11

Von Erben, Scherben und dem Schnittstelle-werden

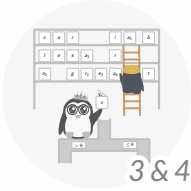
The final Tut

Theorie

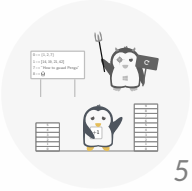
2

13

Grundlagen



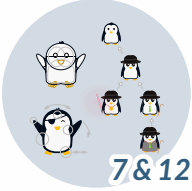
3 & 4



5

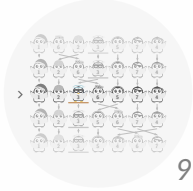
6

Vertiefungen

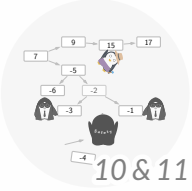


7 & 12

8



9



10 & 11

Kurzwiederholung

Diesmal am Ende, als Zusammenfassung!

Präsenzaufgabe

Präsenzaufgabe

1

Dir streichen wir das Erbe!

RegularPolygon.java

```
public abstract class RegularPolygon {
   protected final int numSides;
   protected final double sideLength;
   public RegularPolygon(int numSides, double sideLength) {
      this.numSides = numSides;
      this.sideLength = sideLength;
   }
   public int getNumSides() { return numSides; }
   public double getCircumference() { return numSides * sideLength; }
   public abstract double getArea();
}
```

Erstellen Sie eine neue Klasse `EquilateralTriangle`, welche ein gleichseitiges Dreieck repräsentieren und von `RegularPolygon` erben soll. Erzeugen Sie einen Konstruktor, welcher die Seitenlänge a als Parameter verarbeitet ($A = 1/4 \cdot \sqrt{3} \cdot a^2$).

How to legacy

EquilateralTriangle.java

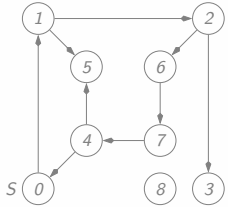
```java
public class EquilateralTriangle extends RegularPolygon {
  public EquilateralTriangle(double sideLength) {
    super(3, sideLength);
  }

  @Override
  public double getArea() {
    return 0.25 * Math.sqrt(2) * sideLength * sideLength;
  }
}
```

Übungsblatt 11

Aufgabe 1: Graphen

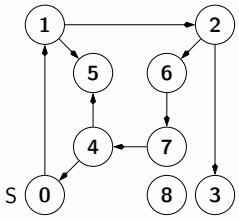
Geben sei der folgende Graph G:



a) *Geben Sie die Adjazenzliste für G an.*
b) *Geben Sie die Adjazenzmatrix für g an.*
c) *In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?*
d) *In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?*

Eine Adjazenzliste

a) Geben Sie die Adjazenzliste für G an.



0 → 1

1 → 2 → 5

2 → 3 → 6

3

4 → 0 → 5

5

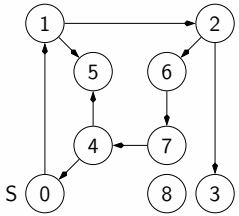
6 → 7

7 → 4

8

Eine Adjazenzmatrix

b) Geben Sie die Adjazenzmatrix für G an.



	0	1	2	3	4	5	6	7	8
0	0	1	0	0	0	0	0	0	0
1	0	0	1	0	0	1	0	0	0
2	0	0	0	1	0	0	1	0	0
3	0	0	0	0	0	0	0	0	0
4	1	0	0	0	0	1	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	1	0
7	0	0	0	0	1	0	0	0	0
8	0	0	0	0	0	0	0	0	0

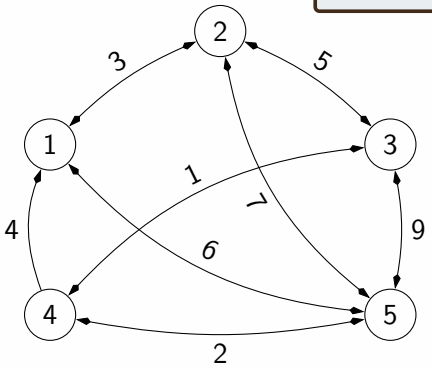
(Nullen können mit einem Kommentar, dass alle freien Einträge automatisch „0“ sind, auch weggelassen werden.)

Beispiel: Die Gegenrichtung

- Nun eine gewichtete Adjazenzmatrix (Kanten wie ⟷ können auch ungerichtet dargestellt werden):

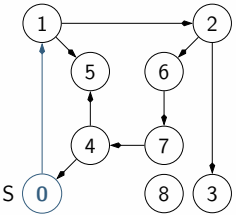
Natürlich können die Kanten in die andere Richtung auch ein anderes Gewicht aufweisen.

