	☐	☑	☐
`Felspingu`					

GÜLTIGKEITSBEREICHE – SCOPES

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

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

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

	1	2	3	4	5
`Pinguin`	☑	☑	☑	☑	☑
`Tiger`	☑	☐	☑	☑	☑

	1	2	3	4	5
`Auto`	☑	☐	☐	☑	☐
`Felspingu`	☑	☐	☑	☑	☐

ES GILT FESTZUHALTEN…

Definition 2: Subroutinen

Eine Subroutine / Funktion ist ein *Unterprogramm*,

ES GILT FESTZUHALTEN…

Definition 2: Subroutinen

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

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,

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.

Kontrollstrukturen und Arrays

FALLUNTERSCHEIDUNG MIT IF

FALLUNTERSCHEIDUNG MIT IF

- Java erlaubt Fallunterscheidungen mit **`if`**:

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
}
```

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,

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

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

  ⟨`Bedingung`⟩ `?` ⟨`Bedingung-wahr`⟩ `:` ⟨`Bedingung-falsch`⟩

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

  ⟨`Bedingung`⟩ `?` ⟨`Bedingung-wahr`⟩ `:` ⟨`Bedingung-falsch`⟩

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

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:

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

  ⟨`Bedingung`⟩ `?` ⟨`Bedingung-wahr`⟩ `:` ⟨`Bedingung-falsch`⟩

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

- Ein Beispiel:

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

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

  ⟨`Bedingung`⟩ `?` ⟨`Bedingung-wahr`⟩ `:` ⟨`Bedingung-falsch`⟩

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

- Ein Beispiel:

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

  › Liefert: `"Nicht volljährig"`.

KOMPAKTE IF-NOTATION

- Eine (einfache) Fallunterscheidung können wir verkürzen:

  ⟨`Bedingung`⟩ `?` ⟨`Bedingung-wahr`⟩ `:` ⟨`Bedingung-falsch`⟩

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

- Ein Beispiel:

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

  › Liefert: `"Nicht volljährig"`.

- Diese Anweisung kann (beliebig tief) verschachtelt werden.

FALLUNTERSCHEIDUNG MIT SWITCH-CASE

FALLUNTERSCHEIDUNG MIT SWITCH-CASE

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

FALLUNTERSCHEIDUNG MIT SWITCH-CASE

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

```
switch(⟨Ausdruck⟩){




}
```

FALLUNTERSCHEIDUNG MIT SWITCH-CASE

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

```
switch(⟨Ausdruck⟩){
    case ⟨Fall 1⟩: // Code für Fall 1









}
```

FALLUNTERSCHEIDUNG MIT SWITCH-CASE

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

```
switch(⟨Ausdruck⟩){
    case ⟨Fall 1⟩: // Code für Fall 1
    case ⟨Fall 2⟩:









}
```

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




}
```

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;

}
```

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.
}
```

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.
}
```

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:

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),

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

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

WHILE

- Schleifen erlauben es in Java bestimmte Programmanweisungen mehrfach auszuführen:

WHILE

- Schleifen erlauben es in Java bestimmte Programmanweisungen mehrfach auszuführen:

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

WHILE

- Schleifen erlauben es in Java bestimmte Programmanweisungen mehrfach auszuführen:

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

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

WHILE UND DO-WHILE

WHILE UND DO-WHILE

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

WHILE UND DO-WHILE

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

WHILE UND DO-WHILE

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

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

WHILE UND DO-WHILE

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

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

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

ZÄHLSCHLEIFEN MIT FOR

ZÄHLSCHLEIFEN MIT FOR

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

ZÄHLSCHLEIFEN MIT FOR

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

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

ZÄHLSCHLEIFEN MIT FOR

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

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

- Betrachten wir ein Beispiel:

ZÄHLSCHLEIFEN MIT FOR

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

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

- Betrachten wir ein Beispiel:

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

ZÄHLSCHLEIFEN MIT FOR

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

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

- Betrachten wir ein Beispiel:

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

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

ZÄHLSCHLEIFEN MIT FOR

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

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

- Betrachten wir ein Beispiel:

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

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

ZÄHLSCHLEIFEN MIT FOR

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

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

- Betrachten wir ein Beispiel:

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

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

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

BREAK UND CONTINUE

BREAK UND CONTINUE

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

BREAK UND CONTINUE

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

BREAK UND CONTINUE

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

BREAK UND CONTINUE

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

BREAK UND CONTINUE

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

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

BREAK UND CONTINUE

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

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

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

ITERIEREN MIT FOREACH

ITERIEREN MIT FOREACH

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

ITERIEREN MIT FOREACH

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

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

ITERIEREN MIT FOREACH

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

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

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

ITERIEREN MIT FOREACH

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

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

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

ITERIEREN MIT FOREACH

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

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

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

- Ein Beispiel:

ITERIEREN MIT FOREACH

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

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

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

- Ein Beispiel:

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

ARRAYS, GRUNDLAGEN

ARRAYS, GRUNDLAGEN

- Arrays sind eine komplexe Datenstruktur,

ARRAYS, GRUNDLAGEN

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

ARRAYS, GRUNDLAGEN

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

ARRAYS, GRUNDLAGEN

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

ARRAYS, GRUNDLAGEN

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

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

ARRAYS, GRUNDLAGEN

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

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

- Die Länge erhalten wir durch ⟨`array`⟩`.length`.

ARRAYS, GRUNDLAGEN

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

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

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

ARRAYS, TECHNISCHER HINTERGRUND

ARRAYS, TECHNISCHER HINTERGRUND

- Arrays werden in Java als ein Block gespeichert.

ARRAYS, TECHNISCHER HINTERGRUND

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

ARRAYS, TECHNISCHER HINTERGRUND

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

ARRAYS, TECHNISCHER HINTERGRUND

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

ARRAYS, TECHNISCHER HINTERGRUND

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

ARRAYS, TECHNISCHER HINTERGRUND

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

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

ARRAYS, TECHNISCHER HINTERGRUND

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

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

  Hier verweisen beide Variablen auf denselben Datensatz:

ARRAYS, TECHNISCHER HINTERGRUND

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

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

  Hier verweisen beide Variablen auf denselben Datensatz:

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

ÜBER ARRAYS ITERIEREN

ÜBER ARRAYS ITERIEREN

- Wir können,

ÜBER ARRAYS ITERIEREN

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

ÜBER ARRAYS ITERIEREN

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

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

ÜBER ARRAYS ITERIEREN

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

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

- Oder mit „for-each“:

ÜBER ARRAYS ITERIEREN

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

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

- Oder mit „for-each“:

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

MEHRDIMENSIONALE ARRAYS

MEHRDIMENSIONALE ARRAYS

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

MEHRDIMENSIONALE ARRAYS

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

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

MEHRDIMENSIONALE ARRAYS

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

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

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

MEHRDIMENSIONALE ARRAYS

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

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

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

```
int x = b[2][3]; // x → 5
```

VERSCHACHTELTE FOR-SCHLEIFEN

VERSCHACHTELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

VERSCHACHTELTE FOR-SCHLEIFEN

- Über ein solches Array können wir auch iterieren:

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




}
```

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++) {




    }
}
```

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++) {

        }

    }
}
```

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] + "␣");
        }

    }
}
```

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();
    }
}
```

OO-Konzepte

PARADIGMA: OBJEKTORIENTIERUNG

PARADIGMA: OBJEKTORIENTIERUNG

- Klassen erlauben es, Objekte der realen Welt abzubilden.

PARADIGMA: OBJEKTORIENTIERUNG

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

PARADIGMA: OBJEKTORIENTIERUNG

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

PARADIGMA: OBJEKTORIENTIERUNG

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

  DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.

PARADIGMA: OBJEKTORIENTIERUNG

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

  DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.

  FUNKTIONEN: diese definieren, was eine `Person` kann.

PARADIGMA: OBJEKTORIENTIERUNG

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

  DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.

  FUNKTIONEN: diese definieren, was eine `Person` kann. Also `gehen()`, `tanzen()`, `reden()`, ...

PARADIGMA: OBJEKTORIENTIERUNG

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

  DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.

  FUNKTIONEN: diese definieren, was eine `Person` kann. Also `gehen()`, `tanzen()`, `reden()`, …. Dies sind die **Methoden** der Klasse, sie definieren das *Verhalten*.

PARADIGMA: OBJEKTORIENTIERUNG

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

  DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.

  FUNKTIONEN: diese definieren, was eine `Person` kann. Also `gehen()`, `tanzen()`, `reden()`, …. Dies sind die **Methoden** der Klasse, sie definieren das *Verhalten*.

- Eine Funktion nennen wir *Methode*, wenn sie an eine Objekt gebunden ist. (Zumindest in Java)

PARADIGMA: OBJEKTORIENTIERUNG

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

  DATEN: diese, wie der Name, das Alter oder die Größe der Person, werden als Variablen an die Klasse gebunden. Sie sind die **Attribute** der Klasse und definieren den *Zustand*.

  FUNKTIONEN: diese definieren, was eine `Person` kann. Also `gehen()`, `tanzen()`, `reden()`, …. Dies sind die **Methoden** der Klasse, sie definieren das *Verhalten*.
- Eine Funktion nennen wir *Methode*, wenn sie an eine Objekt gebunden ist. (Zumindest in Java)
- Klassen erlauben es, komplexe Probleme zu abstrahieren und Funktionalität zu kapseln.

PARADIGMA: OBJEKTORIENTIERUNG

Definition 3: Objekt

Definition 3: Objekt

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

Definition 3: Objekt

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

OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.

Definition 3: Objekt

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

OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.

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

Definition 3: Objekt

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

- OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.
- OBJEKTVERHALTEN: bezeichnet die Reaktion des Objekts auf das Aufrufen von Methoden.
- OBJEKTIDENTITÄT: ermöglicht die eindeutige Identifikation.
  In Java: die Speicheradresse.

Definition 3: Objekt

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

- OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.
- OBJEKTVERHALTEN: bezeichnet die Reaktion des Objekts auf das Aufrufen von Methoden.
- OBJEKTIDENTITÄT: ermöglicht die eindeutige Identifikation.
  In Java: die Speicheradresse.

- Legt man die Objektorientierung streng aus,

Definition 3: Objekt

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

- OBJEKTZUSTAND: wird durch die ihm zugehörigen Attribute definiert.
- OBJEKTVERHALTEN: bezeichnet die Reaktion des Objekts auf das Aufrufen von Methoden.
- OBJEKTIDENTITÄT: ermöglicht die eindeutige Identifikation.
  In Java: die Speicheradresse.

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

PROGRAMMKONTEXT

PROGRAMMKONTEXT

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

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.

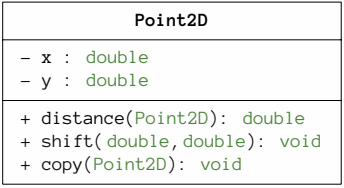
PROGRAMMKONTEXT

- 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*

PROGRAMMKONTEXT

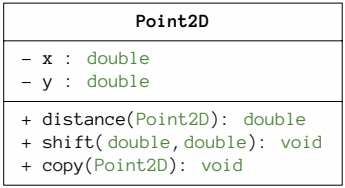
- Das Verhalten von Klassen wird oft auch als ein Botschaft- beziehungsweise Nachrichtensystem betrachtet.
- Hier bezeichnet das Senden einer Botschaft mit Inhalt das Aufrufen der Methode mit entsprechendem Inhalt.
- Aus Sicht der Objektorientierung ist ein *Programm* nicht mehr als das (wechselseitige)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

VOM KLASSENKONZEPT ZUM JAVACODE



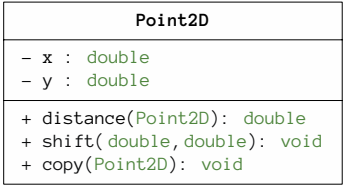
VOM KLASSENKONZEPT ZUM JAVACODE

- Es gelte eine Klasse `Punkt2D` zu kreieren.



VOM KLASSENKONZEPT ZUM JAVACODE

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



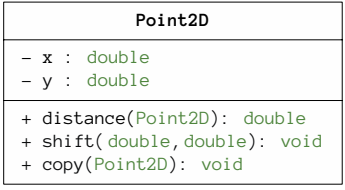
VOM KLASSENKONZEPT ZUM JAVACODE

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

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

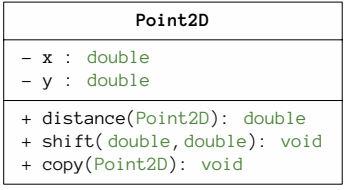
VOM KLASSENKONZEPT ZUM JAVACODE

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



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:

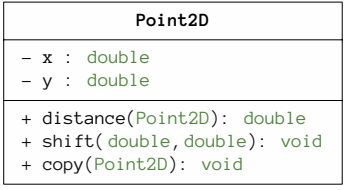


VOM KLASSENKONZEPT ZUM JAVACODE

- 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.

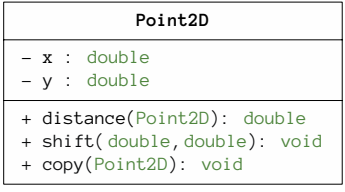


VOM KLASSENKONZEPT ZUM JAVACODE

- 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.

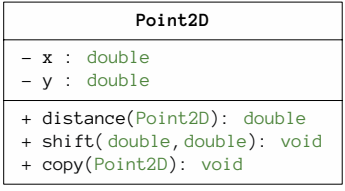


VOM KLASSENKONZEPT ZUM JAVACODE

- 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

IMPLEMENTATION IN JAVA

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

IMPLEMENTATION IN JAVA

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

IMPLEMENTATION IN JAVA

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

IMPLEMENTATION IN JAVA

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

IMPLEMENTATION IN JAVA

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

IMPLEMENTATION IN JAVA

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

IMPLEMENTATION IN JAVA

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

Definition 4: Konstruktor

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

IMPLEMENTATION IN JAVA

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

Definition 4: Konstruktor

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

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

IMPLEMENTATION: KONSTRUKTOR

IMPLEMENTATION: KONSTRUKTOR

```
public class Point2D {









}
```

IMPLEMENTATION: KONSTRUKTOR

```
public class Point2D {

    private double x, y;









}
```

IMPLEMENTATION: KONSTRUKTOR

```java
public class Point2D {

    private double x, y;

    // Leerer Konstruktor
    public Point2D() { this(0.0, 0.0); }









}
```

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

IMPLEMENTATION: METHODEN

- Die Implementation der Methoden läuft wie bekannt!

IMPLEMENTATION: METHODEN

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

IMPLEMENTATION: METHODEN

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

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




}
```

IMPLEMENTATION: METHODEN

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

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

    public void shift(double sx, double sy) {


    }
}
```

IMPLEMENTATION: METHODEN

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

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

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

BESONDERE METHODEN

BESONDERE METHODEN

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

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden,

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:

  equals: die Methode `equals(Object)` prüft das Objekt mit dem Übergebenen auf „Gleichheit“.

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:

  equals: die Methode `equals(Object)` prüft das Objekt mit dem Übergebenen auf „Gleichheit“. So können die hierfür relevanten Attribute genau festgelegt werden.

BESONDERE METHODEN

- Jede Java-Klasse übernimmt (erbt) Methoden der `Object`-Klasse.
- Auf diese Weise gibt es einige wichtige Methoden, die im Kontext von Java eine besondere Bedeutung haben (aber dafür natürlich, wie `equals` überschrieben werden müssen).
- Hier sind die wichtigsten:

  equals: die Methode `equals(Object)` prüft das Objekt mit dem Übergebenen auf „Gleichheit“. So können die hierfür relevanten Attribute genau festgelegt werden.

  toString: wird aufgerufen, um eine Repräsentation als Zeichenkette zu erhalten.

STATISCHE METHODEN UND ATTRIBUTE

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double)`, …).

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double)`, …).
- Diese Methoden deklarieren wir mit **`static`**,

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double)`, …).
- Diese Methoden deklarieren wir mit **`static`**, sie sind nun auch ohne ein Objekt aufrufbar.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double)`, …).
- Diese Methoden deklarieren wir mit **`static`**, sie sind nun auch ohne ein Objekt aufrufbar. So kann man zum Beispiel auch ein statisches `Point2D::distance(Point2D, Point2D)` für zwei Punkte bauen.

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double)`, …).
- Diese Methoden deklarieren wir mit **`static`**, sie sind nun auch ohne ein Objekt aufrufbar. So kann man zum Beispiel auch ein statisches `Point2D::distance(Point2D, Point2D)` für zwei Punkte bauen.
- Auch Variablen, die für alle Objekte einer Klasse identisch sind,

STATISCHE METHODEN UND ATTRIBUTE

- Eine ausführlichere Erklärung hier: 🖇 static.pdf.
- Nichtstatische Methoden, sind an ein Objekt gebunden.
- Manche Methoden sind aber semantisch nur an eine Klasse gebunden und nicht an die Instanzen (`Math.floor(double)`, …).
- Diese Methoden deklarieren wir mit **`static`**, sie sind nun auch ohne ein Objekt aufrufbar. So kann man zum Beispiel auch ein statisches `Point2D::distance(Point2D, Point2D)` für zwei Punkte bauen.
- Auch Variablen, die für alle Objekte einer Klasse identisch sind, können wir mit **`static`** deklarieren.

DER LEBENSZYKLUS EINES OBJEKTS

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe…)

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe…) können wir weitere Referenzen darauf erstellen.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe…) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe…) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.
- Existiert für ein Objekt keine Referenz mehr,

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe…) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.
- Existiert für ein Objekt keine Referenz mehr, gibt es (für Java) keine Möglichkeit mehr darauf zuzugreifen.

DER LEBENSZYKLUS EINES OBJEKTS

- Mit dem Erstellen eines Objekts belegen wir Speicher auf dem Heap.
- Durch weitere Zuweisungen oder (Methodenaufrufe…) können wir weitere Referenzen darauf erstellen.
- Überschreiben wir diese Variablen (oder verlassen ihren Scope) verlieren wir eine Referenz.
- Existiert für ein Objekt keine Referenz mehr, gibt es (für Java) keine Möglichkeit mehr darauf zuzugreifen.
- In diesem Fall wird es (irgendwann) vom Garbage-Collector aufgeräumt.

ENUMERATIONEN

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden,

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann.

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*
- Die einfachste Möglichkeit eine Enumeration zu definieren,

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*
- Die einfachste Möglichkeit eine Enumeration zu definieren, funktioniert über das `enum`-Schlüsselwort:

ENUMERATIONEN

- Mit Java-5 gibt es die Möglichkeit durch **Enumerationen** eigene Datentypen zu definieren.
- Sie können überall dort definiert werden, wo man auch eine Klasse definieren kann. *Technisch betrachtet sind Enumerationen spezielle Java-Klassen.*
- Die einfachste Möglichkeit eine Enumeration zu definieren, funktioniert über das **`enum`**-Schlüsselwort:

```
enum ⟨Name der Enumeration⟩ {
    ⟨Komma separierte Liste an Werten⟩
}
```

ENUMERATIONEN DEFINIEREN

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {
    HOCH, RUNTER, LINKS, RECHTS
}
```

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {
    HOCH, RUNTER, LINKS, RECHTS
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {
    HOCH, RUNTER, LINKS, RECHTS
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.
- *Hinweis:* Anders als in Sprachen wie C++,

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {
    HOCH, RUNTER, LINKS, RECHTS
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.
- *Hinweis:* Anders als in Sprachen wie C++, wird den Konstanten kein (Integer)-Wert zugeordnet.

ENUMERATIONEN DEFINIEREN

- Der Konvention nach, werden alle Werte einer Enumeration in Großbuchstaben und mit Unterstrichen geschrieben.
- Betrachten wir ein Beispiel:

```
enum Richtung {
    HOCH, RUNTER, LINKS, RECHTS
}
```

- Wir können Enumerationen als Datentypen verwenden und über die Punkt-Syntax auf Elemente zugreifen.
- *Hinweis:* Anders als in Sprachen wie C++, wird den Konstanten kein (Integer)-Wert zugeordnet. (Dafür gibt es `ordinal()`.)

ENUMERATIONEN VERWENDEN

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){




}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {




    }

}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es␣geht␣nach␣oben!";




    }

}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es geht nach oben!";
        case RUNTER: return "Es geht nach unten!";


    }

}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es geht nach oben!";
        case RUNTER: return "Es geht nach unten!";
        case LINKS: return "Es geht nach links!";

    }

}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es geht nach oben!";
        case RUNTER: return "Es geht nach unten!";
        case LINKS: return "Es geht nach links!";
        case RECHTS: return "Es geht nach rechts!";
    }

}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es geht nach oben!";
        case RUNTER: return "Es geht nach unten!";
        case LINKS: return "Es geht nach links!";
        case RECHTS: return "Es geht nach rechts!";
    }
    return "Fehler! Richtung unbekannt";
}
```

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es␣geht␣nach␣oben!";
        case RUNTER: return "Es␣geht␣nach␣unten!";
        case LINKS: return "Es␣geht␣nach␣links!";
        case RECHTS: return "Es␣geht␣nach␣rechts!";
    }
    return "Fehler!␣Richtung␣unbekannt";
}
```

- Beispielhafte Verwendung:

ENUMERATIONEN VERWENDEN

- Betrachten wir ein Beispiel mit `Richtung` und **`switch-case`**:

```
public static String wohinGehtEs(Richtung ziel){
    switch(ziel) {
        case HOCH: return "Es geht nach oben!";
        case RUNTER: return "Es geht nach unten!";
        case LINKS: return "Es geht nach links!";
        case RECHTS: return "Es geht nach rechts!";
    }
    return "Fehler! Richtung unbekannt";
}
```

- Beispielhafte Verwendung:

```
System.out.println(wohinGehtEs(Richtung.RECHTS));
```

WAS ENUMERATIONEN LIEFERN

WAS ENUMERATIONEN LIEFERN

- Mittels ⟨`Enum Name`⟩`.values()` erhalten wir ein Array aller Werte.

WAS ENUMERATIONEN LIEFERN

- Mittels ⟨`Enum Name`⟩`.values()` erhalten wir ein Array aller Werte.
- ⟨`Enum Name`⟩`.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen,

WAS ENUMERATIONEN LIEFERN

- Mittels ⟨`Enum Name`⟩`.values()` erhalten wir ein Array aller Werte.
- ⟨`Enum Name`⟩`.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.

WAS ENUMERATIONEN LIEFERN

- Mittels ⟨`Enum Name`⟩`.values()` erhalten wir ein Array aller Werte.
- ⟨`Enum Name`⟩`.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.
- Enumerationen besitzen ein wie zu erwarten funktionierendes `equals(Object)`.

WAS ENUMERATIONEN LIEFERN

- Mittels ⟨`Enum Name`⟩`.values()` erhalten wir ein Array aller Werte.
- ⟨`Enum Name`⟩`.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.
- Enumerationen besitzen ein wie zu erwarten funktionierendes `equals(Object)`.
- Analog funktioniert `toString()` wie zu erwarten.

WAS ENUMERATIONEN LIEFERN

- Mittels ⟨`Enum Name`⟩`.values()` erhalten wir ein Array aller Werte.
- ⟨`Enum Name`⟩`.valueOf(String)` liefert die Enumkonstante mit übergebenem Namen, sofern vorhanden.
- Enumerationen besitzen ein wie zu erwarten funktionierendes `equals(Object)`.
- Analog funktioniert `toString()` wie zu erwarten.
- Da es sich auch um Klassen handelt, können wir den Konstanten auch Datentypen zuordnen!

ALTERNATIVE RICHTUNGS-ENUM

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {









}
```

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {

    private String text;

}
```

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {




    private String text;
    public String getText() { return this.text; }




}
```

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {




    private String text;
    public String getText() { return this.text; }

    Richtung(String text) {
        this.text = text;
    }
}
```

ALTERNATIVE RICHTUNGS-ENUM

