

Modul 411

|  |
| --- |
| Datenstrukturen und Algorithmen |
| Dokumentation |

Inhaltsverzeichnis

[1 Theorie 2](#_Toc526307489)

[1.1 Begriffe 2](#_Toc526307490)

[1.2 Anleitungen 5](#_Toc526307491)

[1.3 Rund um Datentypen 5](#_Toc526307492)

[1.4 Operatoren 6](#_Toc526307493)

[1.5 Dokumentation 7](#_Toc526307494)

[1.6 JUnit 7](#_Toc526307495)

[2 Codebeispiele 8](#_Toc526307496)

[2.1 Schleifen und Verzweigungen 8](#_Toc526307497)

[2.2 Examples 9](#_Toc526307498)

[2.3 Steuerzentrale vom Roboter 10](#_Toc526307499)

[2.4 Exception 11](#_Toc526307500)

[2.5 JUnit 12](#_Toc526307501)

# 

# Theorie

## Begriffe

**Datenstrukturen**

Eine Datenstruktur ist ein Objekt zur Speicherung und Organisation von Daten. Die Daten werden in einer bestimmten Art und Weise angeordnet und verknüpft, um den Zugriff auf sie und ihre Verwaltung effizient zu ermöglichen.

* Array
* ArrayList
* LinkedList
* Stack

**Algorithmen**

Verfahren zur Lösung eines Problems.

* Sortieralgorithmen (Bubble-Sort, Quick-Sort, Insert-Sort)
* Suchalgorithmen
* Kalenderrechnung
* Kryptographie

**Bubble-Sort**

Bubble-Sort dient dazu, die Elemente innerhalb eines Arrays nach Grösse zu sortieren

int t = 0;

int n = ul.length;

**for** (int i=n-1; i>0; i--)**{**

**for**(int j = 0; j<i; j++) **{**

**if**(ul[j] > ul[j+1])**{**

t = ul[j];

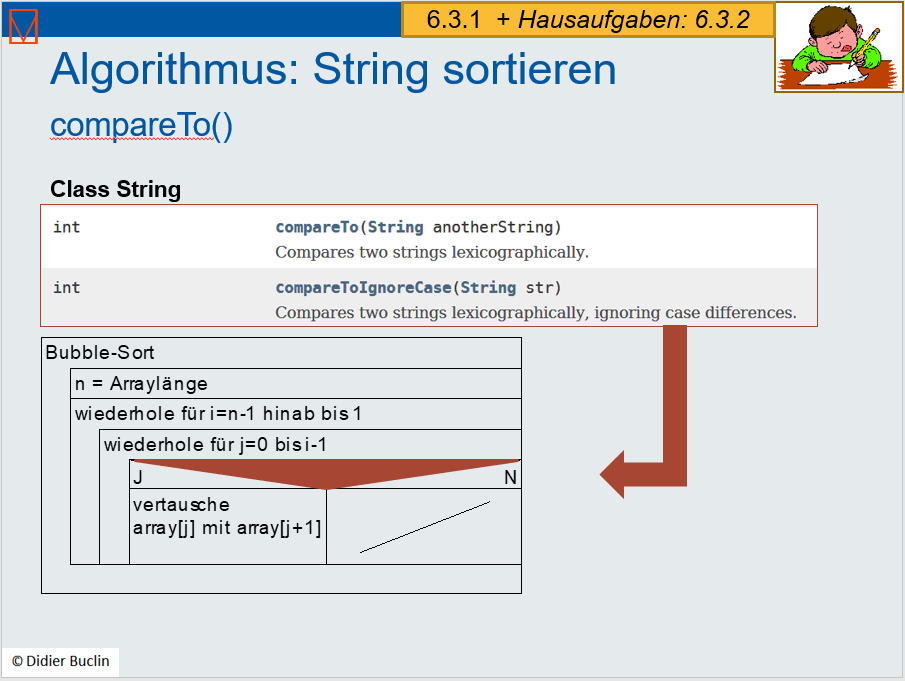
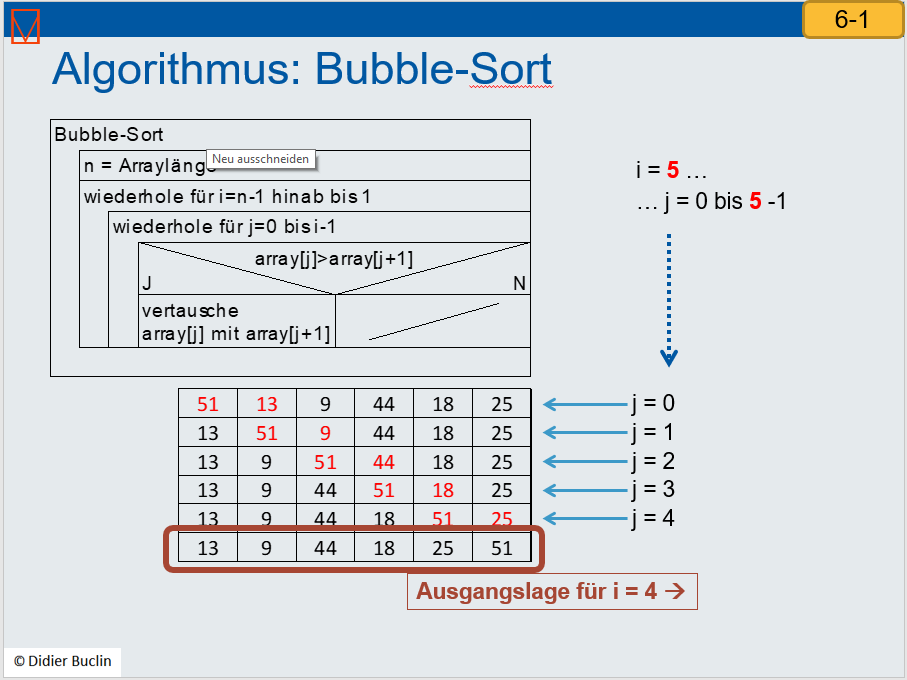
ul[j]=ul[j+1];