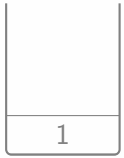
	1	2	3	4	5
1	0	3	0	0	6
2	3	0	5	0	7
3	0	5	0	1	9
4	4	0	1	0	2
5	6	7	9	2	0



Eine Tiefendurchlauf

c) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?





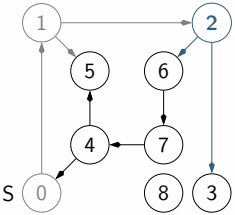
Stack

→ 0

Die Reihenfolge in der mehrere, noch nicht besuchte Nachbarn, auf den Stack gelegt werden, ist willkürlich und hängt von der Implementation ab.

Eine Tiefendurchlauf

c) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?

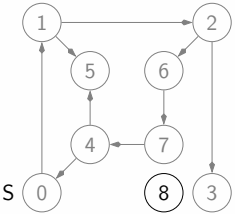


Stack

→ 0, 1, 2

Eine Tiefendurchlauf

c) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Tiefendurchlaufs“ traversiert wird?

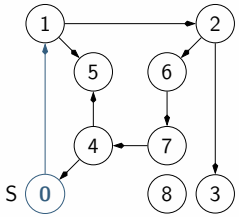


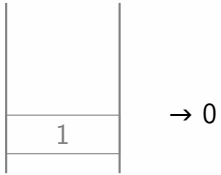
Stack

→ 0, 1, 2, 3, 6, 7, 4, 5

Eine Breitendurchlauf

d) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?

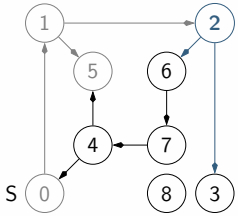


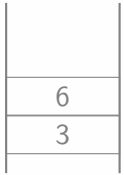


Erneut ist die Reihenfolge bei mehreren Nachbarn willkürlich.

Eine Breitendurchlauf

d) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?



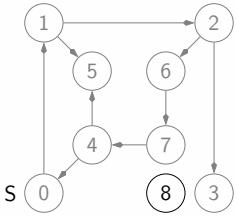


Queue

→ 0, 1, 5, 2

Eine Breitendurchlauf

d) In welcher Reihenfolge werden die Knoten besucht, wenn G mittels eines „Breitendurchlaufs“ traversiert wird?



Queue

→ 0, 1, 5, 2, 3, 6, 7, 4

Aufgabe 2: Stacks

In dieser Aufgabe sollen Sie einen `Stack` implementieren, welcher nach dem `LIFO`-Prinzip arbeitet (Last In First Out) und `int` Werte verwaltet. Nutzen Sie hierfür wieder die Element.java Klasse. Legen Sie die Klasse `Stack.java` an und implementieren Sie die folgenden Methoden:

a) *Implementieren Sie einen Konstruktor, der den Stack erstellt. Der Stack soll eine private Referenz auf das oberste Element, sowie eine private Instanzvariable, die die Größe des Stacks darstellt, besitzen.*

b) *Die Methode `public void push(int value)` soll den übergebenen Wert oben auf dem Stack platzieren.*

c) *Die Methode `public int pop()` soll den obersten Wert vom Stack entfernen und zurückgeben. Beachten Sie, dass der Stack unter Umständen leer sein kann. In diesem Fall soll eine `NoSuchElementException` ausgelöst werden.*

d) *Testen Sie Ihre Implementierungen, indem Sie einige Werte auf den Stack legen und wieder entfernen.*

Ein Stack für die Masse

Stack.java

```
1  public class Stack {
2     private Element top;
3     private int size;
4     public Stack() {
5        this.top = null;
6        this.size = 0;
7     }
8  ›  public void push(int value) {
9        Element newTop = new Element();
10       newTop.setValue(value);
11       newTop.setNextElement(this.top);
12       this.top = newTop;
13       size += 1;
14    }
15
16 }
```

```
15 public int pop() {
16    if (size <= 0)
17       throw new NoSuchElementException();
18    int value = top.getValue();
19    top = top.getNextElement();
20    size -= 1;
21    return value;
22 }
```