```
enum Richtung {
    HOCH("Es␣geht␣nach␣oben!"),
    RUNTER("Es␣geht␣nach␣unten!"),
    LINKS("Es␣geht␣nach␣links!"),
    RECHTS("Es␣geht␣nach␣rechts!");

    private String text;
    public String getText() { return this.text; }

    Richtung(String text) {
        this.text = text;
    }
}
```

ABSCHLUSS ZU ENUMERATIONEN

ABSCHLUSS ZU ENUMERATIONEN

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

ABSCHLUSS ZU ENUMERATIONEN

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

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

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen),

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen), wird *keine* Kopie des Objekts erstellt,

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen), wird *keine* Kopie des Objekts erstellt, sondern eine Kopie der Referenz auf das Objekt übergeben.

KLASSEN ALS PARAMETER: CALL-BY-REFERENCE

- Wenn wir komplexe Datentypen als Parameter übergeben (oder zuweisen), wird *keine* Kopie des Objekts erstellt, sondern eine Kopie der Referenz auf das Objekt übergeben.
- Dieser (Parameter-)Übergabemechanismus ähnelt *call-by-reference* aus anderen Programmiersprachen

DER NETTE BRUDER: CALL-BY-VALUE

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst,

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*,

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*, man übergibt also den eigentlichen Wert der Variable („was auf dem Stack liegt“).

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*, man übergibt also den eigentlichen Wert der Variable („was auf dem Stack liegt“).
- Wenn man komplexe Datentypen kopieren möchte,

DER NETTE BRUDER: CALL-BY-VALUE

- Bei komplexen Datentypen wird also nicht das Objekt selbst, sondern nur eine Referenz übergeben.
- Bei primitiven Datentypen wird der Wert *kopiert*, man übergibt also den eigentlichen Wert der Variable („was auf dem Stack liegt“).
- Wenn man komplexe Datentypen kopieren möchte, erstellt man sie in der Regel neu und greift dabei auf die *call-by-value*-Charakteristik der primitiven Datentypen zurück.

Programmiertheorie

○ ○ ○ ○ 5 ○ ○ ○

Florian Sihler, 2.01.2022

REKURSIVE UND ITERATIVE PROGRAMMIERUNG

Definition 5: Rekursive Methode

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
- *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
- *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.

- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
- *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.

- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:
  ABBRUCHBEDINGUNG: Das lösbare Teilproblem. Ende der Rekursion. Hier wird die Methode nicht weiter referenziert.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
- *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.

- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:
  - Abbruchbedingung: Das lösbare Teilproblem. Ende der Rekursion. Hier wird die Methode nicht weiter referenziert.
  - Rekursiver Zweig: Enthält die rekursiv auszuführende Prozedur.

Definition 5: Rekursive Methode

Eine Methode oder Funktion bezeichnen wir als *rekursiv*, wenn sie sich:

- *direkt* selbst aufruft, sich also selbst referenziert.
- *indirekt* selbst aufruft, also eine andere Methode verwendet, die (über irgendwelche Umwege) wieder die Methode aufruft.

- Eine rekursive Methode lässt sich in zwei Komponenten gliedern:
  - Abbruchbedingung: Das lösbare Teilproblem. Ende der Rekursion. Hier wird die Methode nicht weiter referenziert.
  - Rekursiver Zweig: Enthält die rekursiv auszuführende Prozedur.

- Rekursion und Iteration sind *gleichmächtig*. Iteration ist in der Regel „performanter“, Rekursion ist oft „eleganter“ (rekursive Datenstrukturen, …).

ARTEN VON REKURSION: HEAD UND TAIL

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion.

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){




}
```

```
public void tailDecrement(int i){




}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;



}
```

```
public void tailDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;



}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Rekursion
    else headDecrement(i - 1);


}
```

```
public void tailDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;




}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```
public void headDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Rekursion
    else headDecrement(i - 1);


}
```

```
public void tailDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Verarbeitung
    else System.out.println(i);


}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```java
public void headDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Rekursion
    else headDecrement(i - 1);
    // Verarbeitung
    System.out.println(i);
}
```

```java
public void tailDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Verarbeitung
    else System.out.println(i);
    // Rekursion
    tailDecrement(i - 1);
}
```

ARTEN VON REKURSION: HEAD UND TAIL

- Wir unterscheiden (unter anderem) zwei besondere Rekursionen:
  - HEAD: Der rekursive Aufruf erfolgt zu Beginn: „Alles passiert im Aufstieg.“
  - TAIL: Der rekursive Aufruf erfolgt zu Ende der Rekursion. Ein solcher Abstieg kann einfach in eine Iteration transformiert werden (z.B. durch den Compiler): „Alles passiert im Abstieg.“

```java
// → 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);
}
```

```java
// → 10 9 8 7 6 ...
public void tailDecrement(int i){
    // Abbruchbedingung
    if(i == 0) return;
    // Verarbeitung
    else System.out.println(i);
    // Rekursion
    tailDecrement(i - 1);
}
```

ARTEN VON REKURSION: BAUMARTIGE REKURSION

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg,

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.
- Beliebte Beispiele: die Fibonaccifolge und der Binomialkoeffizient.

ARTEN VON REKURSION: BAUMARTIGE REKURSION

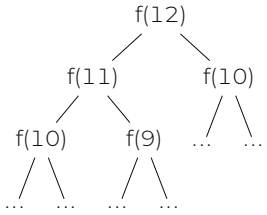
- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.
- Beliebte Beispiele: die Fibonaccifolge und der Binomialkoeffizient.

```
int fib(int n) {
    if(n <= 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```

ARTEN VON REKURSION: BAUMARTIGE REKURSION

- Eine Methode kann sich auch mehrfach (auch indirekt) selbst referenzieren.
- In diesem Fall entsteht kein „linearer“ Abstieg, sondern vielmehr eine baum-/kaskadenartige Verzweigung.
- Beliebte Beispiele: die Fibonaccifolge und der Binomialkoeffizient.

```
int fib(int n) {
    if(n <= 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
```



VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

  GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf.

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

  GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf. Beispielsweise die Ackermannfunktion (nach Péter):

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

  GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf. Beispielsweise die Ackermannfunktion (nach Péter):

```
int a(int n, int m) {
    if(n == 0) return m + 1;
    else if (m == 0) return a(n - 1, 1);
    else return a(n - 1, a(n, m - 1));
}
```

VERSCHRÄNKTE UND GESCHACHTELTE REKURSION

- Es gibt weitere Arten der Rekursion:

GESCHACHTELT: Hier ist (mindestens) ein Parameter im rekursiven Aufruf selbst ein rekursiver Aufruf. Beispielsweise die Ackermannfunktion (nach Péter):

```
int a(int n, int m) {
    if(n == 0) return m + 1;
    else if (m == 0) return a(n - 1, 1);
    else return a(n - 1, a(n, m - 1));
}
```

VERSCHRÄNKT: Hier rufen sich mehrere Funktionen rekursiv gegenseitig auf. Diese Variante lässt sich schwer bis gar nicht (direkt) in eine Schleife übersetzten.

DAS PARADIGMA: DIVIDE AND CONQUER

DAS PARADIGMA: DIVIDE AND CONQUER

- Manche komplizierte Probleme,

DAS PARADIGMA: DIVIDE AND CONQUER

- Manche komplizierte Probleme, lassen sich durch Rekursion in immer kleinere Probleme aufspalten, die dann beherrschbarer sind.

DAS PARADIGMA: DIVIDE AND CONQUER

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

DAS PARADIGMA: BACKTRACKING

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.
- Das Problem wird von einer Teillösung aus bis zur Gesamtlösung erweitert.

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.
- Das Problem wird von einer Teillösung aus bis zur Gesamtlösung erweitert. Eine „Sackgasse“ veranlasst einen neuen Versuch (trial and error).

DAS PARADIGMA: BACKTRACKING

- Backtracking ist eine rekursive Lösungsstrategie.
- Das Problem wird von einer Teillösung aus bis zur Gesamtlösung erweitert. Eine „Sackgasse“ veranlasst einen neuen Versuch (trial and error).
- Im Sackgassen-Fall macht man die Entscheidungen solange rückgängig, bis man eine andere Erweiterung wählen kann und erweitert die Teillösung dann durch diese.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:*

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:* Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:* Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:* Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku.*

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:* Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku.* Füge in erstes freies Feld eine der dort möglichen Zahlen ein.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:* Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku.* Füge in erstes freies Feld eine der dort möglichen Zahlen ein. Verfahre so, bis alle Felder gefüllt sind oder für ein Feld keine Zahl mehr möglich ist.

DAS PARADIGMA: BACKTRACKING, BEISPIELE

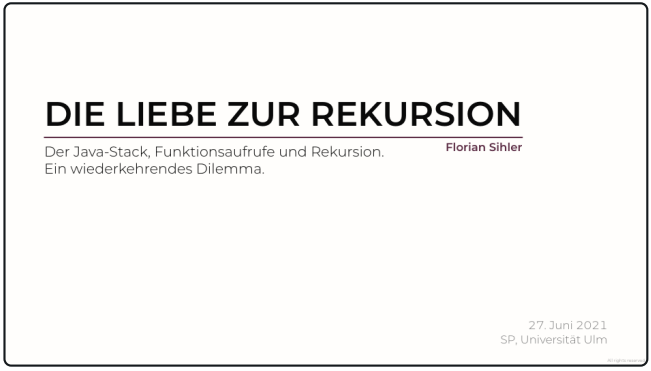
- *Wegfindung im Labyrinth:* Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.

- *Lösung eines Sudoku.* Füge in erstes freies Feld eine der dort möglichen Zahlen ein. Verfahre so, bis alle Felder gefüllt sind oder für ein Feld keine Zahl mehr möglich ist. In diesem Fall: springe zum letzten Punkt zurück, an dem noch andere Zahlen möglich sind. Probiere weiter.

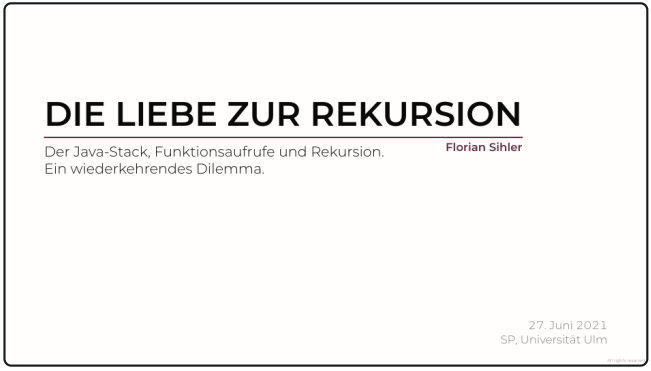
DAS PARADIGMA: BACKTRACKING, BEISPIELE

- *Wegfindung im Labyrinth:* Gehe vom Startfeld aus solange einen Weg entlang, bis am Ziel angekommen. Im Fall einer Sackgasse: springe zur letzten Position zurück, an der es noch einen anderen Weg gab.
- *Lösung eines Sudoku.* Füge in erstes freies Feld eine der dort möglichen Zahlen ein. Verfahre so, bis alle Felder gefüllt sind oder für ein Feld keine Zahl mehr möglich ist. In diesem Fall: springe zum letzten Punkt zurück, an dem noch andere Zahlen möglich sind. Probiere weiter.
- Rucksackproblem, 8-Damen Problem, …

ZUR VERTIEFUNG


DIE LIEBE ZUR REKURSION
Der Java-Stack, Funktionsaufrufe und Rekursion.
Ein wiederkehrendes Dilemma.
Florian Sihler
27. Juni 2021
SP, Universität Ulm

ZUR VERTIEFUNG



Mehr zum Thema „Rekursion“ per Klick…

KOMPLEXITÄTSBETRACHTUNG

KOMPLEXITÄTSBETRACHTUNG

- Da die Ausführungszeit eines Programms von vielen Parametern

KOMPLEXITÄTSBETRACHTUNG

- Da die Ausführungszeit eines Programms von vielen Parametern (Taktrate des Prozessors, andere laufende Programme, …) abhängig ist,

KOMPLEXITÄTSBETRACHTUNG

- Da die Ausführungszeit eines Programms von vielen Parametern (Taktrate des Prozessors, andere laufende Programme, …) abhängig ist, betrachten wir oft nur dessen Skalierung.

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

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.

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

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:

  WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:

  WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.

  BEST-CASE: Die Laufzeitkomplexität im günstigsten Fall für den (spezifischen) Algorithmus.

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:

  WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.

  BEST-CASE: Die Laufzeitkomplexität im günstigsten Fall für den (spezifischen) Algorithmus.

  AVERAGE-CASE: Die Laufzeitkomplexität im durchschnittlichen Fall für den (spezifischen) Algorithmus.

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.

KOMPLEXITÄTSBETRACHTUNG

- Für die Laufzeitkomplexität unterscheidet man:

  WORST-CASE: Die Laufzeitkomplexität im schlechtesten Fall für den (spezifischen) Algorithmus.

  BEST-CASE: Die Laufzeitkomplexität im günstigsten Fall für den (spezifischen) Algorithmus.

  AVERAGE-CASE: Die Laufzeitkomplexität im durchschnittlichen Fall für den (spezifischen) Algorithmus. Dies bezeichnet in der Regel gleichverteilt zufällige Eingaben.

- Dabei werden wir den *average-case* vernachlässigen.

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

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;
}
```

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

- `Zuweisungen`:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

- `Zuweisungen`: $2 + n$

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

- `Zuweisungen`: $2 + n$
- `Vergleiche`:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

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

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

- `Zuweisungen`: $2 + n$
- `Vergleiche`: $n + 1 + \frac{n(n+1)}{2}$
- `Inkrementierungen`:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

- `Zuweisungen`: $2 + n$
- `Vergleiche`: $n + 1 + \frac{n(n+1)}{2}$
- `Inkrementierungen`: $n + \frac{n(n-1)}{2}$

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

- `Zuweisungen`: $2 + n$
- `Vergleiche`: $n + 1 + \frac{n(n+1)}{2}$
- `Inkrementierungen`: $n + \frac{n(n-1)}{2}$
- `Dekrementierungen`:

ERFASSEN DER KOMPLEXITÄT – PRÄZISE

- Die Erfassung der Laufzeitkomplexität erfolgt durch die Auflistung der notwendigen (Rechen-)Schritte:

```
static int methode(int n) {
    int count = 2;
    for(int i = 1; i <= n; i++) {
        for(int j = n; j > i; j--)
            count++;
    }
    return count;
}
```

- `Zuweisungen`: $2 + n$
- `Vergleiche`: $n + 1 + \frac{n(n+1)}{2}$
- `Inkrementierungen`: $n + \frac{n(n-1)}{2}$
- `Dekrementierungen`: $\frac{n(n-1)}{2}$

ERFASSEN DER KOMPLEXITÄT

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

Definition 7: $\mathcal{O}$-Notation

ERFASSEN DER KOMPLEXITÄT

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

Definition 7: $\mathcal{O}$-Notation

Es gilt $T(n) \in \mathcal{O}(f(n))$, wenn $f(n)$ eine obere Schranke von $T(n)$ ist, also:

- Insgesamt ergibt sich damit ein Aufwand von $\frac{3n^2}{2} + \frac{5n}{2} + 3$.
- Da für große Datenmengen Konstanten und Faktoren irrelevant werden, interessiert wie die Funktion skaliert/wächst.

Definition 7: $\mathcal{O}$-Notation

Es gilt $T(n) \in \mathcal{O}(f(n))$, wenn $f(n)$ eine obere Schranke von $T(n)$ ist, also:

$$T(n) \in \mathcal{O}(f(n)) \iff \exists n_0 \in \mathbb{N}\, c \in \mathbb{R}^+ \, \forall n \geq n_0 : T(n) \leq c \cdot f(n).$$

ERFASSEN DER KOMPLEXITÄT

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel:

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen,

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$
Bsp:	42
Bez:	konst.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$
Bsp:	42	$4 \log(3n)$
Bez:	konst.	logarithm.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Bsp:	42	$4 \log(3n)$	$4n - 3$
Bez:	konst.	logarithm.	linear

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$
Bez:	konst.	logarithm.	linear	linear log.

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$	$n^3 - 42n^2$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$	$n^3 - 42n^2$	$14 \cdot 2^n$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch	exponentiell

ERFASSEN DER KOMPLEXITÄT

- Bei der Berechnung helfen gängige mathematische Gesetze (Logarithmus, …)
- Die wichtigste Rechenregel: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max\{f(n), g(n)\})$
- Neben der $\mathcal{O}$ Notation, existieren noch weitere Notationen, wie $\Omega(n)$, welches analog die untere Grenze darstellt.
- In der Regel reichen die folgenden wichtigsten Komplexitätsklassen:

	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^3)$	$\mathcal{O}(2^n)$	$\mathcal{O}(n!)$
Bsp:	42	$4 \log(3n)$	$4n - 3$	$4n \log(2n)$	$n^2 + 2n - 1$	$n^3 - 42n^2$	$14 \cdot 2^n$	$n! \cdot 10^{-42}$
Bez:	konst.	logarithm.	linear	linear log.	quadratisch	kubisch	exponentiell	faktoriell

UML – GRUNDLAGEN

UML – GRUNDLAGEN

- Die Unified Modeling Language (UML) ist eine Kollektion an UML-Diagrammarten,

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.

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:

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.

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.

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.

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 – 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 – EIN ÜBERBLICK

UML

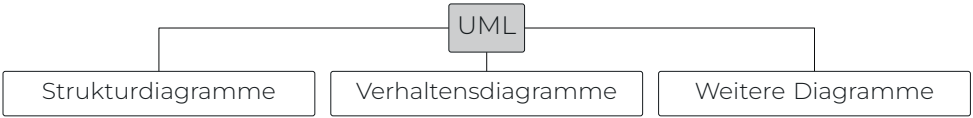
UML – EIN ÜBERBLICK

UML

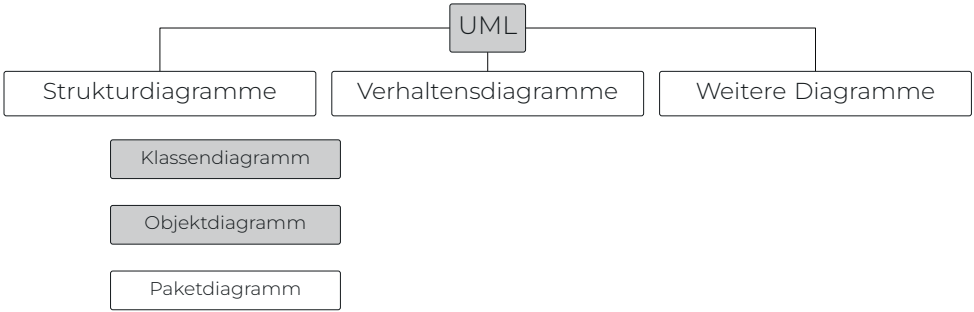
Strukturdiagramme

UML – EIN ÜBERBLICK

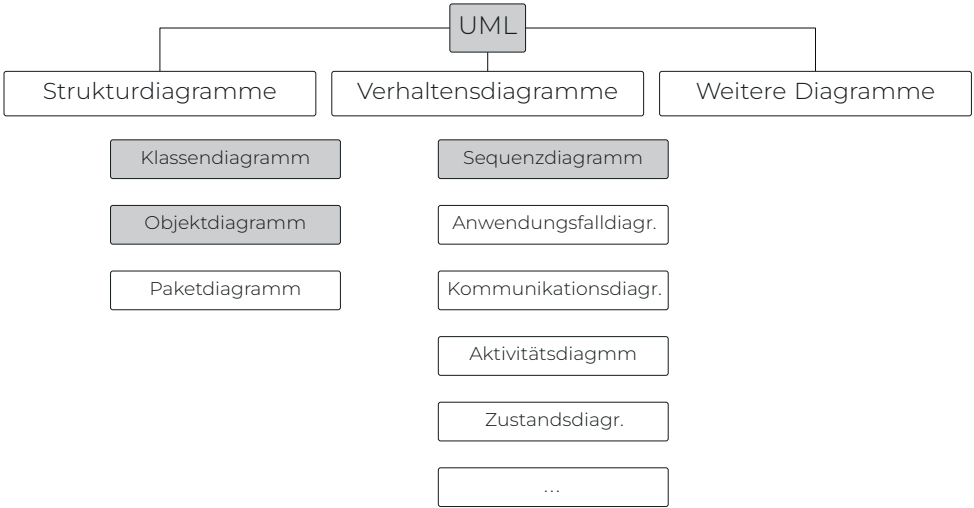
UML – EIN ÜBERBLICK



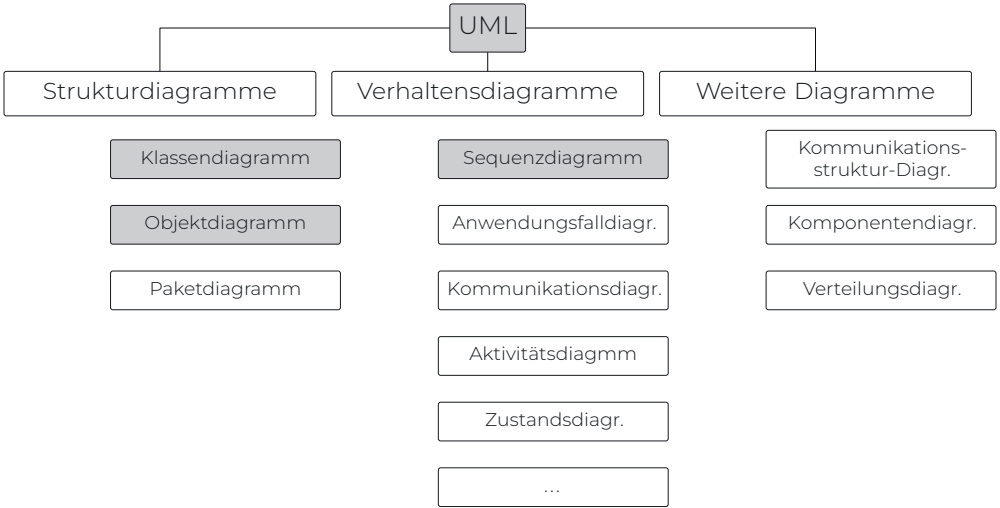
UML – EIN ÜBERBLICK



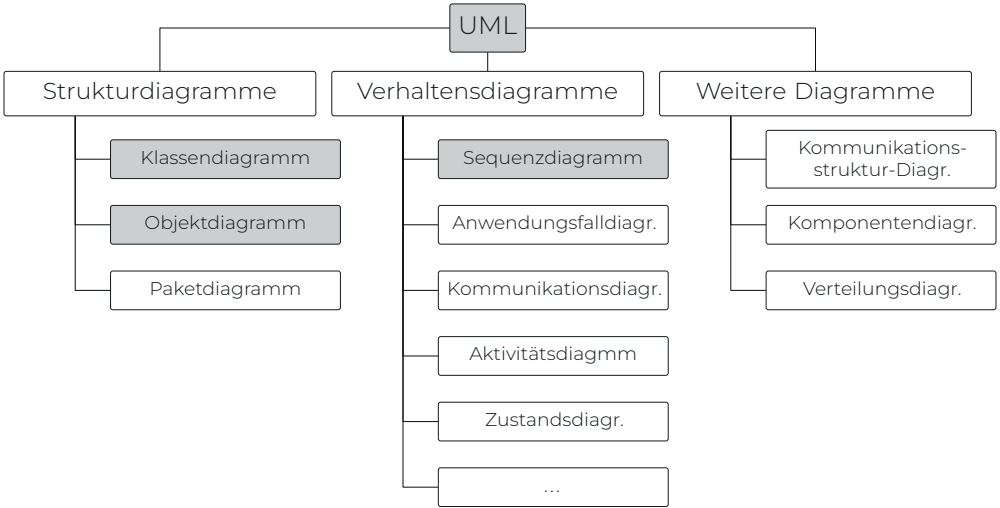
UML – EIN ÜBERBLICK



UML – EIN ÜBERBLICK

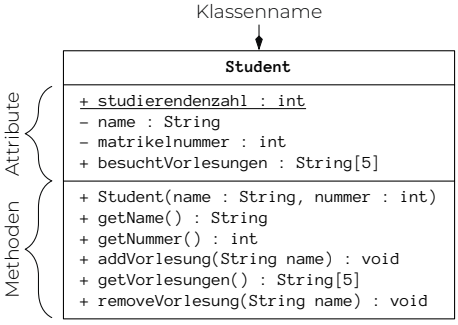


UML – EIN ÜBERBLICK



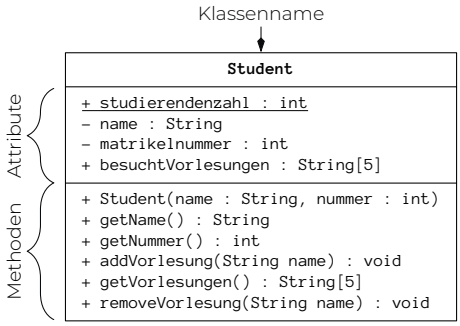
UML – KLASSENDIAGRAMME

UML – KLASSENDIAGRAMME



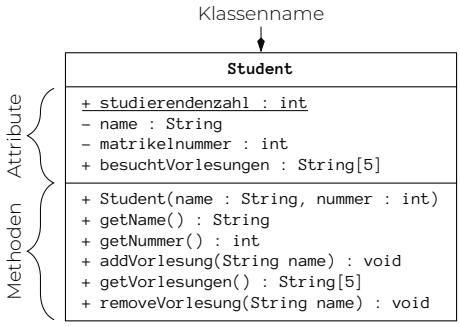
(**–**) steht für ein **`private`** Attribut
(**+**) für ein **`public`** Attribut
(**#**) für ein **`protected`** Attribut
`static`-Komponenten werden unterstrichen.