ul[j+1] = t;

**}**

**}**

**}**



**LIFO**

LIFO steht für **Last in - First out**. Die Methode **push()** funktioniert nach diesem Prinzip. Es fgt ein Element zu oberst ein, während **pop()** das oberste Element abruft und löscht.

public class Stack {

Node top;

int length;

public static void main(String[] args) **{**

Stack stack = new Stack();

String input = "I want to reverse this.";

String[] list = input.split(" ");

**for** (int i = 0; i < list.length; i++) **{**

stack.push(list[i]);

**}**

int stackLength = stack.length;

**for**(int j = 0; j < stackLength; j++) **{**

System.out.println(stack.pop());

**}**

**}**

private class Node **{**

private String value;

private Node next;

**}**

public void push(String value) **{**

Node node = new Node();

node.next = top;

node.value = value;

top = node;

length++;

**}**

public String pop() **{**

**if** (top != null) **{**

String result = top.value;

top = top.next;

length--;

return result;

**}**

else **{**

return null;

**}**

**}**

**}**

**Rekursive Methoden**

Bei der rekursiven Programmierung ruft sich eine Prozedur, Funktion oder Methoe in einem Computerprogramm selbst wieder auf.

Code ist einfacher zum lesen, jedoch komplexer als wenn es iterativ geschrieben wäre (mit Schleifen).

**Pro** // ist übersichtlicher

**Kontra** // schlechtere Performance, Hoher Speicherverbrauch

public class Main **{**

public static void main(String[] args) **{**

System.out.println(*faculty*(5));

**}**

private static int *faculty*(int i) **{**

if(i == 1) **{**

return i;

**}**

return i\**faculty*(i-1);

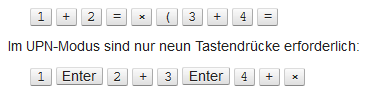
**}**

**}**

**UPN Rechnung**

Umgekehrte polnische Notation ¯\\_(ツ)\_/¯

Das ist diese komische Rechnung, welche Herr Buclin im Unterricht zeigte.



public class Rechner **{**

public void rechnen(String input) **{**

Double ersteZahl;

Double zweiteZahl;

Stack<Double> stack = new Stack<Double>();

// Die einzelnen Zahlen und Operatoren auseinandernehmen und

// als einzelne Elemente eines Array parts speichern.

String[] elements = input.split(" ");

// Die einzelnen parts werden nach einander bearbeitet:

**for**(int i = 0; i < elements.length; i++) **{**

try **{**

// Wenn ein part eine Zahl ist --> Auf den Stack speichern

stack.push(Double.parseDouble(elements[i]));

**}** catch(NumberFormatException e) {

// Wenn ein part ein Operator ist (+-\*/) --> berechnen

Double b = stack.pop();

Double a = stack.pop();

**switch**(elements[i]) **{**

case "+" :

stack.push(a + b);

break;

case "-" :

stack.push(a - b);

break;

case "\*" :

stack.push(a \* b);

break;

case "/" :

stack.push(a / b);

break;

**}**

**}**

**}**

// Ausgabe des Resultats

System.out.println(stack.pop());