▷ display in browser

```
23 public static void main(String[] args) {
24    Stack stack = new Stack();
25    stack.push(3);
26    stack.push(4);
27    System.out.println(stack.pop());
28 }
```

top

Ein Stack für die Masse

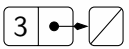
Stack.java

```
1  public class Stack {
2     private Element top;
3     private int size;
4     public Stack() {
5        this.top = null;
6        this.size = 0;
7     }
8     public void push(int value) {
9        Element newTop = new Element();
10       newTop.setValue(value);   value=3
11       newTop.setNextElement(this.top);
12       this.top = newTop;
13 >     size += 1;   size=1
14    }
15
16 }
```

```
15 public int pop() {
16    if (size <= 0)
17       throw new NoSuchElementException();
18    int value = top.getValue();
19    top = top.getNextElement();
20    size -= 1;
21    return value;
22 }
```

▷ display in browser

```
23 public static void main(String[] args) {
24    Stack stack = new Stack();
25    stack.push(3);
26    stack.push(4);
27    System.out.println(stack.pop());
28 }
```



newTop, top

Ein Stack für die Masse

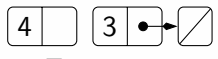
Stack.java

```
1  public class Stack {
2     private Element top;
3     private int size;
4     public Stack() {
5        this.top = null;
6        this.size = 0;
7     }
8     public void push(int value) {
9        Element newTop = new Element();
10 >     newTop.setValue(value);   value=4
11       newTop.setNextElement(this.top);
12       this.top = newTop;
13       size += 1;
14    }
15
16 }
```

```
15 public int pop() {
16    if (size <= 0)
17       throw new NoSuchElementException();
18    int value = top.getValue();
19    top = top.getNextElement();
20    size -= 1;
21    return value;
22 }
```

▷ display in browser

```
23 public static void main(String[] args) {
24    Stack stack = new Stack();
25    stack.push(3);
26    stack.push(4);
27    System.out.println(stack.pop());
28 }
```



Ein Stack für die Masse

Stack.java

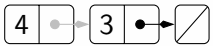
```java
public class Stack {
   private Element top;
   private int size;
   public Stack() {
      this.top = null;
      this.size = 0;
   }
   public void push(int value) {
      Element newTop = new Element();
      newTop.setValue(value);
      newTop.setNextElement(this.top);
      this.top = newTop;
      size += 1;
   }

}
```

```java
public int pop() {
   if (size <= 0)
      throw new NoSuchElementException();
   int value = top.getValue();
   top = top.getNextElement();
   size -= 1;
   return value;
}
```

▷ display in browser

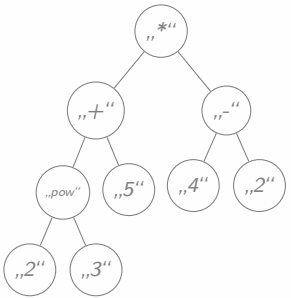
```java
public static void main(String[] args) {
   Stack stack = new Stack();
   stack.push(3);
   stack.push(4);
   System.out.println(stack.pop());   4
}
```



top

Aufgabe 3: Post-Order Traversierung von Binärbäumen

Ein `BinaryExpressionTree` repräsentiert einen Ausdruck, bei dem die Operatoren höchstens zwei Operanden verarbeiten können. So an die Berechnungen $(2^3 + 5) * (4 - 2)$ durch den folgenden Baum ausgedrückt werden:



Die Darstellungsweise erlaubt rekursives abarbeiten des Ausdrucks. Dazu wird jeweils ein Knoten evaluiert: handelt es sich um einen Operator (kein Blatt und somit einen Rekursionsfall), wird dieser wieder evaluiert. Handelt es sich um einen Operanden (Blatt und somit ein Basisfall), kann das Ergebnis berechnet werden.

Wir beschränken uns hier einfachheitshalber nur auf balancierte Binärbäume, das heißt, es können nicht beliebige Ausdrücke dargestellt werden. Sie finden in den Unterlagen (zu diesem Übungsblatt) die Klasse StringNode.java, die die Knoten darstellen soll, sowie die Klasse BinaryExpressionTree.java, die den Baum selbst repräsentiert, sowie BinaryExpressionTreeMain.java als Programmeinstiegspunkt.

Implementieren Sie die Methode `public double traverse(StringNode node)`, die den Binärbaum wie beschrieben rekursiv im post-Order Verfahren traversiert.

Ihre Implementierung soll mindestens die Operatoren `+`, `-`, `*`, `/`, sowie `pow` unterstützten. Enthält ein Knoten einen unbekannten Operator, soll eine `InvalidParameterException` ausgelöst werden. Testen Sie Ihre Implementierung an dem obigen Beispiel. Die Methode `private void insertNodes(...)` der Klasse `BinaryExpressionTree` erstellt einen Binärbaum aus einem gegebenen String-Array.