UML – KLASSENDIAGRAMME



(**–**) steht für ein **`private`** Attribut
(**+**) für ein **`public`** Attribut
(**#**) für ein **`protected`** Attribut
`static`-Komponenten werden unterstrichen.

UML – KLASSENDIAGRAMME

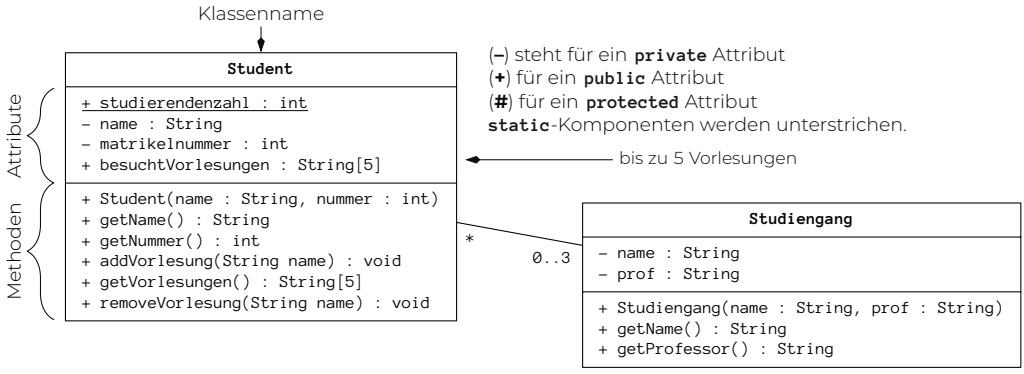


(**–**) steht für ein **`private`** Attribut
(**+**) für ein **`public`** Attribut
(**#**) für ein **`protected`** Attribut
`static`-Komponenten werden unterstrichen.

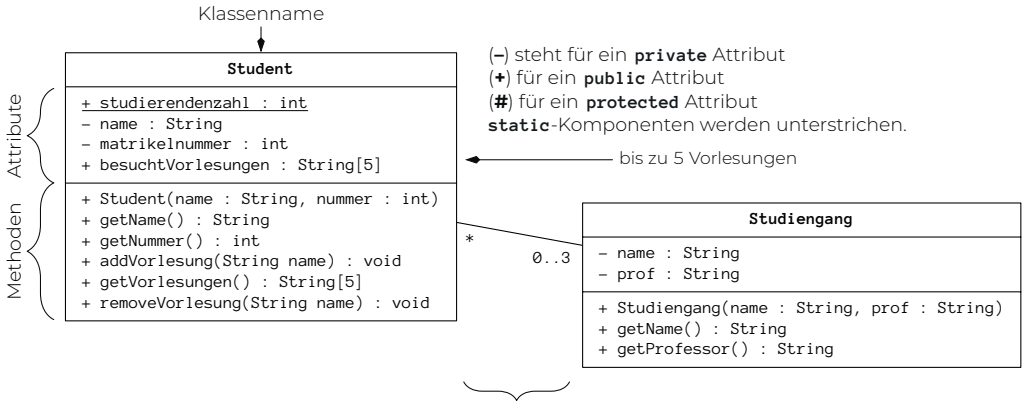
bis zu 5 Vorlesungen

Studiengang
- name : String - prof : String
+ Studiengang(name : String, prof : String) + getName() : String + getProfessor() : String

UML – KLASSENDIAGRAMME



UML – KLASSENDIAGRAMME



UML – KLASSENDIAGRAMME, II

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
  - gerichtete Assoziationen (A $\longrightarrow$ B).

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
  - gerichtete Assoziationen (A $\longrightarrow$ B).
  - Abhängigkeiten (A - - - - - - B).

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
  - gerichtete Assoziationen (A $\longrightarrow$ B).
  - Abhängigkeiten (A $\dashrightarrow$ B).
  - Vererbungen (A $\longrightarrow\!\triangleright$ B).

UML – KLASSENDIAGRAMME, II

- Es gibt noch weitere Assoziationen.
- So gibt es:
  - gerichtete Assoziationen (A ⟶ B).
  - Abhängigkeiten (A ⇢ B).
  - Vererbungen (A ⟶▷ B).
  - Aggregationen (A ◇⟶ B).

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“.

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).

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“.

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.

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,

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

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

UML – OBJEKTDIAGRAMME

- Objektdiagramme ähneln Klassendiagrammen.

UML – OBJEKTDIAGRAMME

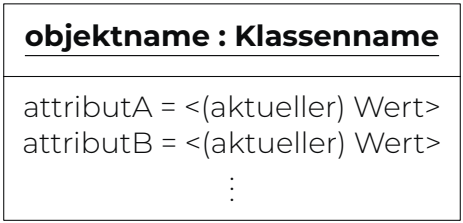
- Objektdiagramme ähneln Klassendiagrammen.
- Allerdings handelt es sich immer um explizite Ausprägungen einer Klasse

UML – OBJEKTDIAGRAMME

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

UML – OBJEKTDIAGRAMME

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



UML – SEQUENZDIAGRAMME

UML – SEQUENZDIAGRAMME

- Bilden den Nachrichtenaustausch ab.

UML – SEQUENZDIAGRAMME

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

UML – SEQUENZDIAGRAMME

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

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.

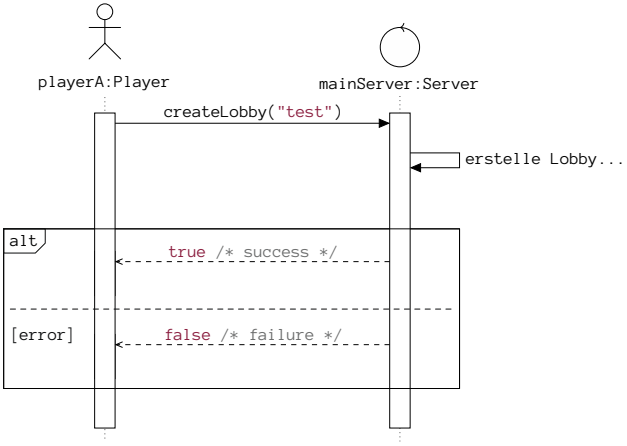
UML – SEQUENZDIAGRAMME

- 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).

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

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

Definition 8: Stellenwertsystem

Bei einer Basis b und einer Zahl z aus den Ziffern $z = z_n z_{n-1} \dots z_0$

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

Definition 8: Stellenwertsystem

Bei einer Basis b und einer Zahl z aus den Ziffern $z = z_n z_{n-1} \dots z_0$ ergibt sich ihr Wert im uns bekannten Dezimalsystem ($b = 10$) durch:

DARSTELLUNG GANZER ZAHLEN

- Um Zahlen darzustellen verwenden wir ein *Stellenwertsystem*.
- Das bedeutet der Wert einer Ziffer ergibt sich nicht nur über die Basis sondern auch über die Stelle.

Definition 8: Stellenwertsystem

Bei einer Basis b und einer Zahl z aus den Ziffern $z = z_n z_{n-1} \dots z_0$ ergibt sich ihr Wert im uns bekannten Dezimalsystem ($b = 10$) durch:

$$z_b = \sum_{i=0}^{n} z_i \cdot b^i$$

DARSTELLUNG GANZER ZAHLEN, II

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind:

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär),

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal)

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \mathrel{\widehat{=}} 10$ bis $F \mathrel{\widehat{=}} 15$ dargestellt.

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \mathrel{\widehat{=}} 10$ bis $F \mathrel{\widehat{=}} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren,

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \mathrel{\widehat{=}} 10$ 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

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.

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:

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:

$$10 \div 2 = 5 \qquad 10 \bmod 2 = 0 \ (\leftarrow \mathit{LSB})$$

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \mathrel{\widehat{=}} 10$ bis $F \mathrel{\widehat{=}} 15$ dargestellt.
- Wir können eine Zahl aus dem Dezimalsystem in jedes andere konvertieren, indem wir sukzessiv die Ziffern durch „ $\mathbf{mod}\ b$“ generieren und die Zahl dann (ohne Rest) durch b teilen.
- Beispiel: $z = 10$ soll ins Dualsystem konvertiert werden:

$$10 \div 2 = 5 \qquad 10 \bmod 2 = 0 \ (\leftarrow \mathit{LSB})$$
$$5 \div 2 = 2 \qquad 5 \bmod 2 = 1$$

DARSTELLUNG GANZER ZAHLEN, II

- Die wichtigsten Basen für uns sind: $b = 2$ (dual/binär), $b = 8$ (Oktal) und $b = 16$ (Hexadezimal)
- Im Hexadezimalsystem werden die Ziffern ≥ 10 durch die Buchstaben $A \mathrel{\widehat{=}} 10$ 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:

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

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

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren,

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren, die es zum Beispiel erlauben, das Hexadezimalsystem direkt ins Dualsystem zu konvertieren.

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren, die es zum Beispiel erlauben, das Hexadezimalsystem direkt ins Dualsystem zu konvertieren. Da jede Ziffer im Hexadezimalsystem vier Bits einnimmt, geht die Konvertierung schneller:

DARSTELLUNG GANZER ZAHLEN, III

- Dies ergibt: $1010_{(2)}$.
- Hinweis: es existieren weitere (schnelle) Konvertierungsverfahren, die es zum Beispiel erlauben, das Hexadezimalsystem direkt ins Dualsystem zu konvertieren. Da jede Ziffer im Hexadezimalsystem vier Bits einnimmt, geht die Konvertierung schneller:

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

Weiterführende Konzepte

6

SUCHVERFAHREN FORMAL

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen,

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.
- Es gibt zwei Varianten:

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.
- Es gibt zwei Varianten:

  EINFACH: Durchlaufen Suchraum auf Basis einer Datenstruktur.

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.

- Es gibt zwei Varianten:

  EINFACH: Durchlaufen Suchraum auf Basis einer Datenstruktur.

  HEURISTISCH: Besitzen zusätzliches Wissen (Verteilung, …) über die Daten, die sie zur Beschleunigung der Suche verwenden.

SUCHVERFAHREN FORMAL

- Suchverfahren haben die Aufgabe in einer Folge an Elementen, die Elemente zu finden, welche einem Muster oder gewissen Eigenschaften entsprechen.
- Es gibt zwei Varianten:

  EINFACH: Durchlaufen Suchraum auf Basis einer Datenstruktur.

  HEURISTISCH: Besitzen zusätzliches Wissen (Verteilung, …) über die Daten, die sie zur Beschleunigung der Suche verwenden.
- Die zu suchenden *Eigenschaften*, nennt man auch *Merkmale*.

KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

- Uns sind zwei Verfahren bekannt, Elemente zu suchen:

KLASSISCHE SUCHVERFAHREN

- Uns sind zwei Verfahren bekannt, Elemente zu suchen:

  LINEAR: Die lineare (naive/sequentielle) Suche beginnt am Anfang der Folge und prüft Element für Element ob es sich um das gesuchte handelt.

KLASSISCHE SUCHVERFAHREN

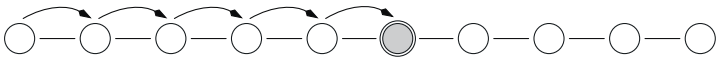
- Uns sind zwei Verfahren bekannt, Elemente zu suchen:

  LINEAR: Die lineare (naive/sequentielle) Suche beginnt am Anfang der Folge und prüft Element für Element ob es sich um das gesuchte handelt. Die Komplexität beträgt $\mathcal{O}(n)$.

KLASSISCHE SUCHVERFAHREN

- Uns sind zwei Verfahren bekannt, Elemente zu suchen:

  LINEAR: Die lineare (naive/sequentielle) Suche beginnt am Anfang der Folge und prüft Element für Element ob es sich um das gesuchte handelt. Die Komplexität beträgt $\mathcal{O}(n)$.



KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

```
int linearSearch(int[] sequence, int key){


}
```

KLASSISCHE SUCHVERFAHREN

```
int linearSearch(int[] sequence, int key){
    for(int i = 0; i < sequence.length; i++)


}
```

KLASSISCHE SUCHVERFAHREN

```
int linearSearch(int[] sequence, int key){
    for(int i = 0; i < sequence.length; i++)
        if(sequence[i] == key) return i;

}
```

KLASSISCHE SUCHVERFAHREN

```
int linearSearch(int[] sequence, int key){
    for(int i = 0; i < sequence.length; i++)
        if(sequence[i] == key) return i;
    return -1; // Magic Number für "nicht gefunden"
}
```

KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

Binär: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.

KLASSISCHE SUCHVERFAHREN

Binär: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.

Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff.

KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.

Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht.

KLASSISCHE SUCHVERFAHREN

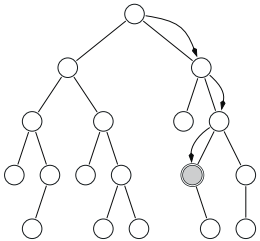
Binär: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.

Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht. Die Komplexität beträgt $\mathcal{O}(\log n)$ (ohne Sortieren), da jeder Vergleich den restlichen Suchbereich halbiert.

KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.

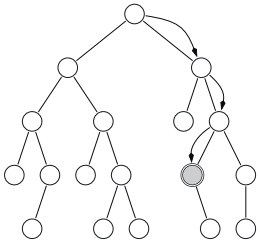
Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht. Die Komplexität beträgt $\mathcal{O}(\log n)$ (ohne Sortieren), da jeder Vergleich den restlichen Suchbereich halbiert.

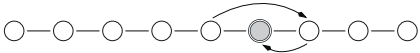


KLASSISCHE SUCHVERFAHREN

BINÄR: Durchsucht eine (z.B. aufsteigend) *sortierte* Folge baumartig.

Vergleicht das mittlere Element des aktuellen Suchbereiches mit dem Suchbegriff. Ist er größer, wird rechts, ist er kleiner, wird links von der Mitte weiter gesucht. Die Komplexität beträgt $\mathcal{O}(\log n)$ (ohne Sortieren), da jeder Vergleich den restlichen Suchbereich halbiert.





KLASSISCHE SUCHVERFAHREN

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){









}
```

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){
    int min = 0; // Suche von min
    int max = sequence.length - 1; // Suche bis max









}
```

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){
    int min = 0; // Suche von min
    int max = sequence.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.









    }

}
```

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){
    int min = 0; // Suche von min
    int max = sequence.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte









    }

}
```

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){
    int min = 0; // Suche von min
    int max = sequence.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(sequence[middle] == key) // gefunden
            return middle;









    }

}
```

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){
    int min = 0; // Suche von min
    int max = sequence.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(sequence[middle] == key) // gefunden
            return middle;
        else if(key < sequence[middle]) // Suche links [min, middle-1]
            max = middle - 1;


    }

}
```

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){
    int min = 0; // Suche von min
    int max = sequence.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(sequence[middle] == key) // gefunden
            return middle;
        else if(key < sequence[middle]) // Suche links [min, middle-1]
            max = middle - 1;
        else // Suche rechts [middle+1, max]
            min = middle + 1;
    }

}
```

KLASSISCHE SUCHVERFAHREN

```
int binarySearch(int[] sequence, int key){
    int min = 0; // Suche von min
    int max = sequence.length - 1; // Suche bis max
    while(min <= max) { // Solange [min, max] nicht ungültig ist.
        int middle = (min + max)/2; // Mitte
        if(sequence[middle] == key) // gefunden
            return middle;
        else if(key < sequence[middle]) // Suche links [min, middle-1]
            max = middle - 1;
        else // Suche rechts [middle+1, max]
            min = middle + 1;
    }
    return -1; // Magic Number für "nicht gefunden"
}
```

SORTIERVERFAHREN FORMAL

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:

  Intern: Hier liegt der gesamte Datenbestand im Arbeitsspeicher,

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:

  INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, …)

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:

  INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, …)

  EXTERN: Der Datenbestand ist (überwiegend) ausgelagert.

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:

  INTERN: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, …)

  EXTERN: Der Datenbestand ist (überwiegend) ausgelagert. Für das Sortieren werden meist nur die obersten Element eines Stapels betrachtet (Dateien, …).

SORTIERVERFAHREN FORMAL

- Ein Prozess, bei dem Elemente auf Basis eines Ordnungskriteriums in eine Reihenfolge gebracht werden.
- Daten werden sortiert um beispielsweise Suchen zu vereinfachen.
- Wir unterscheiden zwei Arten:

  Intern: Hier liegt der gesamte Datenbestand im Arbeitsspeicher, alle Elemente sind zugreifbar (Array, …)

  Extern: Der Datenbestand ist (überwiegend) ausgelagert. Für das Sortieren werden meist nur die obersten Element eines Stapels betrachtet (Dateien, …). Oft erlauben Hilfsmethoden die selben Verfahren wie bei der internen Variante.

SORTIERVERFAHREN FORMAL, II

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

  Sukzessiv: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

  Sukzessiv: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

  Sukzessiv: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

  Divide-and-Conquer: Daten werden aufgeteilt,

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

  Sukzessiv: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

  Divide-and-Conquer: Daten werden aufgeteilt, in Teilen sortiert und dann aus sortierten Teilmengen zusammengefügt.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

  Sukzessiv: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

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

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

  Sukzessiv: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

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

  Halde: Sortieren mit dafür geeigneten Datenstrukturen wie dem *Heap*.

SORTIERVERFAHREN FORMAL, II

- Bei der Vorgehensweise unterscheiden wir (im Kontext der Vorlesung) drei Ansätze:

  Sukzessiv: Schritt für Schritt wird die Anzahl an unsortierten Elementen verringert. In der Regel iterativ implementiert.

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

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

SORTIERVERFAHREN: SELECTIONSORT

SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort (iterativ)

SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort (iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil,

SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort (iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort (iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.

SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort (iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

SORTIERVERFAHREN: SELECTIONSORT

Definition 9: Selectionsort (iterativ)

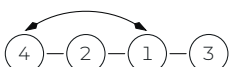
Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).

Definition 9: Selectionsort (iterativ)

Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

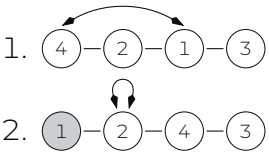
- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).

1. 

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).



Definition 9: Selectionsort (iterativ)

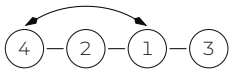
Prinzip: Wahl des kleinsten Elements im unsortierten Teil, Anfügen des Elements als größtes Element des sortierten Teils.

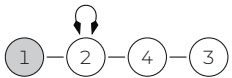
- Kann auch von oben nach unten implementiert werden.
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Ist das kleinste Element an erster Stelle des unsortierten Teils wird dennoch getauscht (nach Vorlesung).

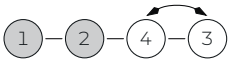
Definition 9: Selectionsort (iterativ)

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. 

SORTIERVERFAHREN: SELECTIONSORT

SORTIERVERFAHREN: SELECTIONSORT

```java
public static void selectionSort(int[] arr) {




}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {









    }
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        int min = i; // Suche Minimum









    }
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        int min = i; // Suche Minimum
        for (int j = i + 1; j < arr.length; j++)









    }
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        int min = i; // Suche Minimum
        for (int j = i + 1; j < arr.length; j++)
            if(arr[j] < arr[min])
                min = j;









    }
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        int min = i; // Suche Minimum
        for (int j = i + 1; j < arr.length; j++)
            if(arr[j] < arr[min])
                min = j;
        // Tausche 'i' und 'min'




    }
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        int min = i; // Suche Minimum
        for (int j = i + 1; j < arr.length; j++)
            if(arr[j] < arr[min])
                min = j;
        // Tausche 'i' und 'min'
        int tmp = arr[i];


    }
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        int min = i; // Suche Minimum
        for (int j = i + 1; j < arr.length; j++)
            if(arr[j] < arr[min])
                min = j;
        // Tausche 'i' und 'min'
        int tmp = arr[i];
        arr[i] = arr[min];

    }
}
```

SORTIERVERFAHREN: SELECTIONSORT

```
public static void selectionSort(int[] arr) {
    for(int i = 0; i < arr.length - 1; i++) {
        int min = i; // Suche Minimum
        for (int j = i + 1; j < arr.length; j++)
            if(arr[j] < arr[min])
                min = j;
        // Tausche 'i' und 'min'
        int tmp = arr[i];
        arr[i] = arr[min];
        arr[min] = tmp;
    }
}
```

SORTIERVERFAHREN: INSERTIONSORT

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort (iterativ)

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort (iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil,

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort (iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort (iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort (iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.

SORTIERVERFAHREN: INSERTIONSORT

Definition 10: Insertionsort (iterativ)

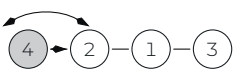
Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.

Definition 10: Insertionsort (iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

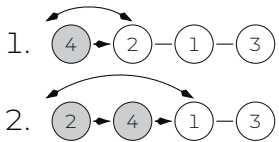
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.

1. 

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.



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.

Definition 10: Insertionsort (iterativ)

Prinzip: Wahl des ersten Elements im unsortierten Teil, Sortieren des Elements in den bereits sortierten Teil.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Das Tauschen erfolgt durch durchgehende Vertauschungen.
- Ist das Element schon an der richtigen Position, wird dennoch getauscht.

SORTIERVERFAHREN: INSERTIONSORT

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {




}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {
    for(int i = 1; i < arr.length; i++) {









    }
}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int pos = i - 1, elem = arr[i];









    }
}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int pos = i - 1, elem = arr[i];
        while(pos >= 0 && arr[pos] > elem) { // Verschiebe


        }

    }
}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int pos = i - 1, elem = arr[i];
        while(pos >= 0 && arr[pos] > elem) { // Verschiebe
            arr[pos + 1] = arr[pos];
            pos--;
        }

    }
}
```

SORTIERVERFAHREN: INSERTIONSORT

Hier eine Variante, die verschiebt:

```
public static void insertionSort(int[] arr) {
    for(int i = 1; i < arr.length; i++) {
        int pos = i - 1, elem = arr[i];
        while(pos >= 0 && arr[pos] > elem) { // Verschiebe
            arr[pos + 1] = arr[pos];
            pos--;
        }
        arr[pos+1] = elem;
    }
}
```

SORTIERVERFAHREN: BUBBLESORT

SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort (iterativ)

SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort (iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort (iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

SORTIERVERFAHREN: BUBBLESORT

Definition 11: Bubblesort (iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Mit jedem Durchgang über das Array wird (mindestens) ein Element an die richtige Position getauscht.

Definition 11: Bubblesort (iterativ)

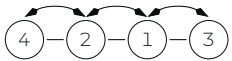
Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

- 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.

Definition 11: Bubblesort (iterativ)

Prinzip: Vertausche benachbarte Elemente wenn sie nicht in Sortierreihenfolge sind.

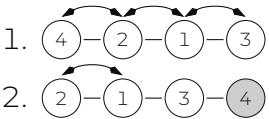
- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Mit jedem Durchgang über das Array wird (mindestens) ein Element an die richtige Position getauscht.
- (Geringfügige) Verbesserung: Shaker-Sort, schiebt abwechselnd das maximale und minimale Element an die (Ziel-)Position.

1. 

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.