**}**

**}**

package m411\_stackRechner;

public class Main **{**

public static void main(String[] args) {

// TODO Auto-generated method stub

Rechner rechner = new Rechner();

rechner.rechnen("4 5 + 6 + 3 /"); // = (4 + 5 + 6) / 3 => 5

rechner.rechnen("4 3 6 + \*"); // = 4 \* ( 3 + 6 ) => 36

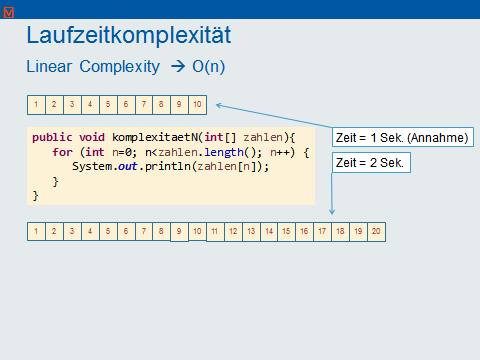
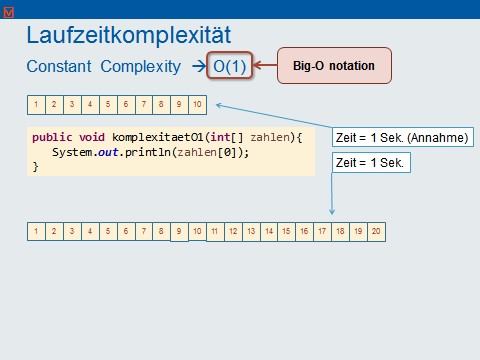
rechner.rechnen("4 2 6 + 2 / \* 5 +"); // = 4 \* ( ( 2 + 6 ) / 2 ) + 5 => 21

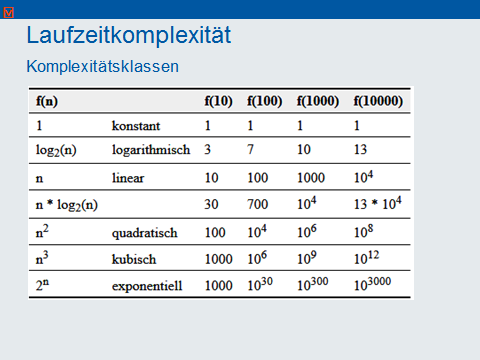
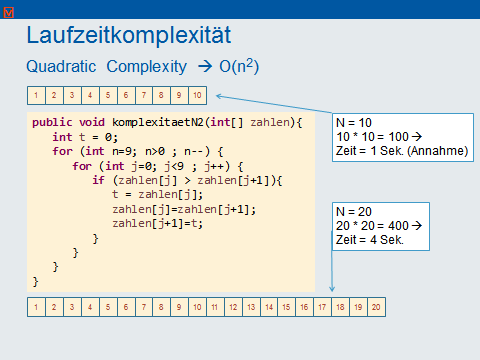
**}**

**}**

**Laufzeitkomplexität**

Ich bin mir sicher, wir haben die Komplexität verstanden. Keine weiteren Erläuterungen.

****



**Ausgabe**

Wörter und Zahlen kann man beide ganz einfach mit dem gleichen Befehl über die Konsole ausgeben und dabei kann man sogar noch rechnen:

int x = 4;

int y = 6;

System.out.println("Die Summe von "+ x +" und "+ y +" ist "+ (x+y));

**Eingabe**

Um hingegen benutzereingaben einzulesen muss man ein Objekt der Klasse Scanner, welche eine Klasse der Standard Java-Library ist erzeugen.

Scanner scan = new Scanner(System.in);

int a = scan.nextInt();

String s = scan.next(); // scannt bis zum ersten Leerzeichen

String s2 = scan.nextLine(); // scannt die gesamte Eingabe

Mit *LocalDate.now().getYear();* erhält man das momentane Jahr.

Mit *LocalDate.now().getMonthValue();* erhält man den momentanen Monat.

Mit *int dayOfMonth = LocalDate.now().getDayOfMonth();* erhält man den aktuellen Tag.

**BufferedReader**

Die Klasse BufferedReader aus der Java Library wird benutzt, um Textfiles einzulesen.

Erstmal um ein BufferedReader-Objekt zu Instanziieren muss man im Konstruktor angeben von welcher Quelle gelesen werden soll. In diesem Beispiel sollen Text-Dateien gelesen werden. Entsprechend ist die Quelle ein FileReader-Objekt. Der Konstruktor der FileReader-Klasse erfordert den Pfad zum zu lesenden File als String. Mit der Instanz des FileReader als Parameter erstellen wir zunächst eine BufferReader Instanz.