Evaluate the World

BinaryExpressionTree.java, StringNode.java

```java
public class BinaryExpressionTree {
   public double traverse(StringNode node) {
      if (node == null) throw new IllegalStateException();
      if (node.isLeaf()) return Double.parseDouble(node.getItem());

      double valueLeft = traverse(node.getLeftChild());
      double valueRight = traverse(node.getRightChild());
      switch (node.getItem()) {
         case "+": return valueLeft + valueRight;
         case "-": return valueLeft - valueRight;
         case "*": return valueLeft * valueRight;
         case "/": return valueLeft / valueRight;
         case "pow": return Math.pow(valueLeft, valueRight);
         default: throw new InvalidParameterException();
      }
   }
}
```

Everything on Main!

BinaryExpressionTreeMain.java

```java
public class BinaryExpressionTreeMain {
    ▷ display in browser
    public static void main(String[] args) {
        String[] items = {"*", "+", "-", "pow", "5", "4", "2", "2", "3"};
        BinaryExpressionTree tree = new BinaryExpressionTree(items);

        double result = tree.evaluate();
        System.out.println("Ergebnis: " + result);
    }
}
```

Aussicht: Übungsblatt `Integer.MAX_VALUE`

Aussicht

- Die wichtigsten Punkte betrachten wir im Rahmen der Wiederholung
- Auch hier lohnt sich eine Wiederholung der bereits bekannten Begriffe
  - Überschreiben
  - Überladen
  - Überschatten
  - Signatur
  - …
- Neben Schnittstellen sind statische & dynamisch Bindung, sowie Polymorphie wichtig

Abschließendes

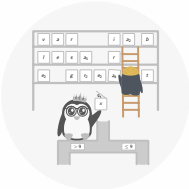
Algorithmen

- Totale Korrektheit
  - *Terminiertheit:* Endliche Schritte für jede Eingabe
  - *Partielle Korrektheit:* Wenn terminiert, dann korrekt

- Weitere Eigenschaften
  - *Determiniertheit:* Gleiche Eingabe → Gleiche Ausgabe
    Einer determinierten Person ist egal, *wie* sie ihr Ziel erreicht.
  - *Determinismus:* Gleiche Eingabe → Gleiche Zustandsfolge

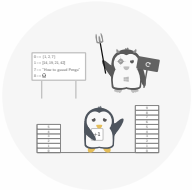
Spezifikation (Begriffe mit Problemrelevanz) › Abstraktion (Gegeben & Gesucht) › **Entwurf** (Algorithmus) › Verifikation (Termination & partielle Korrektheit) › Aufwandsanalyse (Laufzeitverhalten)

Konstrukte



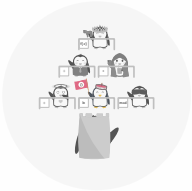
- *Implizit*: byte → short → int → long → float → double
  Zahlen von klein zu groß, sowie: char → int
- *Präzedenzregeln*:
  Post vor Prä, sonst wie Arithmetik & Logik
- *Default-Werte*: Zahlen und Zeichen `0`, Boolean `false`, Rest `null`
  Nur bei: Arrays, Instanz- und Klassenvariablen (JLS17 4.12.5)
- *Überschatten*:
  Lokaler Bezeichner überdeckt Gültigkeit des globalen

Arrays & Iteration



- Arrays sind komplexe Datentypen
- Mehrdimensionale Arrays sind eindimensionale Arrays von seindimensionalen Arrays von…
- Die drei Schleifenarten sind gleich mächtig
  - Maximum bekannt: `for`
  - Mindestens ein mal: `do-while`
  - Sonst: `while`

Unterprogramme



- *Überladung:* Gleicher Name, andere Signatur
  - *Signatur*: Name & Parametertypliste
  - Müssen zudem in selber Klasse sein (später: Vererbung)
- Beim Aufruf macht Java call-by-value:
  - Alle Parameter werden kopiert (Stack)
- `void` gibt als Keyword an, dass die Methode keinen Rückgabetyp hat

Objektorientierte Progammierung

- Eine Klasse definiert die Blaupause für Objekte
  - Attribute definieren den Zustand
  - Methoden definieren den Verhalten
  - Statische Elemente sind nicht Teil der Blaupause (sie gehören der Klasse!)
- Der Konstruktor baut den initialen Zustand
  - *Instanziierung*: Erzeugen eines neuen Objektes
  - Wenn keiner: erzeugt Java den leeren Standardkonstruktor
  - `this` erlaubt Aufruf von überladenen Konstruktoren
- Klassen, Methoden, …: *Sichtbarkeit* (**`public`**, …)
- *Gültigkeit*sbereich: Wo die Variablen „deklariert sind“ (Überschatten, …)