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;
            }
        }
    }
}
```

SORTIERVERFAHREN: MERGESORT

SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort (rekursiv)

SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort (rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen.

SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort (rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen. Zusammenfügen sortierter Teilfolgen.

SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort (rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen. Zusammenfügen sortierter Teilfolgen.

- Verfolgt *easy split*, *hard join*. Das Aufteilen ist leicht (trivial), das Zusammenfügen aufwändig, da hier die Sortierung erfolgen muss.

SORTIERVERFAHREN: MERGESORT

Definition 12: Mergesort (rekursiv)

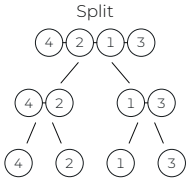
Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen. Zusammenfügen sortierter Teilfolgen.

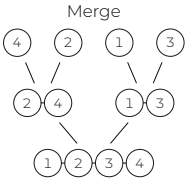
- Verfolgt *easy split*, *hard join*. Das Aufteilen ist leicht (trivial), das Zusammenfügen aufwändig, da hier die Sortierung erfolgen muss.
- Teilt sich auf in: *Split* (aufteilen) und *Merge* (verschmelzen).

Definition 12: Mergesort (rekursiv)

Prinzip: Aufteilen der Elemente in einelementige, sortierte Teilfolgen. Zusammenfügen sortierter Teilfolgen.

- Verfolgt *easy split*, *hard join*. Das Aufteilen ist leicht (trivial), das Zusammenfügen aufwändig, da hier die Sortierung erfolgen muss.
- Teilt sich auf in: *Split* (aufteilen) und *Merge* (verschmelzen).





SORTIERVERFAHREN: QUICKSORT

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort (rekursiv)

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort (rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte).

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort (rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements.

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort (rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

Definition 13: Quicksort (rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.

SORTIERVERFAHREN: QUICKSORT

Definition 13: Quicksort (rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Verfolgt *hard split*, *easy join*. Das Aufteilen ist aufwändig (sortieren), das Zusammenfügen trivial.

Definition 13: Quicksort (rekursiv)

Prinzip: Wahl eines Pivotelements (z.B. das Letzte). Aufteilen der Liste in Teil größer und kleiner des Pivotelements. Wenn Teillisten nach gleichem Prinzip sortiert: Zusammenfügen aus *Links + Pivot + Rechts*.

- Wir arbeiten *in-place*, das heißt wir tauschen die Elemente im Array.
- Verfolgt *hard split*, *easy join*. Das Aufteilen ist aufwändig (sortieren), das Zusammenfügen trivial.

SORTIERVERFAHREN: HEAPSORT

SORTIERVERFAHREN: HEAPSORT

Definition 14: Heapsort (rekursiv)

SORTIERVERFAHREN: HEAPSORT

Definition 14: Heapsort (rekursiv)

Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente.

SORTIERVERFAHREN: HEAPSORT

Definition 14: Heapsort (rekursiv)

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

SORTIERVERFAHREN: HEAPSORT

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.

SORTIERVERFAHREN: HEAPSORT

Definition 14: Heapsort (rekursiv)

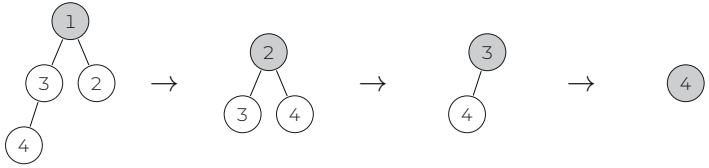
Prinzip: Verwenden eines (Min-)Heaps zur Speicherung der Elemente. Sukzessives Entfernen des Wurzelelements (aktuelles Minimum), zur Wiederherstellung der Heap-Eigenschaft.

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

Definition 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.



EXKURS: HEAP

EXKURS: HEAP

Definition 15: Heap

EXKURS: HEAP

Definition 15: Heap

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

EXKURS: HEAP

Definition 15: Heap

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

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch $\circ$).

EXKURS: HEAP

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.

EXKURS: HEAP

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…

EXKURS: HEAP

Definition 15: Heap

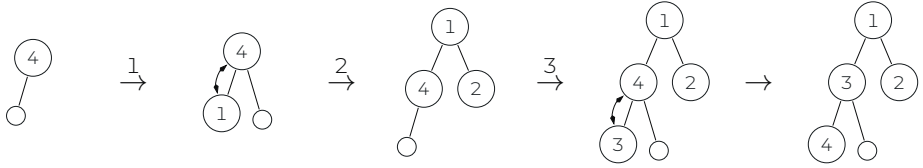
Binärbaum, mit: Eltern $\leq$ Kinder (*Min-Heap*, für *Max-Heap*: $\geq$).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch $\circ$). Verletzen diese die Heap-Eigenschaft, werden solange kleinere Kinder nach oben getauscht, bis sie wiederhergestellt ist.
- Beim Entfernen eines Elements wird das „letzte Blatt“ als neue Wurzel getauscht. Anschließend wird wieder getauscht…
- Beispiel: Einfügen der Elemente $4, 2, 1, 3$.

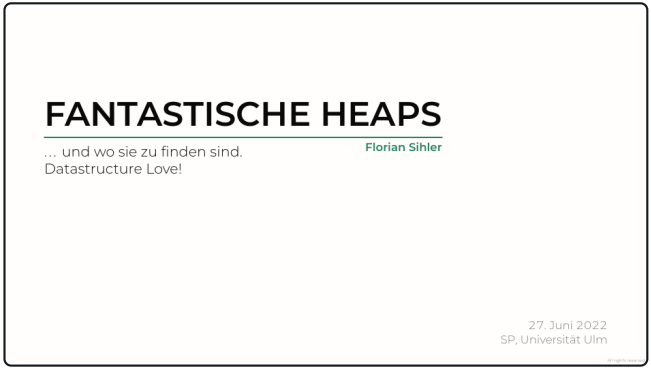
Definition 15: Heap

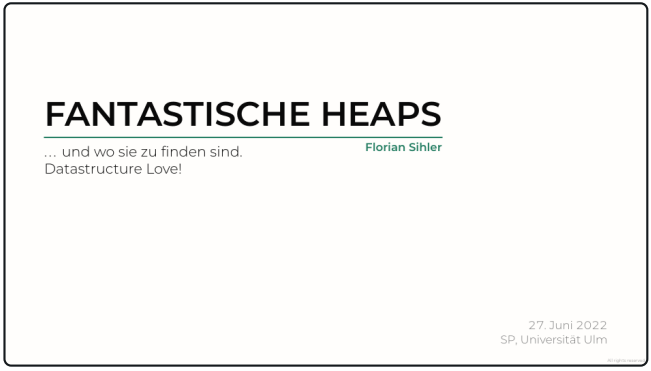
Binärbaum, mit: Eltern $\leq$ Kinder (*Min-Heap*, für *Max-Heap*: $\geq$).

- Neue Elemente werden unten als neues Kind eingefügt (markiert durch ○). Verletzen diese die Heap-Eigenschaft, werden solange kleinere Kinder nach oben getauscht, bis sie wiederhergestellt ist.
- Beim Entfernen eines Elements wird das „letzte Blatt“ als neue Wurzel getauscht. Anschließend wird wieder getauscht…
- Beispiel: Einfügen der Elemente 4, 2, 1, 3.



ZUR VERTIEFUNG


FANTASTISCHE HEAPS
... und wo sie zu finden sind.
Datastructure Love!
Florian Sihler
27. Juni 2022
SP, Universität Ulm



Mehr zu „Heaps“ per Klick…

SORTIERVERFAHREN: ZUSAMMENFASSUNG

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

EIGENSCHAFT: STABIL

Definition 16: Stabiles Sortierverfahren

EIGENSCHAFT: STABIL

Definition 16: Stabiles Sortierverfahren

Ein Sortierverfahren heißt stabil, wenn es die Reihenfolge (aus Sicht des Sortierschlüssels) gleicher Element nicht ändert.

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.

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.

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	

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):

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.

Dynamische Datenstrukturen

○ ○ ○ ○ ○ ○ 7 ○

Florian Sihler, 2.01.2022

LISTEN – DYNAMISCHE ARRAYS

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.
- Eine ArrayList ist die einfachste Implementation einer Liste.

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.
- Eine ArrayList ist die einfachste Implementation einer Liste.
- Wir halten intern ein Array, welches wir dynamisch vergrößern oder Verkleinern.

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.
- Eine ArrayList ist die einfachste Implementation einer Liste.
- Wir halten intern ein Array, welches wir dynamisch vergrößern oder Verkleinern.
- Das Problem: Einfüge- und Löschoperationen bedeuten einen großen Aufwand (durch das Kopieren der Elemente).

LISTEN – DYNAMISCHE ARRAYS

- Für eine Implementation: Betrachte das Weihnachtsblatt.
- Eine ArrayList ist die einfachste Implementation einer Liste.
- Wir halten intern ein Array, welches wir dynamisch vergrößern oder Verkleinern.
- Das Problem: Einfüge- und 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

LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.

LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **`null`**) zeigt, nennen wir *Tail* oder *Foot*.

LISTEN – EINFACH VERKETTETE LISTE

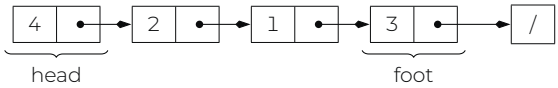
- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **`null`**) zeigt, nennen wir *Tail* oder *Foot*.

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

LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **`null`**) zeigt, nennen wir *Tail* oder *Foot*.

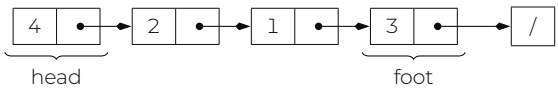
```
class Element {
    public int value;
    public Element next;
    // ...
}
```



LISTEN – EINFACH VERKETTETE LISTE

- Jedes Element besitzt als Objekt eine Referenz auf das nächste.
- Das erste Element bezeichnen wir als *Head*. Das Letzte, welches auf einen definierten letzten Wert (wie **`null`**) zeigt, nennen wir *Tail* oder *Foot*.

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



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

LISTEN – EINFACH VERKETTETE LISTE, II

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketten Liste:

LISTEN – EINFACH VERKETTETE LISTE, II

- Vorteil einer einfach verketten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$).

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.

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.

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.

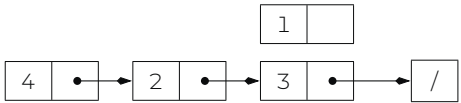
LISTEN – EINFACH VERKETTETE LISTE, II

- 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:

LISTEN – EINFACH VERKETTETE LISTE, II

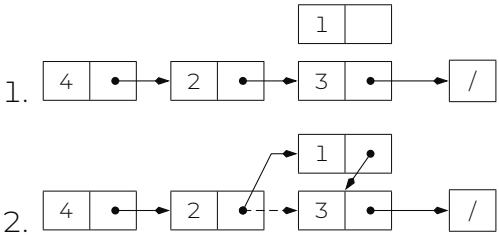
- Vorteil einer einfach verketten Liste: Das Löschen und Hinzufügen neuer Elemente ist leicht ($\mathcal{O}(1)$). Nachteil: Der direkte Zugriff ist nur durch ein Traversieren möglich.
- Hinweis: Manche Operationen benötigen eine Sonderbehandlung im Falle einer leeren Liste. So zum Beispiel das Einfügen.
- Betrachten wir einmal das Einfügen eines neuen Elements:

1.



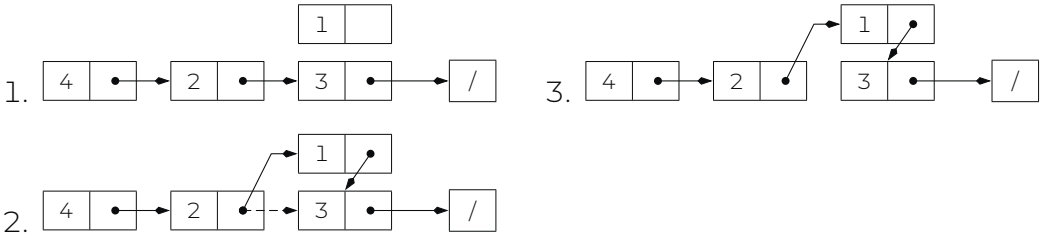
LISTEN – EINFACH VERKETTETE LISTE, II

- 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:



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:



LISTEN – DOPPELT VERKETTETE LISTE

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“.

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

LISTEN – DOPPELT VERKETTETE LISTE

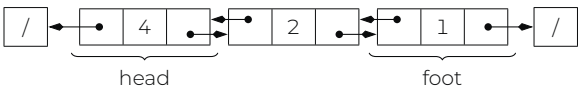
- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

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

LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

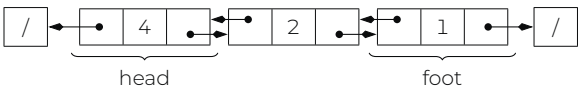
```
class Element {
    public int value;
    public Element next;
    public Element prev;
    // ...
}
```



LISTEN – DOPPELT VERKETTETE LISTE

- Mit einfach verketteten Listen können wir in „eine Richtung iterieren“. Einen Vorgänger kann man damit nur sehr aufwändig bestimmen.
- Die doppelt verkettete Liste hält nun jeweils auch die Referenz auf das vorherige Element.

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



- Den „Datentyp“ einfach verkettete Liste lagern wir in eine neue Klasse `LinkedList` aus, die auch die Operationen wie hinzufügen und löschen übernimmt.

DATENSTRUKTUR: STACK

DATENSTRUKTUR: STACK

- Ein Stack arbeitet nach dem LIFO-Prinzip (Last-In, First-Out).

DATENSTRUKTUR: STACK

- Ein Stack arbeitet nach dem LIFO-Prinzip (Last-In, First-Out).
- Elemente können nur oben auf dem Stapel abgelegt und von dort entnommen werden.

DATENSTRUKTUR: STACK

- Ein Stack arbeitet nach dem LIFO-Prinzip (Last-In, First-Out).
- Elemente können nur oben auf dem Stapel abgelegt und von dort entnommen werden.
- Die Verwaltung rekursiver Methoden erfolgt über eine Art Stack. So „überlagern“ die Parameter im rekursiven Aufruf die alten, bis die Methode wieder verlassen wird.

DATENSTRUKTUR: STACK, II

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren,

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.
- Auf Stacks kennzeichnen wir zwei wichtige Operationen:

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.
- Auf Stacks kennzeichnen wir zwei wichtige Operationen:
  PUSH: Legt das Element oben auf dem Stack ab.

DATENSTRUKTUR: STACK, II

- Stacks lassen sich als Listen implementieren, die nur Zugriffe wie `appendLast(Element)` und `removeLast(Element)` zulassen.
- Auf Stacks kennzeichnen wir zwei wichtige Operationen:
  - PUSH: Legt das Element oben auf dem Stack ab.
  - POP: Entfernt das oberste Element und liefert es zurück.

DATENSTRUKTUR: QUEUE

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).
- Elemente können nur hinten in der Schlange eingereiht und nur vorne von der Schlange entfernen werden.

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).
- Elemente können nur hinten in der Schlange eingereiht und nur vorne von der Schlange entfernen werden.
- Queues können zum Puffern verwendet werden (analog zum Schalter).

DATENSTRUKTUR: QUEUE

- Eine Warteschlange arbeitet nach dem FIFO-Prinzip (First-In, First-Out).
- Elemente können nur hinten in der Schlange eingereiht und nur vorne von der Schlange entfernen werden.
- Queues können zum Puffern verwendet werden (analog zum Schalter).
- Es existieren Derivate, wie die *double-ended queue* (ein- und ausreihen auf beiden Seiten) oder die *priority queue* (Elemente mit höherer Priorität werden bei `enqueue` zuerst aus der Schlange genommen).

DATENSTRUKTUR: QUEUE, II

DATENSTRUKTUR: QUEUE, II

- Queues lassen sich als eine Liste implementieren,

DATENSTRUKTUR: QUEUE, II

- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.

DATENSTRUKTUR: QUEUE, II

- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.
- Auf Queues kennzeichnen wir zwei wichtige Operationen:

DATENSTRUKTUR: QUEUE, II

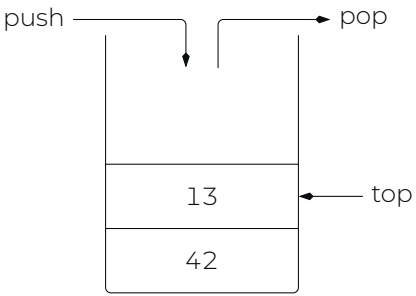
- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.
- Auf Queues kennzeichnen wir zwei wichtige Operationen:
  ENQUEUE: Reiht ein Element (hinten) in die Schlange ein.

DATENSTRUKTUR: QUEUE, II

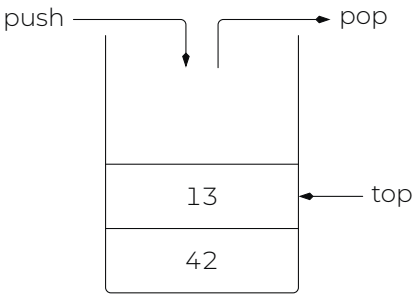
- Queues lassen sich als eine Liste implementieren, die nur Zugriffe wie `prependFirst(Element)` und `removeLast(Element)` zulässt.
- Auf Queues kennzeichnen wir zwei wichtige Operationen:
  ENQUEUE: Reiht ein Element (hinten) in die Schlange ein.
  DEQUEUE: Nimmt ein Element (vorne) aus der Schlange.

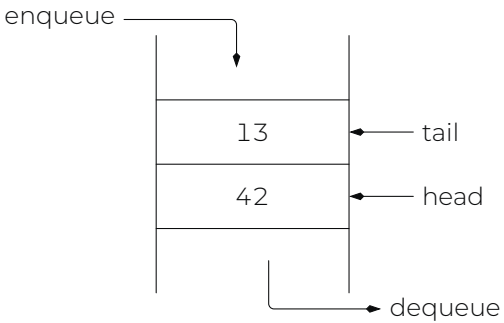
VERGLEICH: STACK UND QUEUE

VERGLEICH: STACK UND QUEUE



VERGLEICH: STACK UND QUEUE





BÄUME: FORMAL

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:

  WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:

  WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.

  BLATT: Ein Blatt ist ein Knoten ohne Nachfolger.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:

  WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.

  BLATT: Ein Blatt ist ein Knoten ohne Nachfolger.

  INNERER: Ein innerer Knoten ist jeder Knoten, der weder Blatt noch Wurzel ist.

BÄUME: FORMAL

- Im Gegensatz zu Knoten in Listen halten Knoten in Bäumen Referenzen auf mehrere Nachfolger.
- Ein Spezialfall sind Binärbäume, die maximal zwei Nachfolger haben.
- Knoten erhalten verschiedene Bezeichner:

  WURZEL: Ein Knoten ohne Vorgänger, der den Baum anführt. Ein Baum hat *eine* Wurzel.

  BLATT: Ein Blatt ist ein Knoten ohne Nachfolger.

  INNERER: Ein innerer Knoten ist jeder Knoten, der weder Blatt noch Wurzel ist. Ein innerer Knoten hat genau einen Elternknoten.

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 $E \subseteq V \times V$.

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.

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.

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.

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

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)

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).

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.

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.

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

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.
- Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kinder.

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.
- Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kinder.
- Binärbäume sind von einer besonderen Bedeutung für die Informatik.

BÄUME: MATHEMATISCH, II

- Die Höhe des Baumes ist die tiefste Ebene auf der ein Knoten existiert.
- Der *Verzweigungsgrad* eines Knotens ist die Anzahl seiner Kinder.
- Binärbäume sind von einer besonderen Bedeutung für die Informatik.
  (Sie lassen sich z.B. einfach in einem Array repräsentieren.)

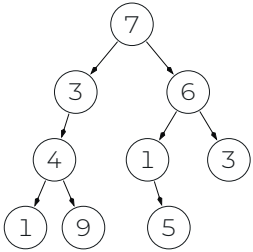
BÄUME: IMPLEMENTATION

BÄUME: IMPLEMENTATION

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

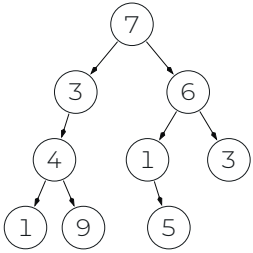
BÄUME: IMPLEMENTATION

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



BÄUME: IMPLEMENTATION

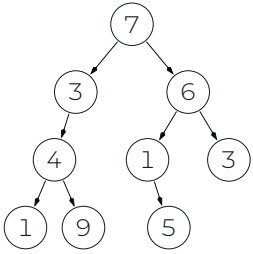
```
class Node {
    public int value;
    public Node left;
    public Node right;
    // ...
}
```



- Wie bei einer einfach verketteten Liste können wir den Baum traversieren.

BÄUME: IMPLEMENTATION

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



- Wie bei einer einfach verketteten Liste können wir den Baum traversieren.
- Analog zur doppelt verketteten Liste gibt es einen Binärbaum mit Verweis auf den Elternknoten.

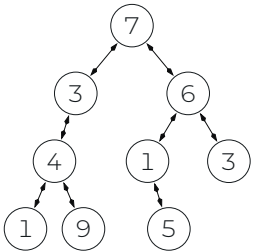
BÄUME: IMPLEMENTATION, II

BÄUME: IMPLEMENTATION, II

```
class Node {
    public Node parent;

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

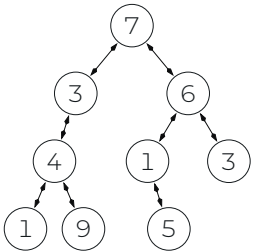
BÄUME: IMPLEMENTATION, II

```
class Node {
    public Node parent;

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



BÄUME: IMPLEMENTATION, II

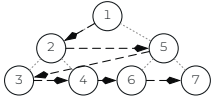
```
class Node {
    public Node parent;

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



- Diese Verkettung kann vorteilhaft sein und erspart beispielsweise die Rekursion bei der Traversierung.

TRAVERSIERUNGSVERFAHREN

TRAVERSIERUNGSVERFAHREN



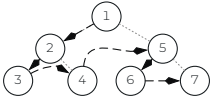
Breitendurchlauf

Ebenen

Tiefendurchlauf

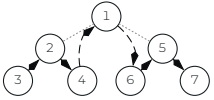
Pre-Order

root, left, right



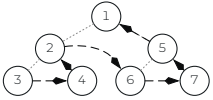
In-Order

left, root, right



Post-Order

left, right, root



TRAVERSIERUNG: LEVEL-ORDER

TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.

TRAVERSIERUNG: LEVEL-ORDER

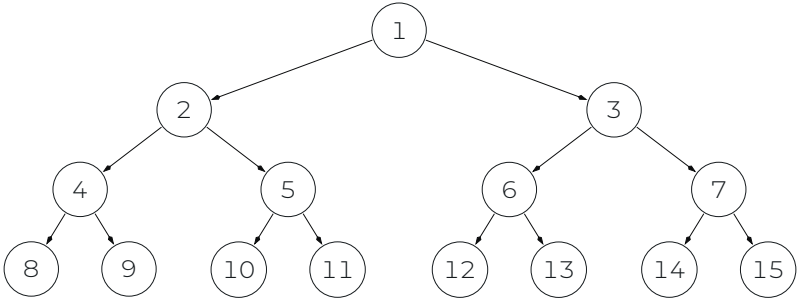
- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.

TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.
- Es handelt sich um einen Breitendurchlauf (*breadth-first*).

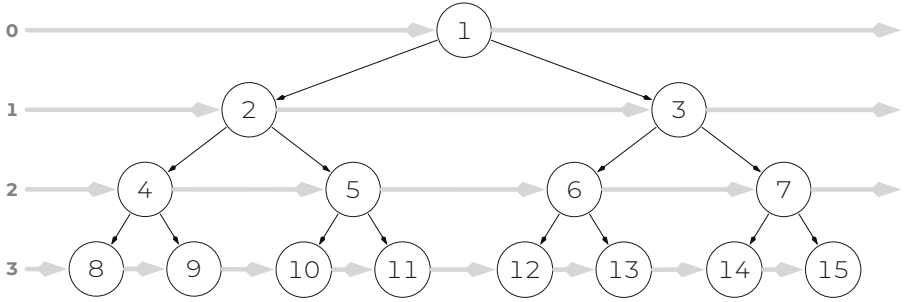
TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.
- Es handelt sich um einen Breitendurchlauf (*breadth-first*).



TRAVERSIERUNG: LEVEL-ORDER

- Die Knoten werden Ebene für Ebene von links nach rechts besucht.
- „Besucht“ steht hier für alle möglichen Varianten der Bearbeitung.
- Es handelt sich um einen Breitendurchlauf (*breadth-first*).



TRAVERSIERUNG: PRE-ORDER

TRAVERSIERUNG: PRE-ORDER

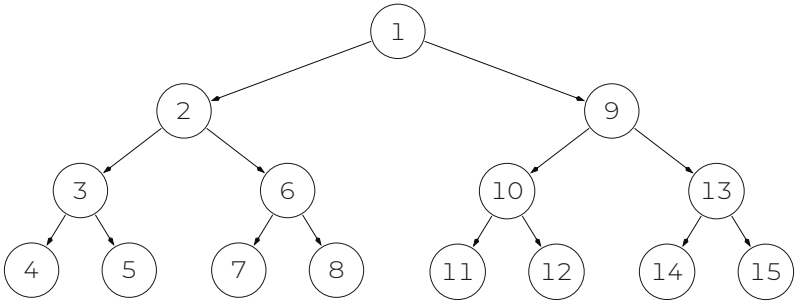
- Es handelt sich um einen Tiefendurchlauf (*depth-first*).

TRAVERSIERUNG: PRE-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit der Pre-Order Strategie werden erst der aktuelle Knoten und dann rekursiv beide Teilbäume nach der gleichen Regel besucht.

TRAVERSIERUNG: PRE-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit der Pre-Order Strategie werden erst der aktuelle Knoten und dann rekursiv beide Teilbäume nach der gleichen Regel besucht.