Mit dem folgenden Code kann man den Text (in diesem Fall ist es keine Text Datei, sondern ein String welcher die Struktur einer CSV Datei hat) bei jedem Semikolon unterteilt und dann ein wert pro Zeile ausgegeben.

String line = "Huber;Peter;1955";

String[] lineParts = line.split(";");

for (String p : lineParts) {

System.out.println(p);

}

Da man dem Benutzer eine Fehlermeldung geben möchte, wenn es ein Problem beim öffnen der Datei gab muss man dies mit try { } und catch { } machen.

Man kann für das einlesen von Files auch die Scanner Klasse verwenden was sich jedoch nicht empfiehlt da diese nicht "thred safe" ist, was bedeutet das man verhindern muss das ein gleichzeitiger Dateizugriff von Aussen stattfindet.

Als nächstes haben wir eine Aufgabe erledigt, bei welcher es darum ging eine CSV (Comma Separated Values) Datei die folgendermassen aus:

Huber;Peter;1955  
Meier;Irene;1960  
Muster;Anna;1974

Bei dieser Aufgabe ging es nicht mehr bei einem Zeilenumbruch den String zu teilen, sondern nach jedem Semikolon. Ausserdem mussten wir den Jahrgang in Jahre umwandeln.

**Codebeispiel BufferedReader**

public class main {

public main() {

/\* create reader variable \*/

BufferedReader bfr = null;

try {

String line;

/\* instantiate reader variable \*/

bfr = new BufferedReader(new FileReader(fileName));

/\* read as long as possible \*/

while ((line = bfr.readLine()) != null) {

String[ ] parts = line.split(";");

String name = parts[0];

String vorname = parts[1];

int age = LocalDate.now().getYear() - Integer.parseInt(parts[2]);  
 System.out.println(name+", "+vorname+", Alter: "+age+"Jahre");

}

bfr.close();

}

/\* in case something goes wrong... \*/

catch(IOException ex){

System.out.println(ex.toString());

ex.printStackTrace();

}

}

public static void main(String[ ] args) {

new main();

}

In einer CSV Datei werden die Werte mit Strichpunkten getrennt. Hierbei muss man also diese trennen. Dies kann man mit String[] werte = input.split(";"); bewerkstelligen. Um anschliessend diese zu iterieren benötigt man folgenden Code:

for (String w: werte) {

}

Weil Zahlen beim einlesen als String wahrgenommen werden, muss man diese «parsen». Dies kann man mit Integer.parseInt(i); oder Double.parseDouble(d); tun.

Die Abkürzung generate toString() in Eclipse erzeugt die geerbte Methode toString() von Objekt in die zu bearbeitende Klasse, damit man statt einer Objektreferenz eine selbstbestimmte Ausgabe erhält.

**Array**

Ein Array definiert sich dadurch, dass es eine feste Anzahl einzelner Daten eines bestimmten Datentyps zusammenhängend speichert, welche anhand eines Indexes zugänglich sind. Gespeichert werden können elementare Datentypen oder auch Objekte.

Ein Nachteil der Arrays ist die fest definierte Länge. Vorteilhaft jedoch ist die Effizienz auf SIMD-Computern und es ist die erste Wahl für Bildverarbeitung, Games und 3D.

Ein Array lässt sich zudem in einem Schritt erstellen mit nachfolgendem Code.

int testarray[] = { 5.3, 6.2 };

Nützliche Members sind beispielsweise das Attribut length, die Methoden clone() und sort().

Es ist ermöglicht ein Array direkt mit einer Grösse von 0 zu instanziieren und anschliessend dieses zu erweitern.

int[] baseArray = new int[0];

baseArray füllen:

baseArray = prepend(baseArray, i);

public static int[] prepend(int[] arr, int elem)**{**

int[] result = new int[arr.length+1];

result[0]=elem;

**for** (int i=0; i < arr.length; i++)**{**

result[i+1]=arr[i];

**}**

return result;

**}**

**Arraylist**

Ein ArrayList kann nur Objektreferenzen speichern, ist eine geordnete Liste bzw. Sequenz und ist dynamisch erweiterbar, d.h. man kann an beliebigen Orten Elemente einfügen.

**Pro** // vorteilhaft ist der effiziente Zugriff auf Elemente via Positionsindex.

**Kontra** // nachteilhaft ist die Ineffizienz bei der Löschung von Elementen und die Tatsache, dass es nicht thread-safe ist.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
|  |  | ElementC |  |  |

**Vector**

Ein Vector kann nur Objektreferenzen speichern, ist eine geordnete Liste bzw. Sequenz und ist dynamisch erweiterbar, d.h. man kann an beliebigen Orten Elemente einfügen.