Rekursion

- Methoden, die sich direkt oder indirekt selbst aufrufen sind rekursiv.
  - Ruft sich eine Methode maximal einmal selbst auf, ist sie *linear rekursiv*.

$$f(x) = \begin{cases} 1 & \text{if } x < 2 \\ f(x-1) \cdot x & \text{otherwise} \end{cases}$$

```java
public int f(int x) {
   if(x < 2) return 1;
   else return f(x - 1) * x;
}
```

  - *Kopfrekursiv*, wenn dieser Aufruf das erste Statement ist (alles passiert im Aufstieg)
  - *Endrekursiv*, wenn dieser Aufruf das letzte Statement ist (alles passiert im Abstieg)

Head-Recursive

```java
public int f(int x) {
   if(x < 2) return 1;
   else return f(x - 1) * x;
}
```

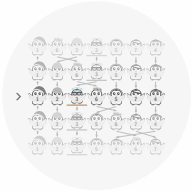
Tail-Recursive, call as f(x, 1)

```java
public static int f(int x, int acc) {
   if(x < 2) return acc;
   else return f(x - 1, acc * x);
}
```

  - Ruft sie sich auch mehrfach pro Rekursionsfall auf, ist sie *verzweigt rekursiv*.

Suchen und Sortieren

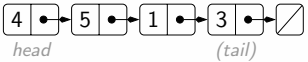
Stabil?		Laufzeit ($\mathcal{O}(\ldots)$) best	worst	Ansatz
✓	Bubble	n^2, n	n^2	Vertausche benachbarte El., solange unsortiert.
✓	Insertion	n	n^2	Sortiere 1. unsortiertes El. in sortierten Teil ein.
	Selection	n^2	n^2	Kleinstes unsortiertes El. an Ende des sortierten Teils.
✓	Merge	n log n	n log n	Aufteilen bis einel., wiederholtes mergen sortierter Teillisten.
	Quick	n log n	n^2	Pivot → Ende, ℓ solang <, r solang ≥. Treffen → tausche Pivot.
	Heap	n log n	n log n	Baue Heap, entferne wiederholt Wurzel, heapify.



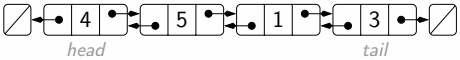
Dies folgt den Implementationen der Vorlesung. Bereits leichte Modifikationen (wie: prüfe zuerst ob die Liste bereits sortiert ist) können die Daten verändern (beispielsweise einen best-case von n *im Falle von Bubblesort).*

Dynamische Datenstrukturen

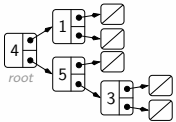
Einfach verkettete Liste:



Doppelt verkettete Liste:



Einfach verketteter Binärbaum:




```
void inorder(Node n) {
   if (n == null) return;
   inorder(n.left);
   System.out.println(n.value);
   inorder(n.right);
}
```

```
private class Elem {
  public final int value;
  public Elem prev, next;
  public Elem(int v, Elem p, Elem n) {
    value = v; prev = p; next = n;
  }
}       Elem head, tail;
        public void addFront(int value) {
           Elem elem = new Elem(value, null, head);
           if(head == null) tail = elem;
           else head.prev = elem;
           head = elem;
        }
```

Weiterführende Konzepte der Objektorientierung

- Vererbung (`extends`) erfolgt in Java nach dem Erweiterungsprinzip
  - Man kann nur von einer Klasse erben (keine Mehrfachvererbung)
  - Methoden und Attribute der Elternklasse sind über `super` erreichbar
  - Attribute werden statisch, Methoden dynamisch gebunden!
- Erbt `Y` von `X` (`Y extends X`) gilt „Y *is-a* X“

```
X x = new Y();
```

`x instanceof X` ⇝ true `x instanceof Y` ⇝ true

- Wir können Methoden der Elternklasse *überschreiben* (gleiche Signatur & sichtbar)
- Von *abstrakten Klassen* kann keine Instanz erzeugt werden
  - Sie erlauben die Deklaration abstrakter Methoden
  - Nicht-abstrakte erbende Subklassen müssen abstrakte Methoden überschrieben!
- *Interfaces* (`interface`) fordern Methoden von implementierenden Klassen
  - Eine Klasse kann mehrere Interfaces implementieren (`A implements X, Y, Z`)
  - Interfaces fordern nur Verhalten (`default`, keinen Zustand, vs. abstrakte Klasse)
  - ⇝ keine „Attribute“ (automatisch `final` & `static`)