TRAVERSIERUNG: IN-ORDER

TRAVERSIERUNG: IN-ORDER

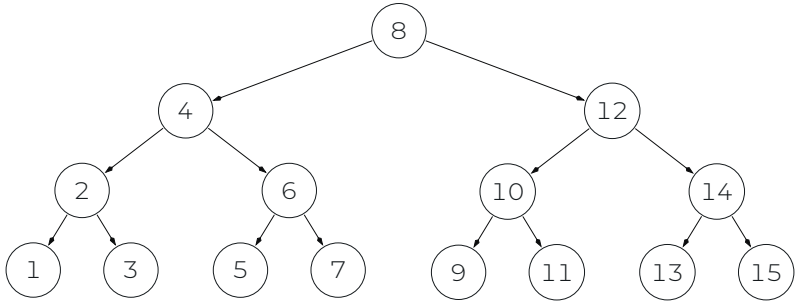
- Es handelt sich um einen Tiefendurchlauf (*depth-first*).

TRAVERSIERUNG: IN-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit dieser Strategie wird erst rekursiv ein Teilbaum bearbeitet, dann der Knoten selbst und dann rekursiv der andere Teilbaum.

TRAVERSIERUNG: IN-ORDER

- Es handelt sich um einen Tiefendurchlauf (*depth-first*).
- Mit dieser Strategie wird erst rekursiv ein Teilbaum bearbeitet, dann der Knoten selbst und dann rekursiv der andere Teilbaum.



TRAVERSIERUNG: POST-ORDER

TRAVERSIERUNG: POST-ORDER

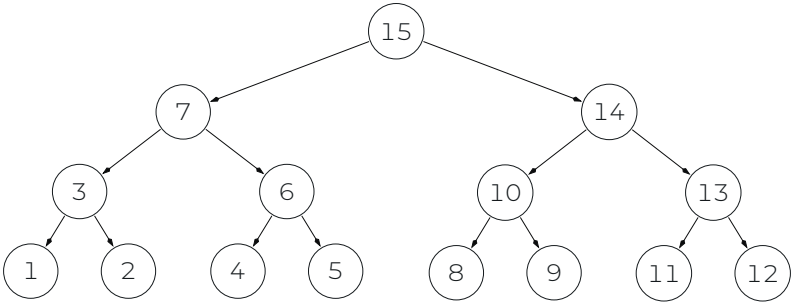
- Es handelt sich um einen Tiefendurchlauf (*depth-first*).

TRAVERSIERUNG: POST-ORDER

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

TRAVERSIERUNG: POST-ORDER

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



ZUR VERTIEFUNG


TRAVERSIERUNGSVARIANTEN
Verwandte Besuchen: Wie, wann und wo ein gutes Kind zu den geliebten Eltern rennt.
Florian Sihler
13. Juli 2022
SP, Universität Ulm

ZUR VERTIEFUNG

Mehr zu „Traversierungen“ per Klick…

SUCHBÄUME

SUCHBÄUME

- Bäume können auch zur Sortierung nützliche Datenstrukturen sein.

SUCHBÄUME

- Bäume können auch zur Sortierung nützliche Datenstrukturen sein.
- So existieren Suchbäume. (Alle Elemente im linken Teilbaum sind kleiner oder gleich der Wurzel, alle rechten größer ⇒ binäre Suche.)

SUCHBÄUME

- Bäume können auch zur Sortierung nützliche Datenstrukturen sein.
- So existieren Suchbäume. (Alle Elemente im linken Teilbaum sind kleiner oder gleich der Wurzel, alle rechten größer ⇒ binäre Suche.)
- Vergleiche hierzu auch die Datenstruktur Heap.

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!

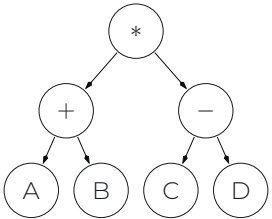
ARITHMETISCHE BÄUME

ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.

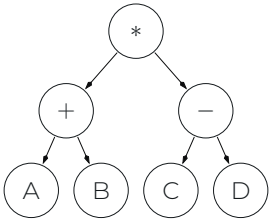
ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.



ARITHMETISCHE BÄUME

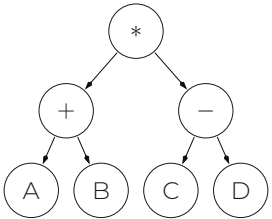
- Binärbäume helfen uns, arithmetische Operationen darzustellen.



- Die Traversierungsverfahren entsprechen Notationsschemata:

ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.

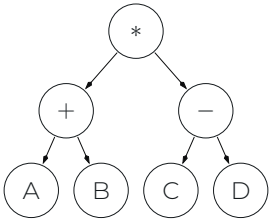


- Die Traversierungsverfahren entsprechen Notationsschemata:

  In-Order: repräsentiert die geläufige Infix-Notation: $(A + B) * (C - D)$.
  (Die Klammern entsprechen der Ausführreihenfolge und sind kein Teil.)

ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.



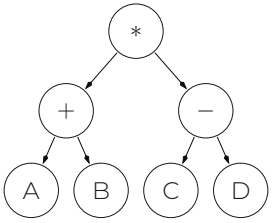
- Die Traversierungsverfahren entsprechen Notationsschemata:

  In-Order: repräsentiert die geläufige Infix-Notation: $(A + B) * (C - D)$. (Die Klammern entsprechen der Ausführreihenfolge und sind kein Teil.)

  Pre-Order: entspricht der Präfix-Notation: $* + AB - CD$.

ARITHMETISCHE BÄUME

- Binärbäume helfen uns, arithmetische Operationen darzustellen.



- Die Traversierungsverfahren entsprechen Notationsschemata:

  In-Order: repräsentiert die geläufige Infix-Notation: $(A + B) * (C - D)$. (Die Klammern entsprechen der Ausführreihenfolge und sind kein Teil.)

  Pre-Order: entspricht der Präfix-Notation: $* + AB - CD$.

  Post-Order: entspricht der Postifx-Notation: $AB + CD - *$.

GRAPHEN

GRAPHEN

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

GRAPHEN

- 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 $E \subseteq \binom{V}{2}$.

GRAPHEN

- 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.

GRAPHEN

- 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 $E \subseteq V \times V$.

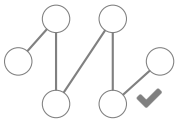
BEGRIFFE IN EINEM GRAPHEN

BEGRIFFE IN EINEM GRAPHEN

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

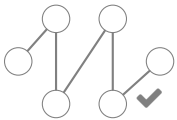
BEGRIFFE IN EINEM GRAPHEN

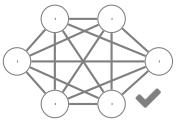
ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



BEGRIFFE IN EINEM GRAPHEN

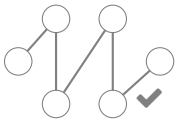
ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.

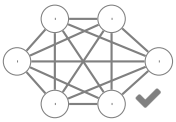


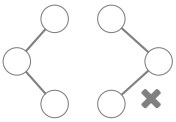


BEGRIFFE IN EINEM GRAPHEN

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

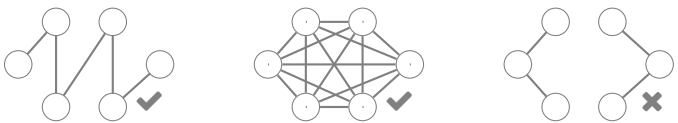






BEGRIFFE IN EINEM GRAPHEN

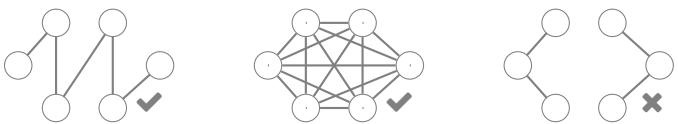
ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind.

BEGRIFFE IN EINEM GRAPHEN

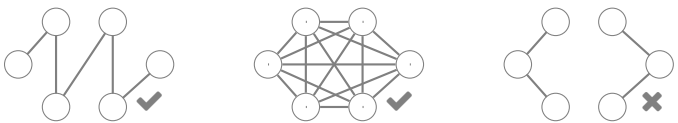
Zusammenhang: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



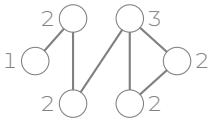
Grad: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.

BEGRIFFE IN EINEM GRAPHEN

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

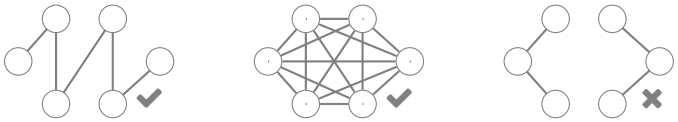


GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.



BEGRIFFE IN EINEM GRAPHEN

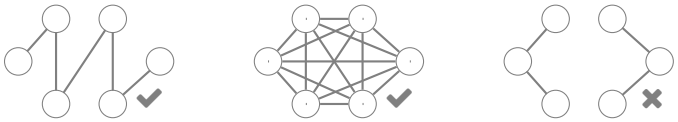
ZUSAMMENHANG: Wenn jeder Knoten von jedem anderen aus (über die Kanten) erreicht werden kann.



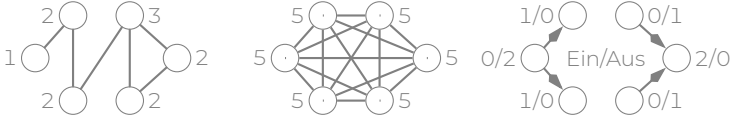
GRAD: Die Anzahl an Knoten, die mit einem Knoten verbunden sind. Ist der Graph gerichtet, so wird zwischen Eingangs- und Ausgangsgrad unterschieden.

BEGRIFFE IN EINEM GRAPHEN

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



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



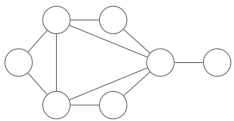
BEGRIFFE IN EINEM GRAPHEN, II

BEGRIFFE IN EINEM GRAPHEN, II

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

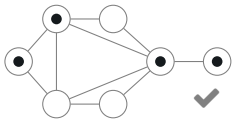
BEGRIFFE IN EINEM GRAPHEN, II

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



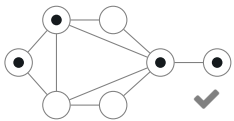
BEGRIFFE IN EINEM GRAPHEN, II

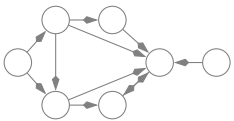
Weg: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



BEGRIFFE IN EINEM GRAPHEN, II

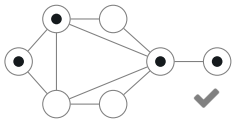
Weg: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.

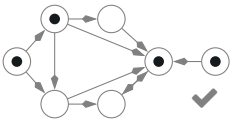




BEGRIFFE IN EINEM GRAPHEN, II

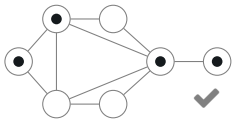
Weg: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.

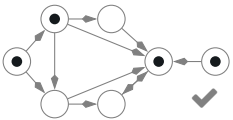


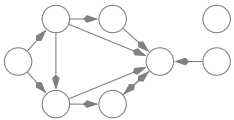


BEGRIFFE IN EINEM GRAPHEN, II

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

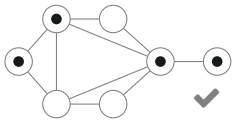


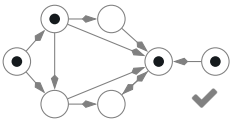


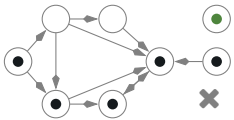


BEGRIFFE IN EINEM GRAPHEN, II

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

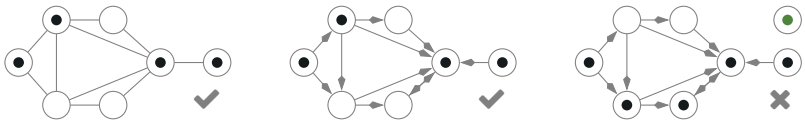






BEGRIFFE IN EINEM GRAPHEN, II

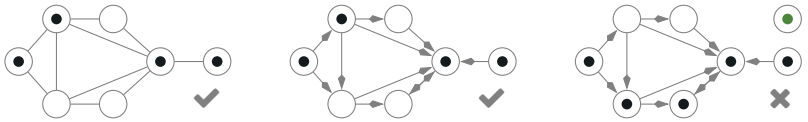
Weg: Eine endliche Folge an Knoten, die durch Kanten (egal welcher Richtung), verbunden sind.



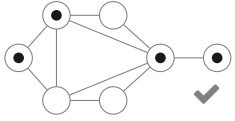
Pfad: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.

BEGRIFFE IN EINEM GRAPHEN, II

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

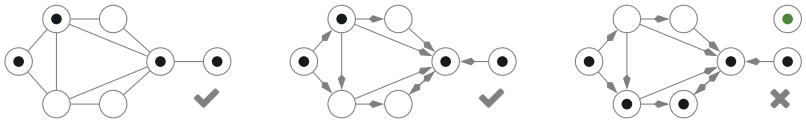


Pfad: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.

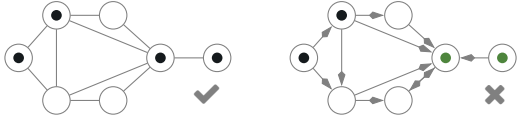


BEGRIFFE IN EINEM GRAPHEN, II

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

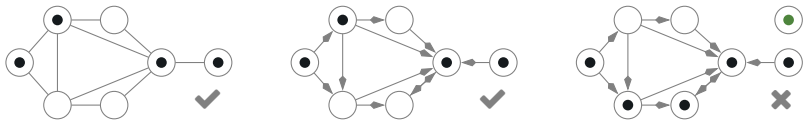


Pfad: Ein Weg, der die Kantenrichtung beachtet oder (je nach Definition) keinen Knoten doppelt enthält.

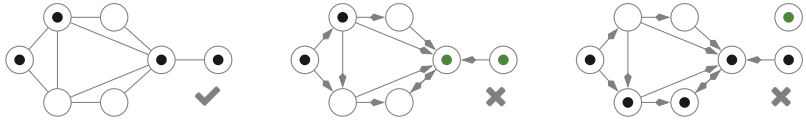


BEGRIFFE IN EINEM GRAPHEN, II

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



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



VARIANTEN VON GRAPHEN

VARIANTEN VON GRAPHEN

- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \to M$ einen Wert aus einer Menge M zu. (Beispielsweise $M = \mathbb{N}$)

VARIANTEN VON GRAPHEN

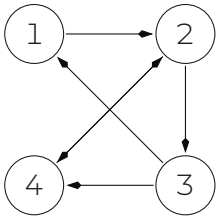
- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \to M$ einen Wert aus einer Menge M zu. (Beispielsweise $M = \mathbb{N}$)
- Wir können einen Graphen durch eine *Adjazenzmatrix* beschreiben.

VARIANTEN VON GRAPHEN

- Ein gewichteter Graph weist den Kanten durch eine Abbildung $w : E \to M$ 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).

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

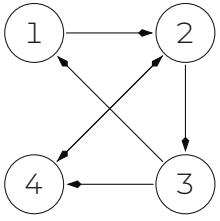
- Adjazenzlisten sind eine andere Variante Graphen zu repräsentieren.

ADJAZENZLISTEN

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

ADJAZENZLISTEN

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



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

ALGORITHMEN AUF GRAPHEN

ALGORITHMEN AUF GRAPHEN

- Eine gängiges Problem ist es herauszufinden ob ein Graph Zyklen enthält und/oder ob er zusammenhängend ist.

ALGORITHMEN AUF GRAPHEN

- Eine gängiges Problem ist es herauszufinden ob ein Graph Zyklen enthält und/oder ob er zusammenhängend ist.
- Wegfindungsproblem wie das kürzeste Wege Probleme (Routenfindung, …), Traveling Salesman Problem.

ALGORITHMEN AUF GRAPHEN

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

ALGORITHMEN AUF GRAPHEN

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

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

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

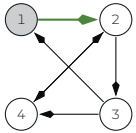
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

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

GRAPHEN DURCHSUCHEN: TIEFENSUCHE

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

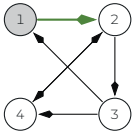
1.



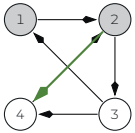
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- 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.

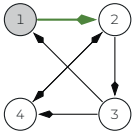


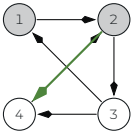
2.

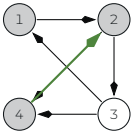


GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- 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.

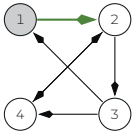
1. 

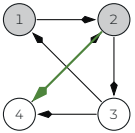
2. 

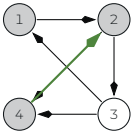
3. 

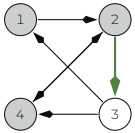
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

- 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. 

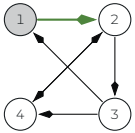
2. 

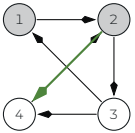
3. 

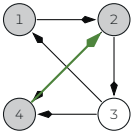
4. 

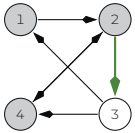
GRAPHEN DURCHSUCHEN: TIEFENSUCHE

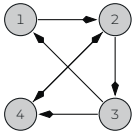
- In der Tiefensuche besucht man von einem Startknoten aus die anderen benachbarten Knoten und markiert diese.
- Dies wird solange vollzogen bis man auf einen bereits beobachteten Knoten trifft.
- In diesem Fall wird eine andere Kante besucht. Existiert keine mehr, so geht man zum letzten Knoten zuvor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

1. 

2. 

3. 

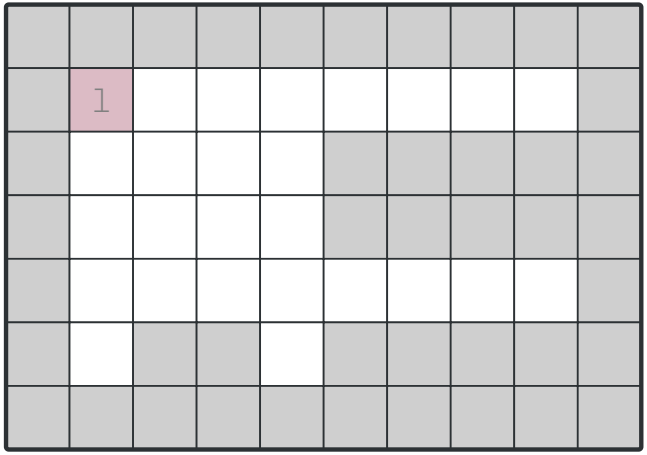
4. 

5. 

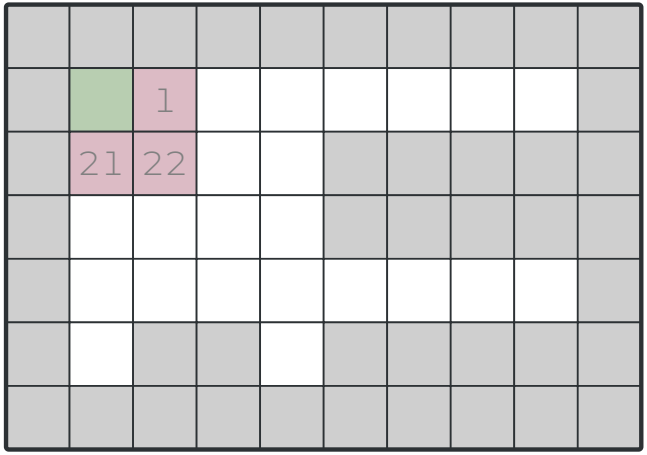
EIN BEISPIEL FÜR DIE TIEFENSUCHE

EIN BEISPIEL FÜR DIE TIEFENSUCHE

EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



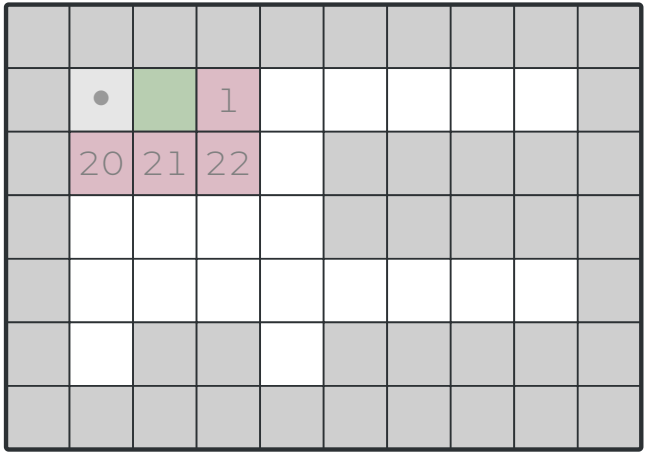
TRAVERSIERUNGSVARIANTEN

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



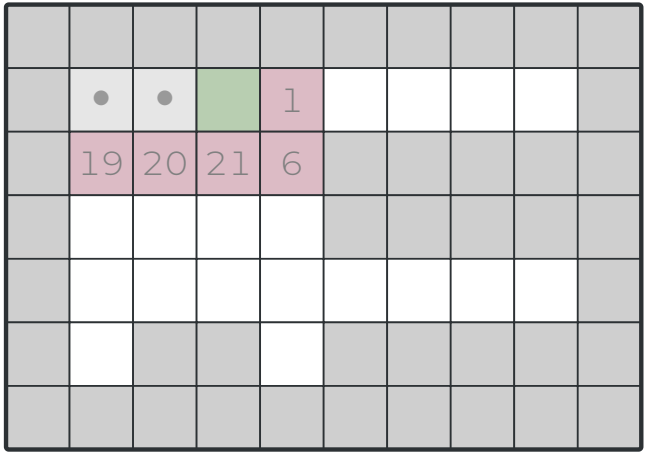
TRAVERSIERUNGSVARIANTEN

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



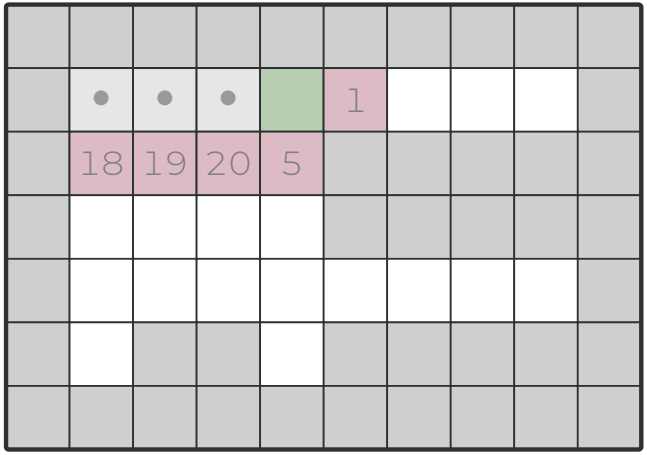
TRAVERSIERUNGSVARIANTEN

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



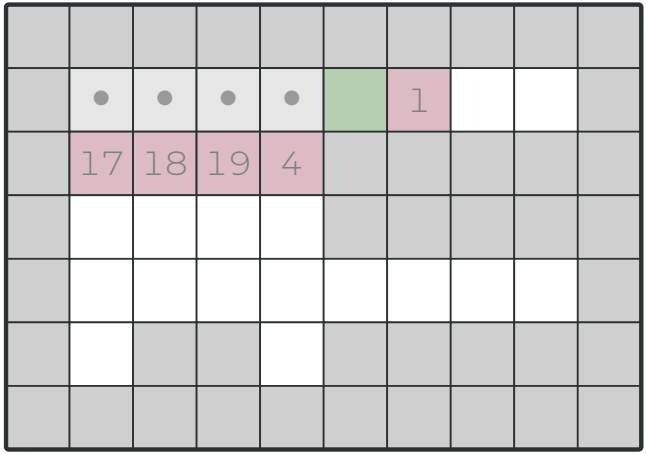
TRAVERSIERUNGSVARIANTEN

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

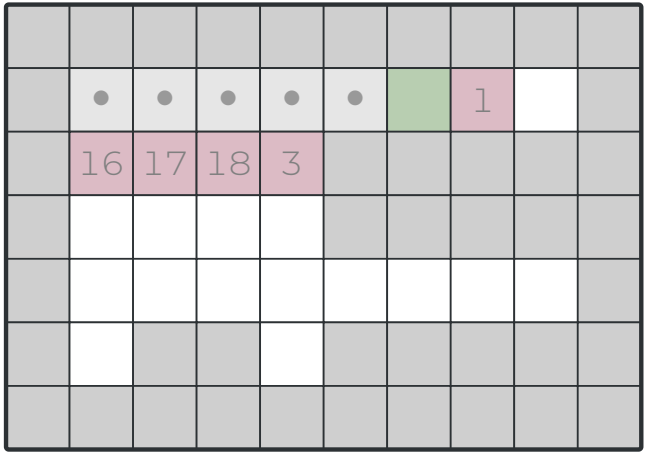
Florian Sihler

13. Jul 2022
SP, Universität Ulm

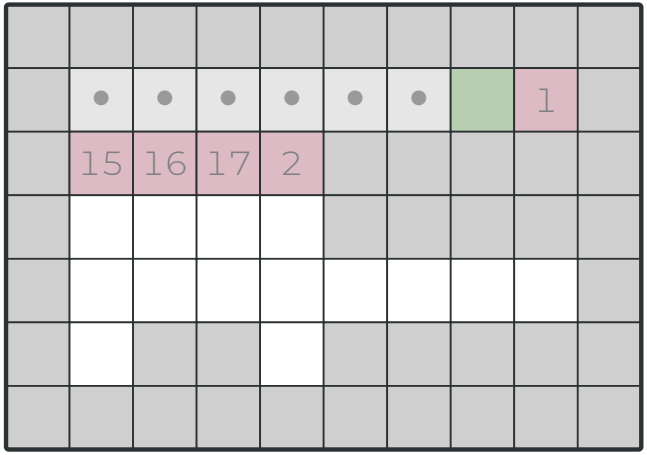
EIN BEISPIEL FÜR DIE TIEFENSUCHE



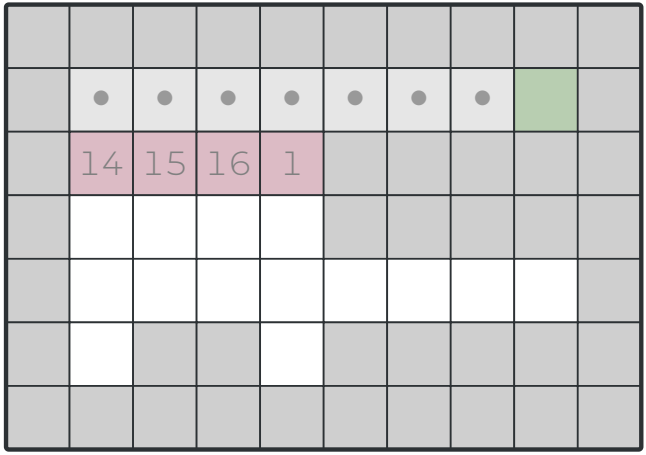
EIN BEISPIEL FÜR DIE TIEFENSUCHE



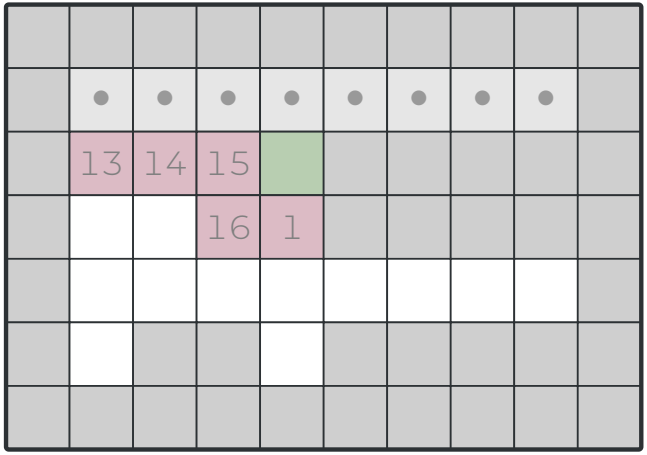
EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



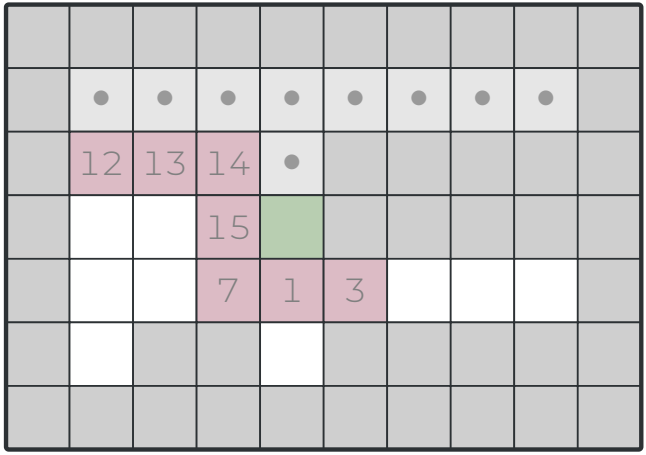
TRAVERSIERUNGSVARIANTEN

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



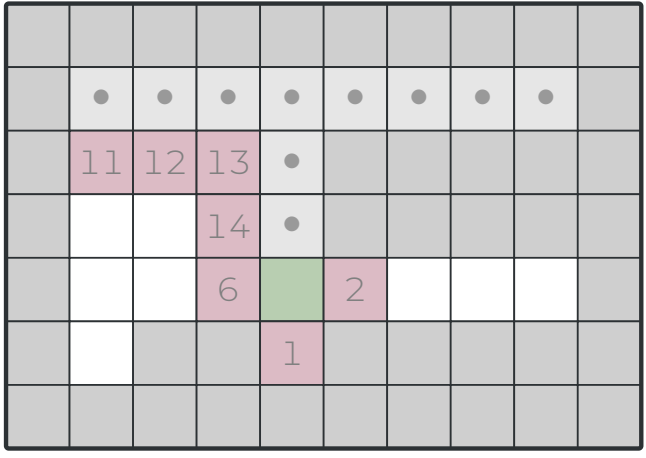
TRAVERSIERUNGSVARIANTEN

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

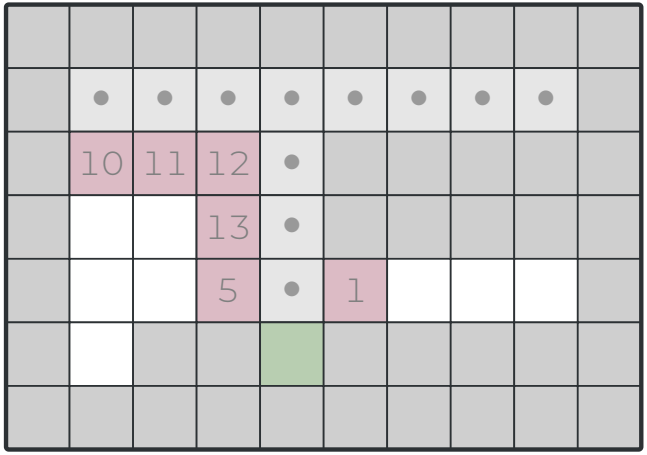
Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



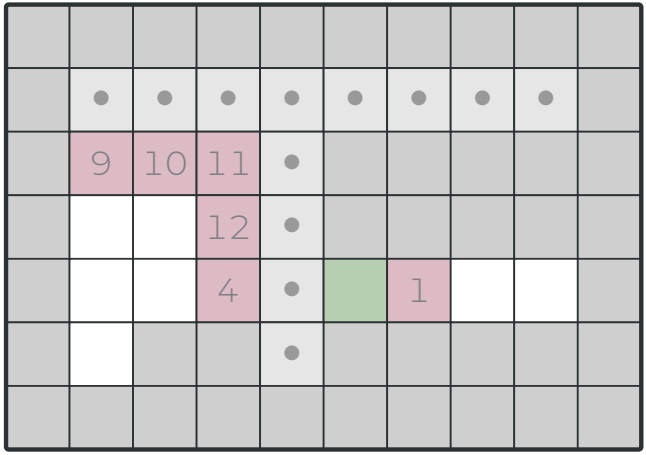
TRAVERSIERUNGSVARIANTEN

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



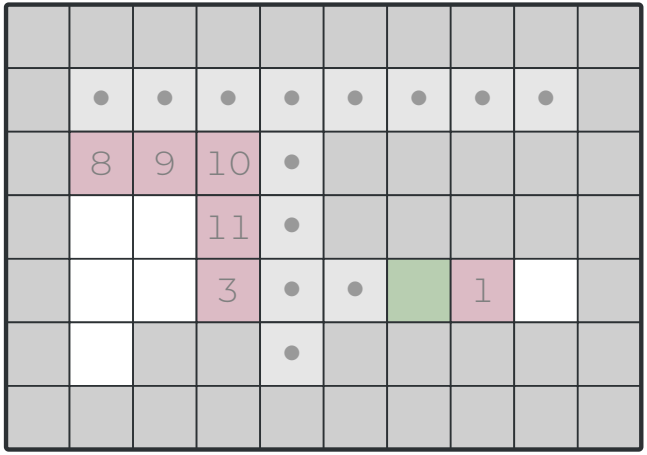
TRAVERSIERUNGSVARIANTEN

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



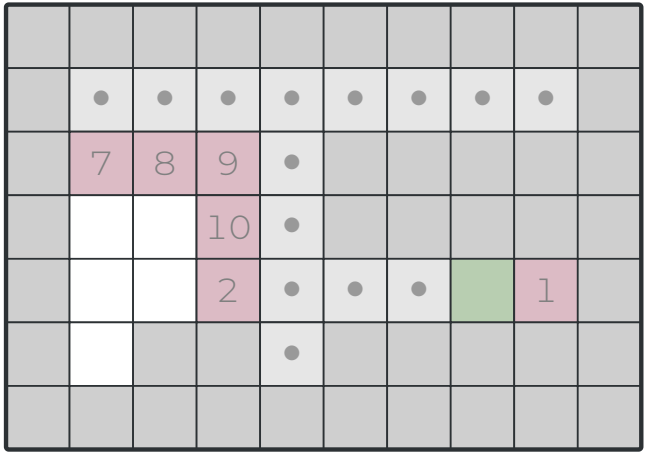
TRAVERSIERUNGSVARIANTEN

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



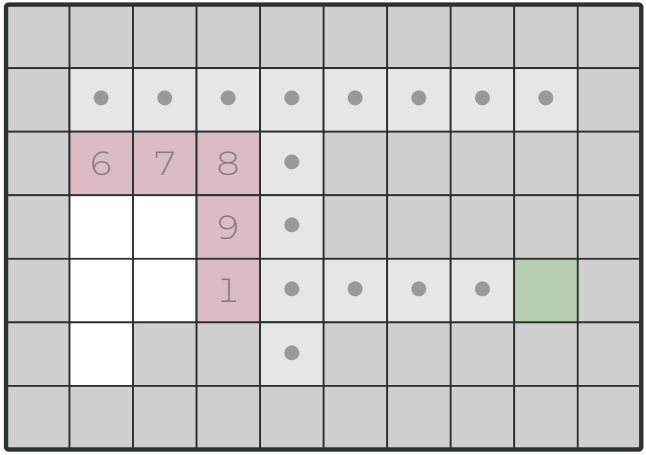
TRAVERSIERUNGSVARIANTEN

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



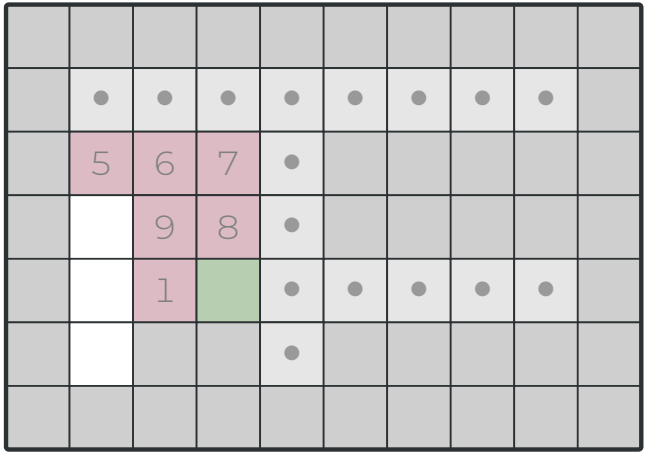
TRAVERSIERUNGSVARIANTEN

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



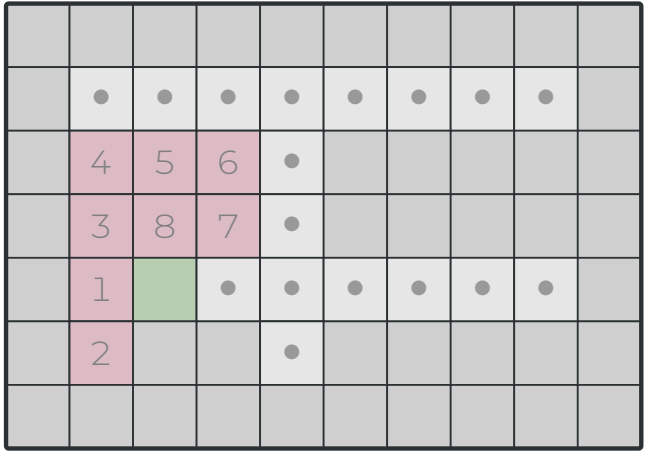
TRAVERSIERUNGSVARIANTEN

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

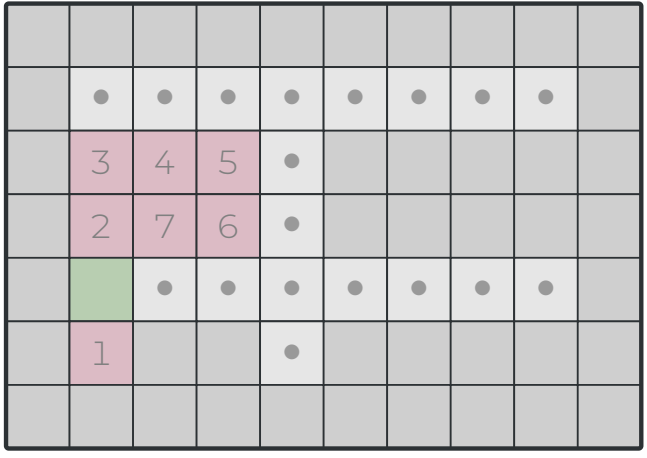
Florian Sihler

13. Jul 2022
SP, Universität Ulm

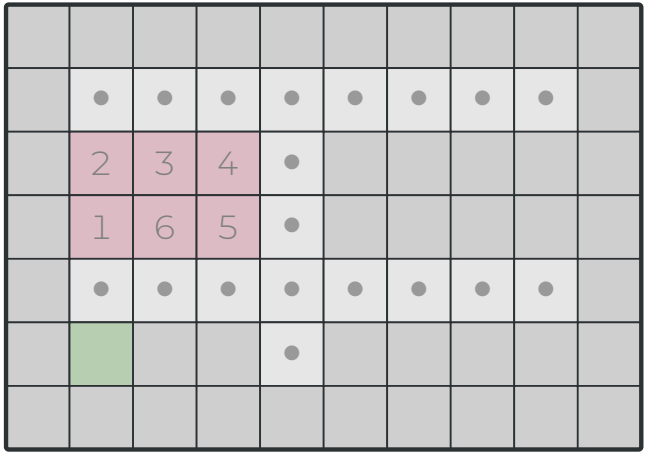
EIN BEISPIEL FÜR DIE TIEFENSUCHE



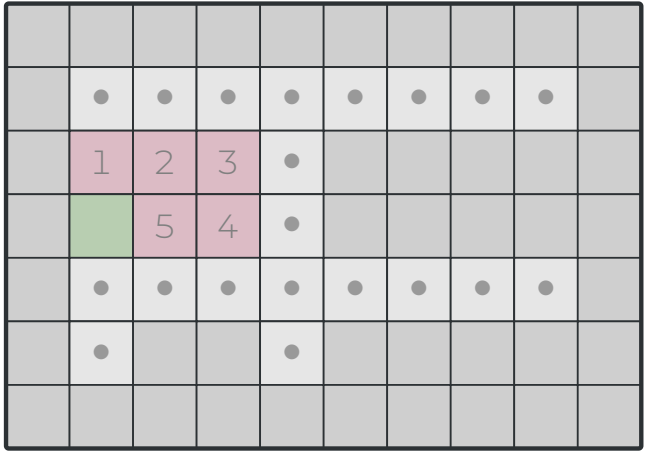
EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



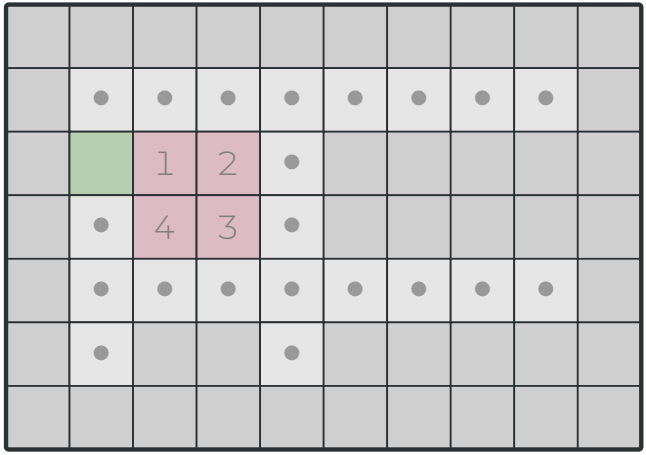
TRAVERSIERUNGSVARIANTEN

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



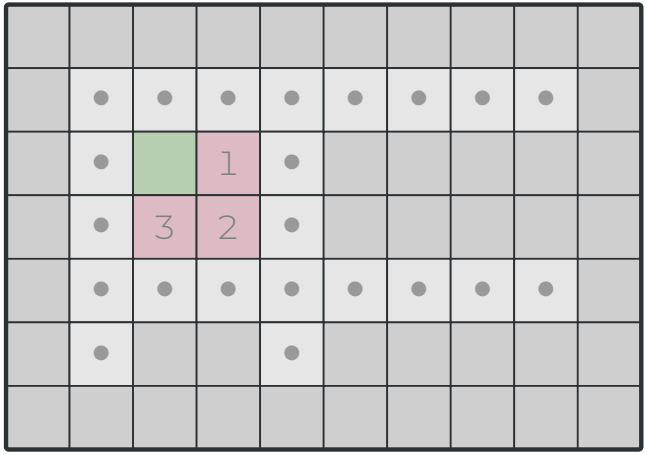
TRAVERSIERUNGSVARIANTEN

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

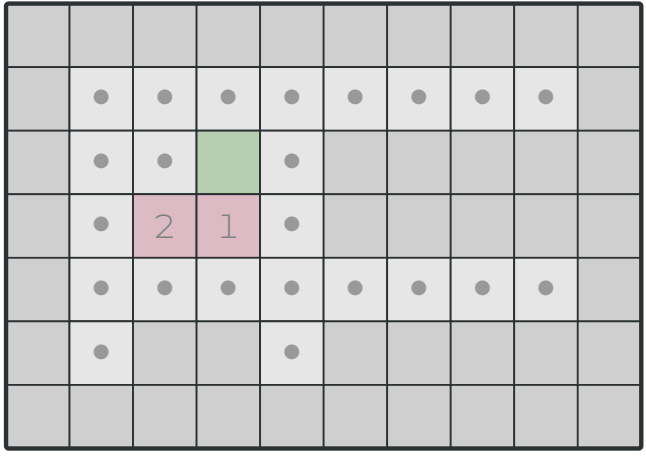
Florian Sihler

13. Jul 2022
SP, Universität Ulm

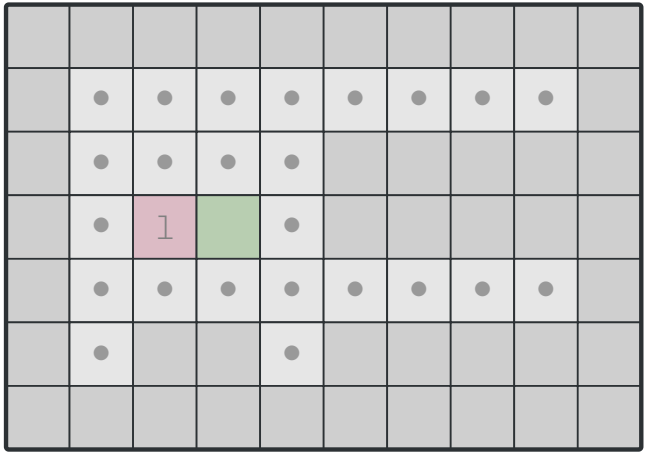
EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



EIN BEISPIEL FÜR DIE TIEFENSUCHE



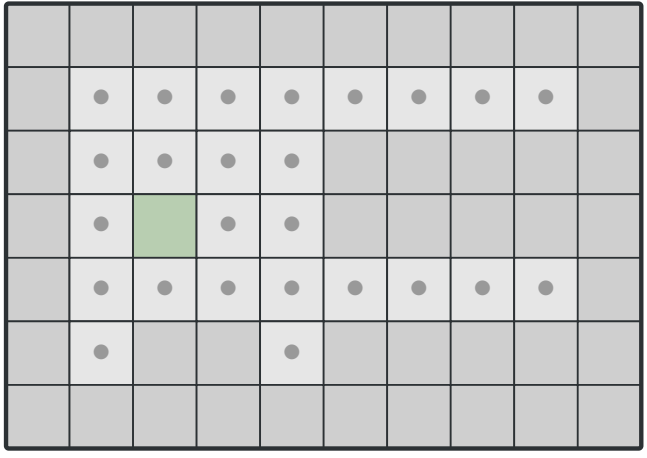
TRAVERSIERUNGSVARIANTEN

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



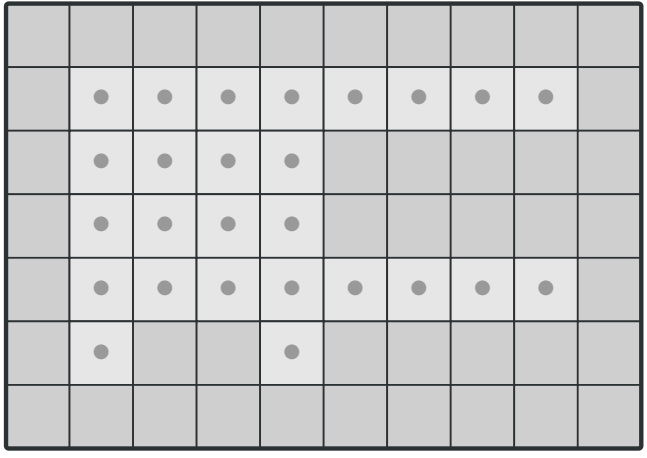
TRAVERSIERUNGSVARIANTEN

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



TRAVERSIERUNGSVARIANTEN

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

Florian Sihler

13. Jul 2022
SP, Universität Ulm

GRAPHEN DURCHSUCHEN: BREITENSUCHE

GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.

GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.

GRAPHEN DURCHSUCHEN: BREITENSUCHE

- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil:

GRAPHEN DURCHSUCHEN: BREITENSUCHE

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

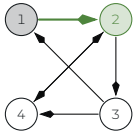
GRAPHEN DURCHSUCHEN: BREITENSUCHE

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

GRAPHEN DURCHSUCHEN: BREITENSUCHE

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

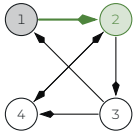
1.



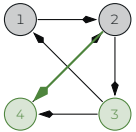
GRAPHEN DURCHSUCHEN: BREITENSUCHE

- 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.

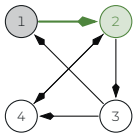


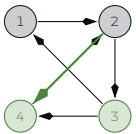
2.

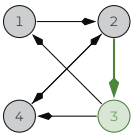


GRAPHEN DURCHSUCHEN: BREITENSUCHE

- 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.

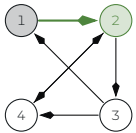
1. 

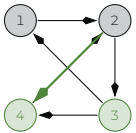
2. 

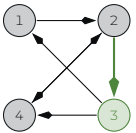
3. 

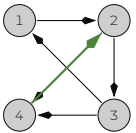
GRAPHEN DURCHSUCHEN: BREITENSUCHE

- 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. 

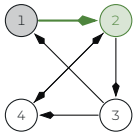
2. 

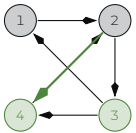
3. 

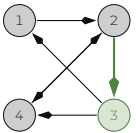
4. 

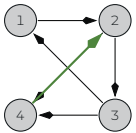
GRAPHEN DURCHSUCHEN: BREITENSUCHE

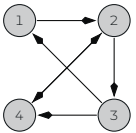
- In der Breitensuche besucht man von einem Startknoten aus erst alle benachbarten Knoten.
- In diesem Zuge merkt man sich die Nachbarn dieser Knoten (mit einer Queue) und besucht anschließend diese.
- Der Vorteil: Wenn alle Kanten gleich gewichtet sind, lässt sich so direkt der kürzeste Weg identifizieren.

1. 

2. 

3. 

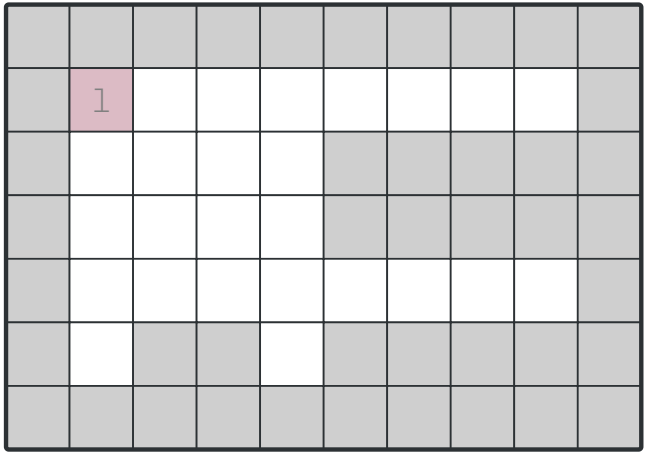
4. 

5. 

EIN BEISPIEL FÜR DIE BREITENSUCHE

EIN BEISPIEL FÜR DIE BREITENSUCHE

EIN BEISPIEL FÜR DIE BREITENSUCHE



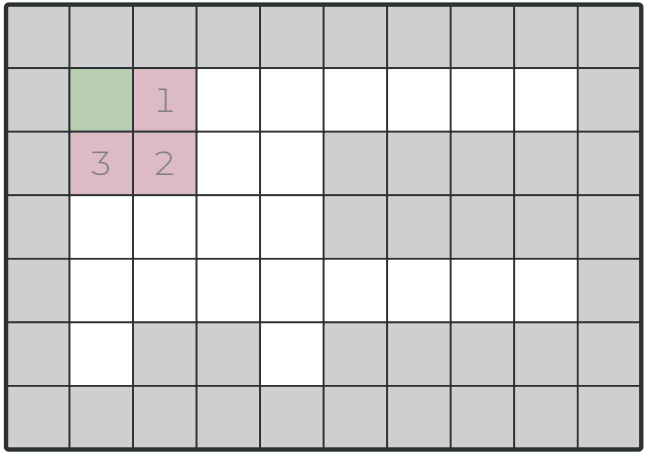
TRAVERSIERUNGSVARIANTEN

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



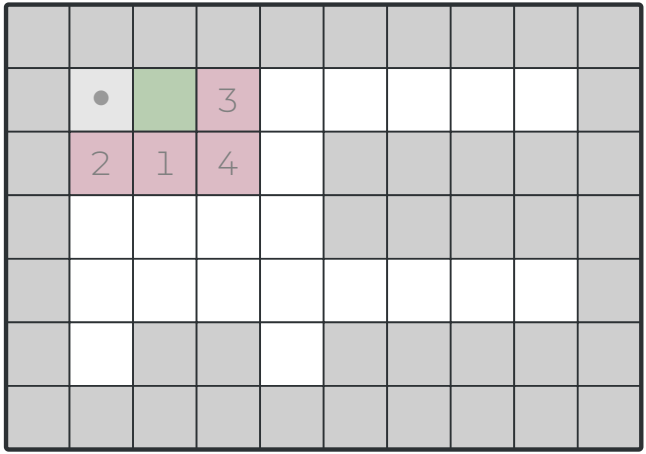
TRAVERSIERUNGSVARIANTEN

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



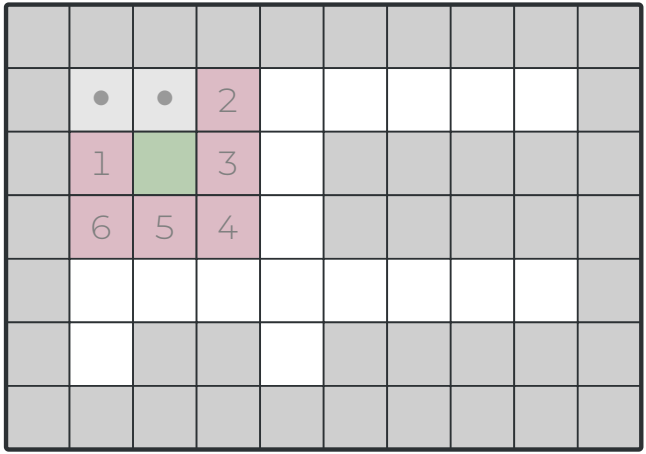
TRAVERSIERUNGSVARIANTEN

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

Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



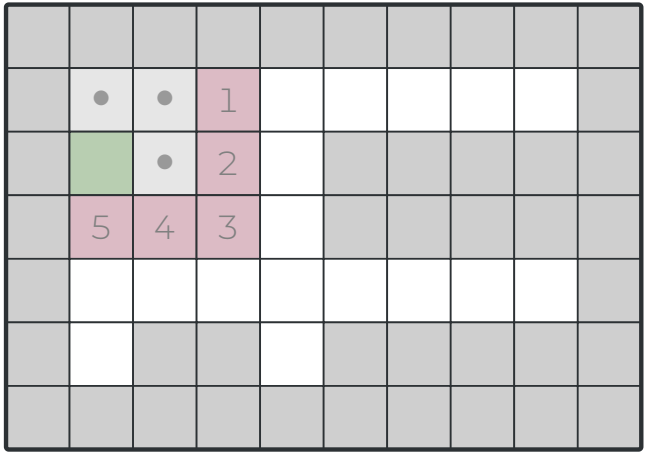
TRAVERSIERUNGSVARIANTEN

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



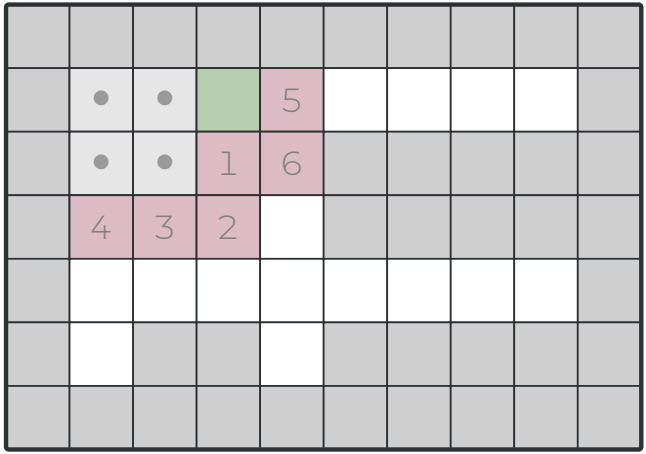
TRAVERSIERUNGSVARIANTEN

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

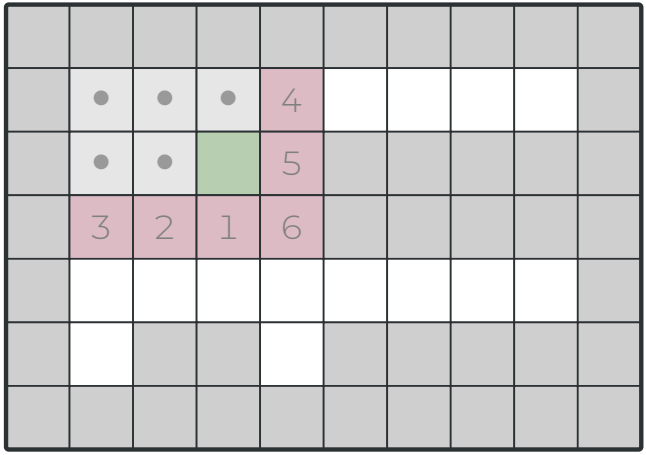
Florian Sihler

13. Jul 2022
SP, Universität Ulm

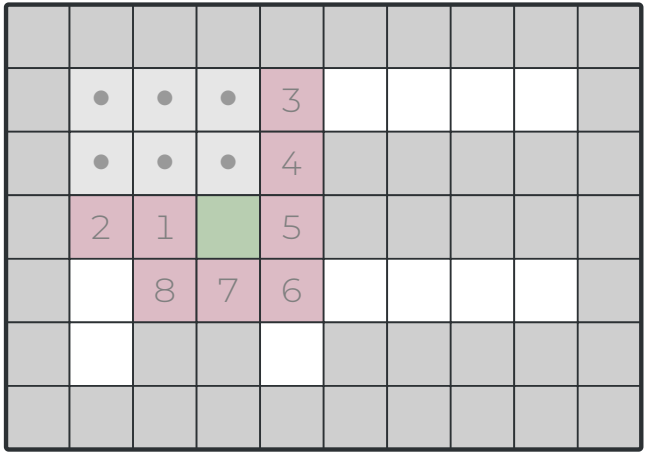
EIN BEISPIEL FÜR DIE BREITENSUCHE



EIN BEISPIEL FÜR DIE BREITENSUCHE



EIN BEISPIEL FÜR DIE BREITENSUCHE



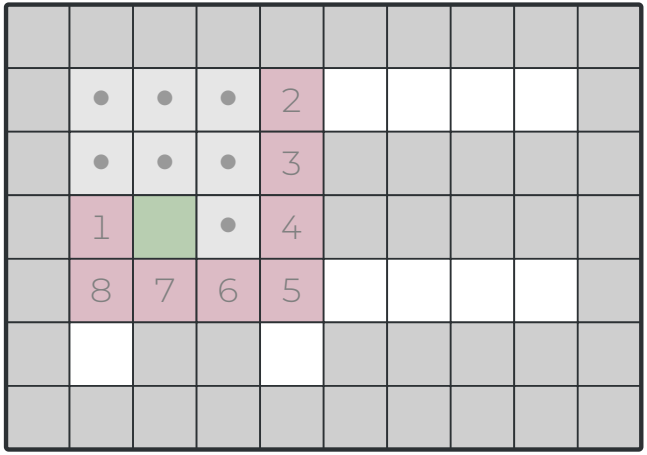
TRAVERSIERUNGSVARIANTEN

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

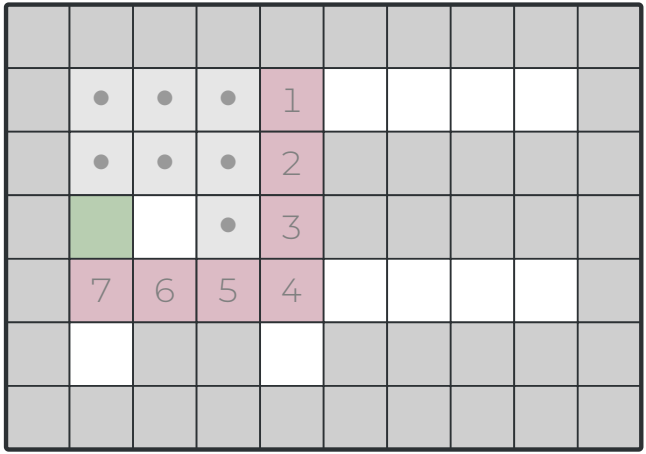
Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



EIN BEISPIEL FÜR DIE BREITENSUCHE



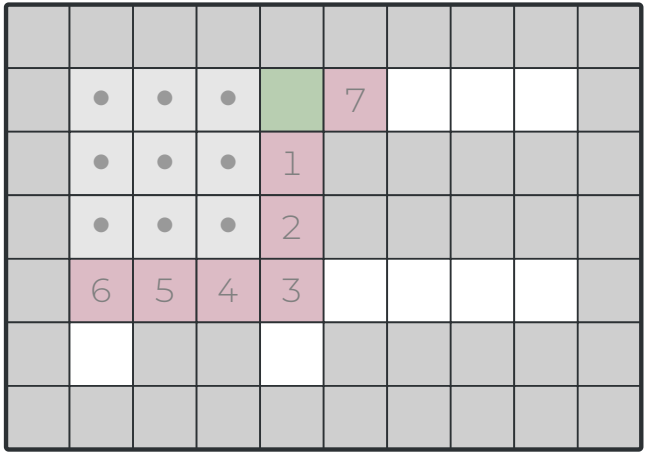
TRAVERSIERUNGSVARIANTEN

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



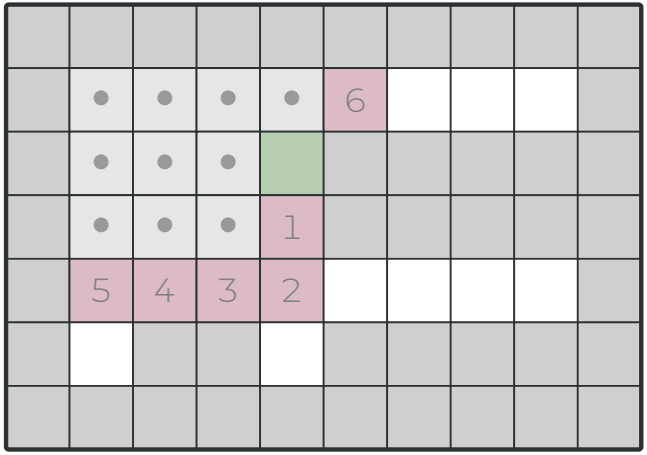
TRAVERSIERUNGSVARIANTEN

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



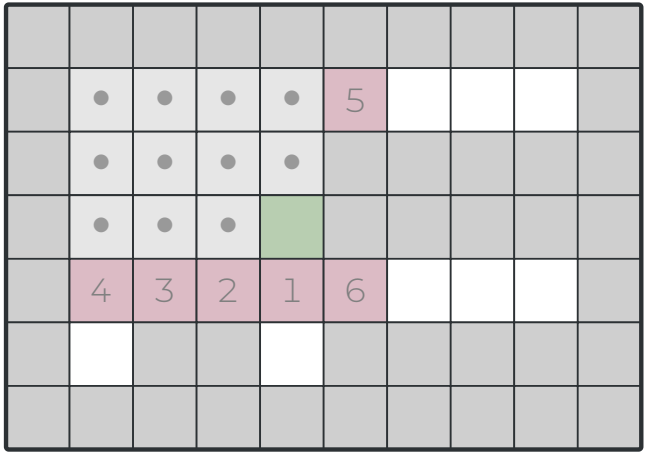
TRAVERSIERUNGSVARIANTEN

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

Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



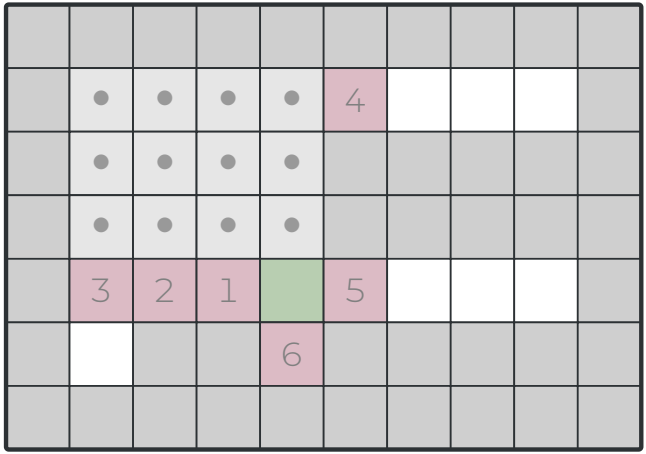
TRAVERSIERUNGSVARIANTEN

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



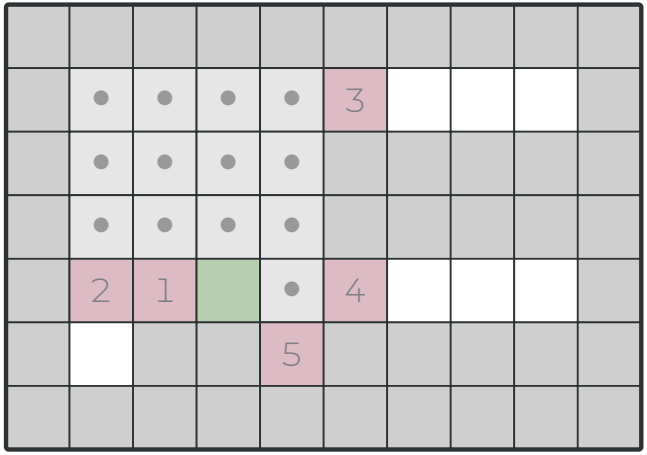
TRAVERSIERUNGSVARIANTEN

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



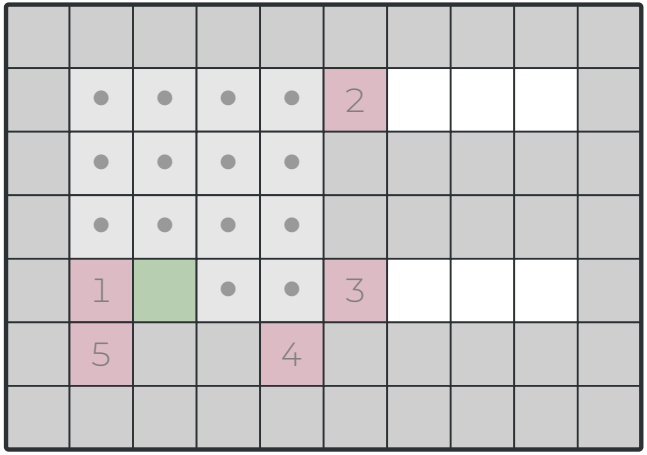
TRAVERSIERUNGSVARIANTEN

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

Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



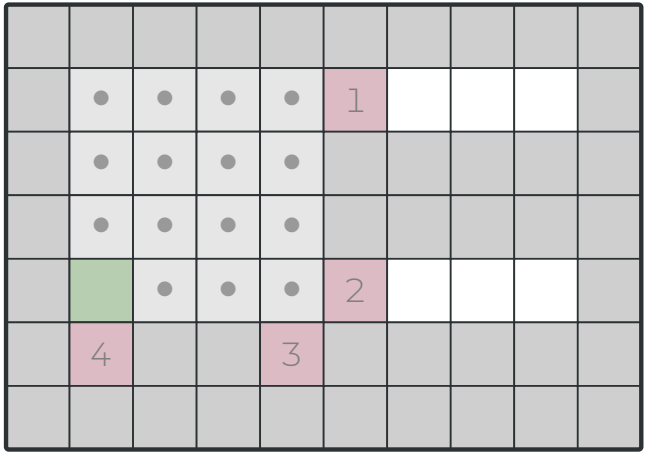
TRAVERSIERUNGSVARIANTEN

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



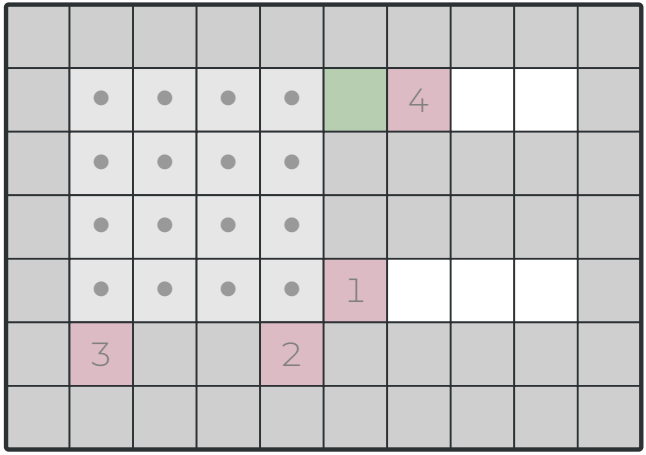
TRAVERSIERUNGSVARIANTEN

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



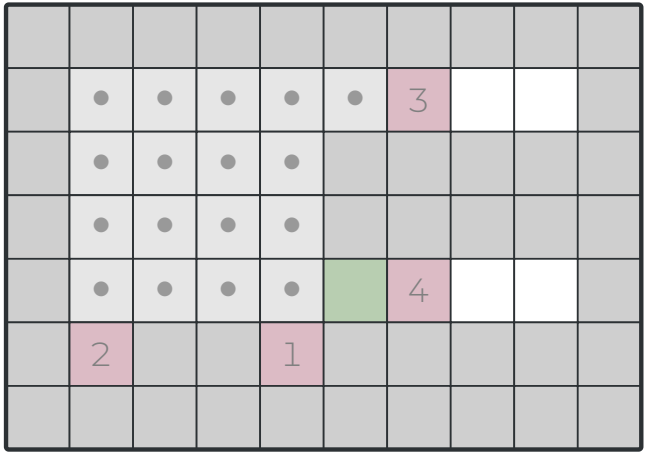
TRAVERSIERUNGSVARIANTEN

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

Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



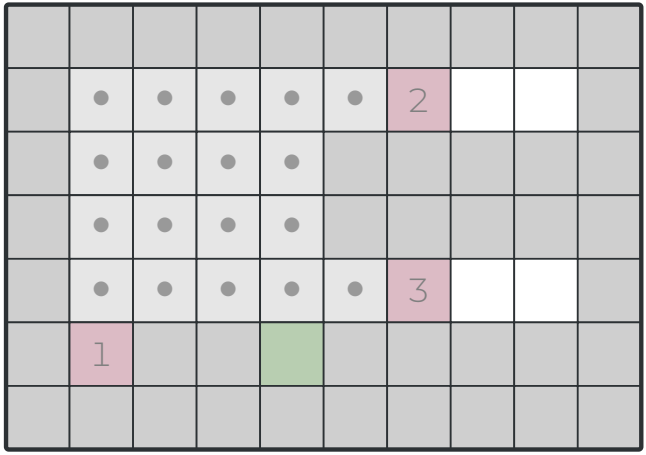
TRAVERSIERUNGSVARIANTEN

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



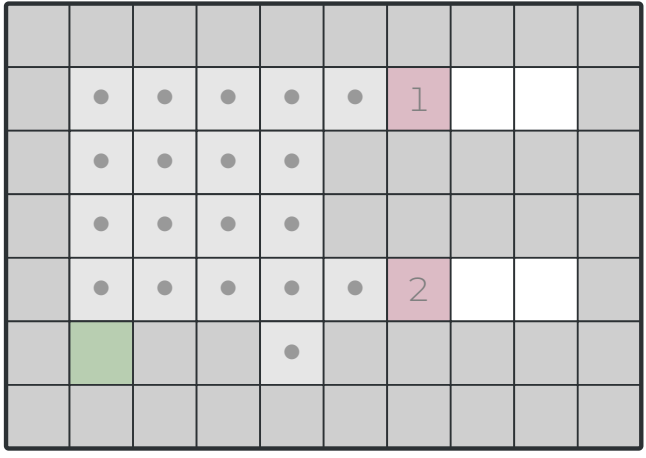
TRAVERSIERUNGSVARIANTEN

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



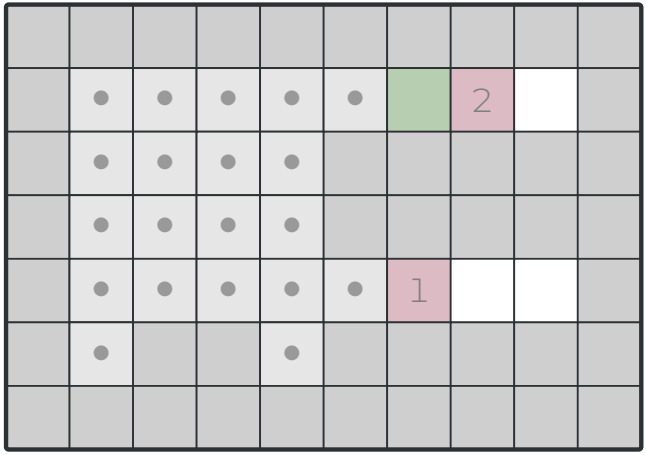
TRAVERSIERUNGSVARIANTEN

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

Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



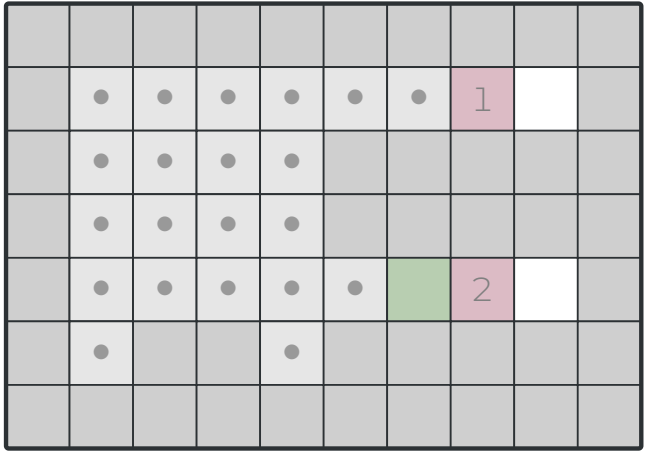
TRAVERSIERUNGSVARIANTEN

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



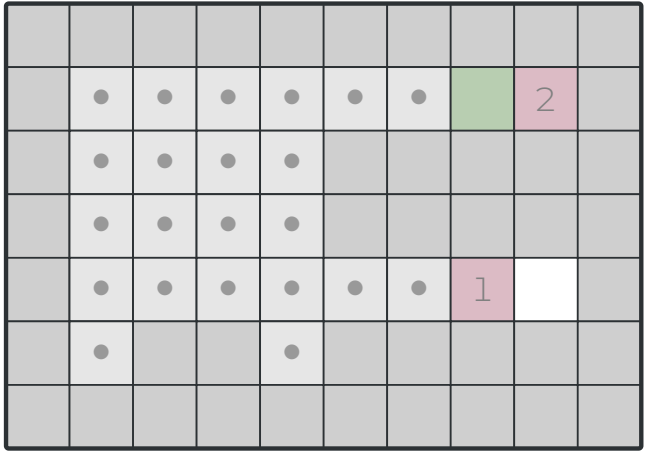
TRAVERSIERUNGSVARIANTEN

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

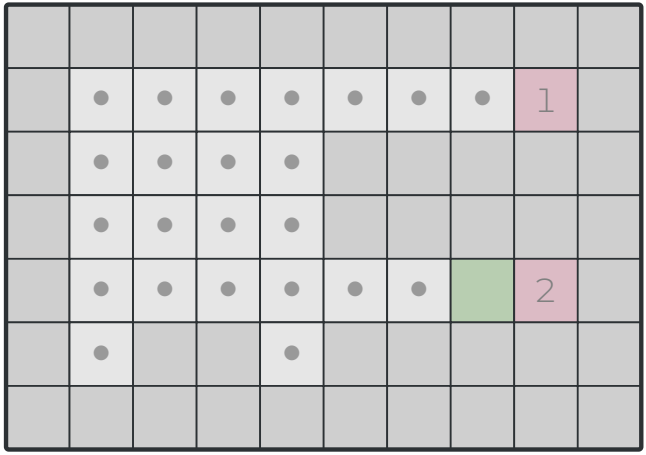
Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



EIN BEISPIEL FÜR DIE BREITENSUCHE



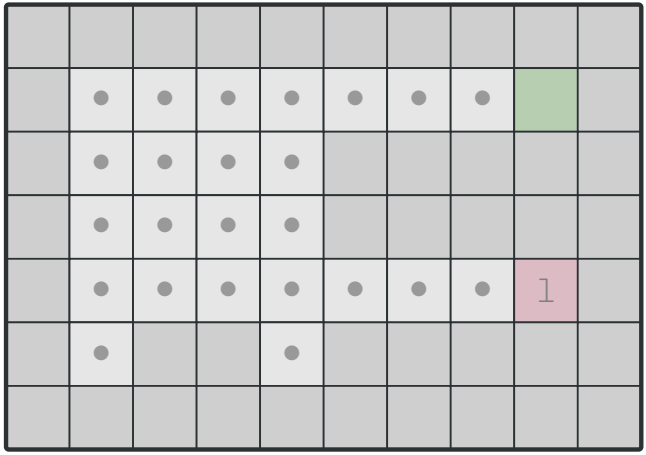
TRAVERSIERUNGSVARIANTEN

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

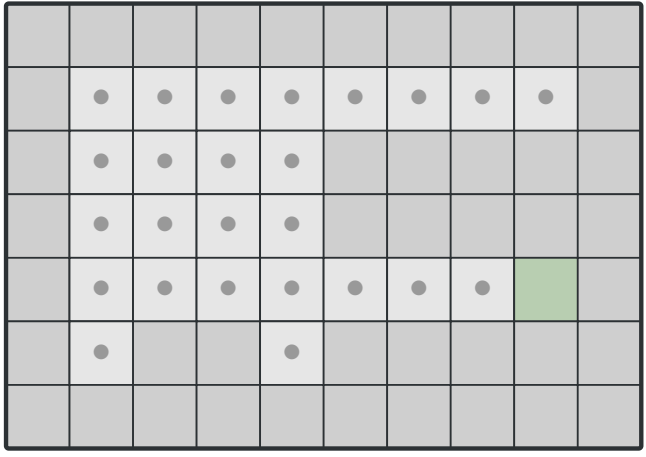
Florian Sihler

13. Jul 2022
SP, Universität Ulm

EIN BEISPIEL FÜR DIE BREITENSUCHE



EIN BEISPIEL FÜR DIE BREITENSUCHE



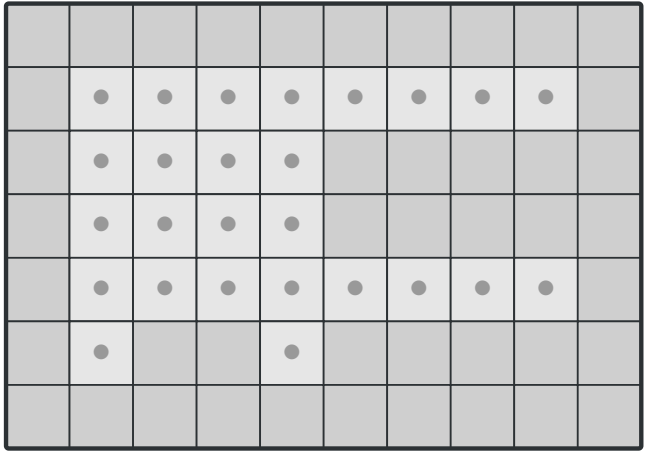
TRAVERSIERUNGSVARIANTEN

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

Java-Advanced

DAS KONZEPT DER VERERBUNG

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
  - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
  - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
  - als Erweiterung: die Methoden werden modifiziert.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
  - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
  - als Erweiterung: die Methoden werden modifiziert.
- Häufig wird das Konzept der *Erweiterung* verfolgt.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
  - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
  - als Erweiterung: die Methoden werden modifiziert.
- Häufig wird das Konzept der *Erweiterung* verfolgt.
- Zwischen vererbenden Klassen gilt eine „is-a“ Relation.

DAS KONZEPT DER VERERBUNG

- Eine erbende Klasse („Subklasse“) übernimmt alle Eigenschaften und kann diese verändern.
- Die Subklasse kann hier auf zwei Weisen betrachtet werden:
  - als Spezialisierung: die Eigenschaften und Operationen des Supertyps bleiben unverändert.
  - als Erweiterung: die Methoden werden modifiziert.
- Häufig wird das Konzept der *Erweiterung* verfolgt.
- Zwischen vererbenden Klassen gilt eine „is-a“ Relation. So gilt beispielsweise „Student *is-a* Mensch *is-a* Lebewesen“.

VERERBUNG IN JAVA

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **`extends`** realisiert.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **`extends`** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **`extends`** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.
- In Java kann eine Klasse nur von maximal einer anderen Erben.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **`extends`** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.
- In Java kann eine Klasse nur von maximal einer anderen Erben.
- Auf Attribute der Elternklasse, die **`private`** sind, können wir in erbenden Kindklassen nicht zugreifen.

VERERBUNG IN JAVA

- Vererbung wird in Java durch den Schlüsselbegriff **`extends`** realisiert.
- Java verwendet Vererbung zur *Erweiterung*.
- In Java kann eine Klasse nur von maximal einer anderen Erben.
- Auf Attribute der Elternklasse, die **`private`** sind, können wir in erbenden Kindklassen nicht zugreifen.
- Der Konstruktor sowie Methoden der Superklasse können (soweit sichtbar) durch `super` erreicht werden.

EIN BEISPIEL IN JAVA

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

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

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

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.

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

```
class Student extends Mensch {
   long matrikelnummer;

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

EIN BEISPIEL IN JAVA

- Die Attribute sind lediglich exemplarisch und nicht wertend gemeint.

```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()

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie.

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

  ```
  Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
  ```

  Die Klasse `Student` *erbt* von `Mensch`.

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

```
Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
```

Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

  ```
  Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
  ```

  Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

- Ob `herbert` wirklich ein Student ist,

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

  ```
  Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
  ```

  Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

- Ob `herbert` wirklich ein Student ist, kann man mit **`instanceof`** prüfen:

INSTANCEOF UND GETCLASS()

- Das Prinzip erlaubt uns Polymorphie. So ist folgender Code valide:

  ```
  Mensch herbert = new Student(19, 184.12f, "herbert", 12345);
  ```

  Die Klasse `Student` *erbt* von `Mensch`. Wir können nicht auf zusätzliche Attribute und Funktionen von `Student` zugreifen.

- Ob `herbert` wirklich ein Student ist, kann man mit **`instanceof`** prüfen:

  ```
  herbert instanceof Student // → true
  herbert instanceof Mensch // → true
  herbert instanceof Lebewesen // → true
  herbert instanceof Dozent // → false
  ```

  Dabei liefert **`null instanceof X`** immer `false`, für beliebige Klassen `X`.

KOMMENTARE

KOMMENTARE

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.

KOMMENTARE

- Das Prinzip der Vererbung ist integral für die Objektorientierung und unglaublich mächtig.
- Methoden der Superklasse lassen sich überladen (gleicher Name, andere Signatur),

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)

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,

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.

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

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {
    public int foo(int i) { return i + 1; }
}
```

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {
    public int foo(int i) { return i + 1; }
}
public class B extends A {
    public int foo(int i) { return i * i; }
}
```

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {
    public int foo(int i) { return i + 1; }
}
public class B extends A {
    public int foo(int i) { return i * i; }
}
```

- Hier überschreibt `B::foo(int)` die Methode `A::foo(int)`.

ÜBERSCHREIBEN

- Analog zum *Überladen* (gleicher Name, unterschiedliche Signatur) gibt es das „Überschreiben“.
- Eine Methode in einer erbenden Klasse *überschreibt* eine (sichtbare) Methode in der Superklasse mit der gleichen Signatur.

```
public class A {
    public int foo(int i) { return i + 1; }
}
public class B extends A {
    public int foo(int i) { return i * i; }
}
```

- Hier überschreibt `B::foo(int)` die Methode `A::foo(int)`.
- `new B().foo(3)` produziert damit `9`. In `B` steht `A::foo(int)` über `super` zur Verfügung.

ABSTRAKTE KLASSEN

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, …

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, … die diese Operationen implementieren.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, … die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind,

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, … die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **`abstract`** bezeichnen.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, … die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **`abstract`** bezeichnen.
- Von **`abstract`** Klassen darf kein Objekt erstellt werden.
  Von ihnen kann aber geerbt werden.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, … die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **`abstract`** bezeichnen.
- Von **`abstract`** Klassen darf kein Objekt erstellt werden.
  Von ihnen kann aber geerbt werden.
- Eine abstrakte Klasse kann abstrakte Methoden (ohne Rumpf!) definieren.

ABSTRAKTE KLASSEN

- Vererbung ermöglicht Polymorphie (erst richtig).
- So lässt sich eine Klasse `List` schaffen (in Java: `AbstractList`, bereits vorhanden), welche die Zugriffe definiert und entsprechend Klassen `ArrayList`, `LinkedList`, … die diese Operationen implementieren.
- Sind diese Klassen (wie `List`) wirklich nur Blaupausen/Vorlagen sind, können wir sie als **`abstract`** bezeichnen.
- Von **`abstract`** Klassen darf kein Objekt erstellt werden.
  Von ihnen kann aber geerbt werden.
- Eine abstrakte Klasse kann abstrakte Methoden (ohne Rumpf!) definieren. Diese *müssen* von einer erbenden, nicht-abstrakten Klasse implementiert werden.

ABSTRAKTE KLASSEN, EIN BEISPIEL

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {




}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {

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









}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {

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

    abstract void add(Element e);




}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {

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

    abstract void add(Element e);

    int size() {
        // ...
    }

}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
abstract class AbstractList {

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

    abstract void add(Element e);

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

ABSTRAKTE KLASSEN, EIN BEISPIEL

ABSTRAKTE KLASSEN, EIN BEISPIEL

```
class ArrayList extends AbstractList {









}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

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









}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

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

    public ArrayList() {


    }









}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

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

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









}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

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

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

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

}
```

ABSTRAKTE KLASSEN, EIN BEISPIEL

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

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

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

STATISCHE BINDUNG

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  STATISCH: Nehmen wir an, wir haben zwei Klassen `A` und `B`, wobei `B` von `A` erbt.

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  STATISCH: Nehmen wir an, wir haben zwei Klassen `A` und `B`, wobei `B` von `A` erbt. Beide deklarieren die Variable `i`, in `A` wird sie auf 32, in `B` auf 42 gesetzt.

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  STATISCH: Nehmen wir an, wir haben zwei Klassen `A` und `B`, wobei `B` von `A` erbt. Beide deklarieren die Variable `i`, in `A` wird sie auf 32, in `B` auf 42 gesetzt. Es ergibt sich:

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  STATISCH: Nehmen wir an, wir haben zwei Klassen `A` und `B`, wobei `B` von `A` erbt. Beide deklarieren die Variable `i`, in `A` wird sie auf 32, in `B` auf 42 gesetzt. Es ergibt sich:

```
A a = new A();
B b = new B();
A c = new B();
```

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  STATISCH: Nehmen wir an, wir haben zwei Klassen `A` und `B`, wobei `B` von `A` erbt. Beide deklarieren die Variable `i`, in `A` wird sie auf 32, in `B` auf 42 gesetzt. Es ergibt sich:

```
A a = new A();
B b = new B();
A c = new B();
System.out.println(a.i); // → 32
```

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  STATISCH: Nehmen wir an, wir haben zwei Klassen `A` und `B`, wobei `B` von `A` erbt. Beide deklarieren die Variable `i`, in `A` wird sie auf 32, in `B` auf 42 gesetzt. Es ergibt sich:

```
A a = new A();
B b = new B();
A c = new B();
System.out.println(a.i); // → 32
System.out.println(b.i); // → 42
```

STATISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  STATISCH: Nehmen wir an, wir haben zwei Klassen `A` und `B`, wobei `B` von `A` erbt. Beide deklarieren die Variable `i`, in `A` wird sie auf 32, in `B` auf 42 gesetzt. Es ergibt sich:

```
A a = new A();
B b = new B();
A c = new B();
System.out.println(a.i); // → 32
System.out.println(b.i); // → 42
System.out.println(c.i); // → 32
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  DYNAMISCH: Angenommen die Klasse `Mensch` hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt.

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  DYNAMISCH: Angenommen die Klasse `Mensch` hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. `Student` überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus.

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

  DYNAMISCH: Angenommen die Klasse `Mensch` hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. `Student` überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCH: Angenommen die Klasse `Mensch` hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. `Student` überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);
Student b = new Student(/*...*/);
Mensch c = new Student(/*...*/);
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCH: Angenommen die Klasse `Mensch` hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. `Student` überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);
Student b = new Student(/*...*/);
Mensch c = new Student(/*...*/);
a.hallo(); // → Hallo ich bin ein Mensch
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCH: Angenommen die Klasse `Mensch` hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. `Student` überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);
Student b = new Student(/*...*/);
Mensch c = new Student(/*...*/);
a.hallo(); // → Hallo ich bin ein Mensch
b.hallo(); // → Hallo ich bin ein Student
```

DYNAMISCHE BINDUNG

- Attribute werden Java *statisch* und Methoden *dynamisch* gebunden.
- Doch was bedeuten diese Bindungsarten:

DYNAMISCH: Angenommen die Klasse `Mensch` hat die Methode `hallo()`, die einfach „Hallo ich bin ein Mensch“ ausgibt. `Student` überschreibt nun `hallo()`. Sie gibt nun „Hallo ich bin ein Student“ aus. Wir erhalten:

```
Mensch a = new Mensch(/*...*/);
Student b = new Student(/*...*/);
Mensch c = new Student(/*...*/);
a.hallo(); // → Hallo ich bin ein Mensch
b.hallo(); // → Hallo ich bin ein Student
c.hallo(); // → Hallo ich bin ein Student
```

WENN VERERBUNG NICHT REICHT

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
  Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
  Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.

- So können wir über unsere Listen iterieren,

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
  Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.

- So können wir über unsere Listen iterieren, über Bäume aber auch.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
  Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **`implements`** implementieren.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
  Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **`implements`** implementieren.
- Ein solches **`interface`** fordert gewisse Methoden vom Implementor.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
  Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **`implements`** implementieren.
- Ein solches **`interface`** fordert gewisse Methoden vom Implementor.
- Weiter kann ein **`interface`** seit Java 8 durch **`default`** Standardimplementationen für die Methoden liefern.

WENN VERERBUNG NICHT REICHT

- Java erlaubt *keine* Mehrfachvererbung.
  Allerdings können Klassen mehrere „Eigenschaften“ erfüllen.
- So können wir über unsere Listen iterieren, über Bäume aber auch.
- Eine Java Klasse kann (beliebig viele) Schnittstellen mittels **`implements`** implementieren.
- Ein solches **`interface`** fordert gewisse Methoden vom Implementor.
- Weiter kann ein **`interface`** seit Java 8 durch **`default`** Standardimplementationen für die Methoden liefern.
- Ein **`interface`** kann *keine* Attribute definieren, diese werden automatisch zu Konstanten.

WENN VERERBUNG NICHT REICHT

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.
- Da *Generics* kein Teil der Vorlesung sind, wurde das Interface *Sortable* definiert:

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.
- Da *Generics* kein Teil der Vorlesung sind, wurde das Interface *Sortable* definiert:

```
public interface Sortable {
    boolean le(Sortable o); // '<='
    boolean lt(Sortable o); // '<'
    boolean eq(Sortable o); // '=='
}
```

WENN VERERBUNG NICHT REICHT

- So gibt es zum Beispiel das Java-Interface `Iterable` welches anzeigt, dass man über den Datentyp (mittels for-each) iterieren kann.
- Da *Generics* kein Teil der Vorlesung sind, wurde das Interface *Sortable* definiert:

```
public interface Sortable {
    boolean le(Sortable o); // '<='
    boolean lt(Sortable o); // '<'
    boolean eq(Sortable o); // '=='
}
```

- Dies zeigt auch, wie man Interface-Bezeichnet wie (abstrakte) Klassen als Typanforderung liefern kann. (Polymorphieeee)

INTERFACE BEISPIEL

```
public class Element implements Sortable /*, interfaceB, ... */ {
    int value;
    Element next;

    // Nur als Beispiel:
    public boolean le(Sortable o) {
        return this.value <= ((Element)o).value;
    }

    // ...
}
```

WEITERE KOMMENTARE

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **`instanceof`** erfüllt.

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **`instanceof`** erfüllt.
- Ein Interface wie `Sortable` wird in Java durch `Comparable` mit nur einer einzelnen Funktion `compareTo()` gelöst.

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **`instanceof`** erfüllt.
- Ein Interface wie `Sortable` wird in Java durch `Comparable` mit nur einer einzelnen Funktion `compareTo()` gelöst.
- Interfaces sollten dann verwendet werdenn wenn es um eine Funktionalität geht (wie `Drawable`, `Moveable`, `Consumeable`, …)

WEITERE KOMMENTARE

- Auch für ein Interface wird die *is-a* Relation durch **`instanceof`** erfüllt.
- Ein Interface wie `Sortable` wird in Java durch `Comparable` mit nur einer einzelnen Funktion `compareTo()` gelöst.
- Interfaces sollten dann verwendet 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

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

  SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

  SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.

  LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

  SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.

  LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.

- Wie schwer ein Fehler ist, hängt von der Situation ab.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

  SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.

  LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.

- Wie schwer ein Fehler ist, hängt von der Situation ab.

- Die Fehlerbehandlung erfolgt meist durch die aufrufende Methode.

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

  Schwere: solche Fehler sind kritisch und führen zu einem Programmabbruch.

  Leichte: solche Fehler können (zur Laufzeit) korrigiert werden.

- Wie schwer ein Fehler ist, hängt von der Situation ab.

- Die Fehlerbehandlung erfolgt meist durch die aufrufende Methode.

- Fehler sind in Java Objekte von Klasse,

ARTEN VON FEHLERN

- Wir unterscheiden zwei Arten von Fehlern:

  SCHWERE: solche Fehler sind kritisch und führen zu einem Programmabbruch.

  LEICHTE: solche Fehler können (zur Laufzeit) korrigiert werden.

- Wie schwer ein Fehler ist, hängt von der Situation ab.

- Die Fehlerbehandlung erfolgt meist durch die aufrufende Methode.

- Fehler sind in Java Objekte von Klasse, die von `Exception` erben.

ARTEN VON FEHLERN

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **`throw`** werfen.

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **`throw`** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **`null`**, …)

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **`throw`** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **`null`**, …)
- Wir werfen eine explizite Ausnahme (allgemein: ein `Throwable`):

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **`throw`** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **`null`**, …)
- Wir werfen eine explizite Ausnahme (allgemein: ein `Throwable`):

```
throw new IndexOutOfBoundsException();
```

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **`throw`** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **`null`**, …)
- Wir werfen eine explizite Ausnahme (allgemein: ein `Throwable`):

```
throw new IndexOutOfBoundsException();
```

- Wird eine Ausnahme nicht behandelt, bricht das Programm ab.

ARTEN VON FEHLERN

- Eine *explizite* Ausnahme können wir mittels **`throw`** werfen.
- Eine *implizite* Ausnahme wird von der Java Virtual Machine geworfen (Division durch Null, Zugriff auf **`null`**, …)
- Wir werfen eine explizite Ausnahme (allgemein: ein `Throwable`):

```
throw new IndexOutOfBoundsException();
```

- Wird eine Ausnahme nicht behandelt, bricht das Programm ab.
- Java-Errors, wie ein `InternalError` lassen sich nicht sinnvoll behandeln!

BEHANDELN VON FEHLERN

BEHANDELN VON FEHLERN

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

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

BEHANDELN VON FEHLERN

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

```
try {
    // Anweisungen, Methodenaufrufe, ...
} catch(⟨FehlerKlasseA⟩ ⟨Bezeichner⟩) {
    // 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

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.
- *Checked* Exceptions *müssen* behandelt werden:

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.
- *Checked* Exceptions *müssen* behandelt werden:
  - Entweder direkt durch ein **`try-catch`** (welches die Exception behandelt).

CHECKED- & UNCHECKED EXCEPTIONS

- Alle Exceptions erben in Java von `Exception`.
- Solche, die von `RuntimeException` erben sind *unchecked*, die anderen sind *checked*.
- *Checked* Exceptions *müssen* behandelt werden:
  - Entweder direkt durch ein **`try-catch`** (welches die Exception behandelt).
  - Oder durch eine Weiterreichung per **`throws`** bei der Methoden-Deklaration:

```
public void foo() throws Exception/*, ExceptionB, ...*/ {
    throw new Exception();
}
```

EXCEPTIONS WEITERWERFEN

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **`try-catch`** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **`try`**-**`catch`** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **`try`**-**`catch`**-Konstrukte verschachteln.

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **`try-catch`** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **`try-catch`**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {
    try {
        throw new Exception();
    } catch (Exception ex) {



    }
}
```

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **`try-catch`** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **`try-catch`**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {
    try {
        throw new Exception();
    } catch (Exception ex) {



    }
}
```

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **`try-catch`** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **`try-catch`**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {
    try {
        throw new Exception();
    } catch (Exception ex) {
        // do stuff ..

    }
}
```

EXCEPTIONS WEITERWERFEN

- Die anderen Blöcke eines **`try-catch`** sind nicht magisch. Auch in ihnen können Exceptions geworfen werden.
- So lassen sich übrigens auch **`try-catch`**-Konstrukte verschachteln.
- Rethrows können die Verantwortung auch weiterreichen:

```
public void foo() throws Exception {
    try {
        throw new Exception();
    } catch (Exception ex) {
        // do stuff ..
        throw ex; // oder sowas wie: throw new Exception(ex)
    }
}
```

BEHANDELN VON FEHLERN

BEHANDELN VON FEHLERN

- Bei mehreren **`catch`**-Blöcken wird von oben nach unten ein passender gesucht.

BEHANDELN VON FEHLERN

- Bei mehreren **`catch`**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von `Exception`),

BEHANDELN VON FEHLERN

- Bei mehreren **`catch`**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von `Exception`), sollten die „allgemeineren“ weiter unten stehen.

BEHANDELN VON FEHLERN

- Bei mehreren **`catch`**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von `Exception`), sollten die „allgemeineren“ weiter unten stehen.
- Beispiel:

BEHANDELN VON FEHLERN

- Bei mehreren **`catch`**-Blöcken wird von oben nach unten ein passender gesucht. Erben die Fehler also voneinander (so wie alle von `Exception`), sollten die „allgemeineren“ weiter unten stehen.

- Beispiel:

```
try {
    System.out.println(42 / 0);
} catch (ArithmeticException ex) {
    // Behandlung
    System.err.println("Division␣durch␣0!");
    ex.printStackTrace();
}
```

WEITERES — DIE WEIHNACHTSWIEDERHOLUNG

WEITERES — DIE WEIHNACHTSWIEDERHOLUNG

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