**Pro** // vorteilhaft ist die Tatsache, dass es thread-safe ist, da es synchronisiert wird.

**Kontra** // wegen der Synchronisation ist es ineffizienter als ein ArrayList.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 |
| ElementA | ElementB | ElementC | ElementD | ElementE |

**LinkedList**

Ein LinkedList ist eine doppelt verkettete Liste, kann nur Objektreferenzen speichern, ist eine geordnete Liste bzw. Sequenz und ist dynamisch erweiterbar, d.h. man kann an beliebigen Orten Elemente einfügen.

**Pro** // vorteilhaft ist die Möglichkeit, die Liste von vorne nach hinten durchzuschreite und dass dasLöschen und Hinzufügen von Elementen verglichen zur ArrayList effizienter ist.

**Kontra** // nachteilhaft ist die Tatsache, dass es nicht thread-safe ist und dass das Auslesen von Elementen ineffizienter ist als die ArrayList.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 🗘 🗘 🗘 🗘 🗘 | | | | |
| ElementE | ElementA | ElementB | ElementC | ElementD |

Wichtige Befehle für die LinkedList sind Nachfolgende.

linkedlist.add(4);

linkedlist.add(6);

linkedlist.add(7);

linkedlist.add(10);

linkedlist.add(12);

linkedlist.addFirst(2);

System.out.println(linkedlist); // Gewünschte Output: [2, 4, 6, 7, 10, 12]

linkedlist.addLast(14);

linkedlist.add(4, 8);

System.out.println(linkedlist); // Gewünschte Output: [2, 4, 6, 7, 8, 10, 12, 14]

linkedlist.set(4, 88);

System.out.println(linkedlist); // Gewünschte Output: [2, 4, 6, 7, 88, 10, 12, 14]

linkedlist.removeLast();

System.out.println(linkedlist); // Gewünschte Output: [2, 4, 6, 7, 88, 10, 12]

linkedlist.remove(3);

System.out.println(linkedlist); // Gewünschte Output: [2, 4, 6, 88, 10, 12]

linkedlist.sort(Comparator.naturalOrder());

System.out.println(linkedlist); // Gewünschte Output: [2, 4, 6, 10, 12, 88]

linkedlist.sort(Comparator.reverseOrder());

System.out.println(linkedlist); // Gewünschte Output: [88, 12, 10, 6, 4, 2]

**Collections**

public class Main **{**

public static void main(String[] args) **{**

List<String> a = new ArrayList<String>();

a.add("Maersu");

a.add("Neeemo");

a.add("Raphij");

//alter Standard

**for**(int i = 0; i < a.size(); i++) **{**

System.out.println(a.get(i));

**}**

//mit Iterator

Iterator<String> li = a.listIterator();

**while**(li.hasNext()) **{**

System.out.println(li.next());

**}**

//foreach mit Iterator im Hintergrund

**for**(String s : a) **{**

System.out.println(s);

**}**

//foreach stream

a.stream().forEach(e -> System.out.println(e));

//Abkürzung

a.forEach(e -> System.out.println(e));

**}**

**}**

**HashSet**

private void HashSetExperiment () **{**

// HashSet<String> meinSet = new HashSet<String>();

Set<String> meinSet = new HashSet<String>();

meinSet.add("Genf");

meinSet.add("Lausanne");

meinSet.add("Freiburg");

meinSet.add("Bern");

meinSet.add("Bern");

// Kontrolle:

// Bern kommt nur einmal vor

System.out.println(meinSet);

// Ist "Lausanne" schon im Set?

System.out.println(meinSet.contains("Lausanne"));

// "Lausanne" entfernen und danach Erfolg kontrollieren

meinSet.remove("Lausanne")

System.out.println(meinSet.contains("Lausanne"));

**for** (String element : meinSet) **{**

System.out.println(element.getClass().getName() + ": " + element);

System.out.println(element.hashCode());

**}**

**}**

**Sortieren**

**Integer ArrayList**

List<Integer> integerListe = new ArrayList<Integer>();

Collections.sort(intergerListe, Collections.reverseOrder());

IntegerListe.sort((z1, z2) -> z1 - z2);

// Umgekehrte Reihenfolge

integerListe.sort((z1, z2) -> z2 - z1);

**String ArrayList**

Collections.sort(stringListe, Collections.reverseOrder());

Collections.sort(stringListe, String.CASE\_INSENSITIVE\_ORDER);

public void meineObjekte(){

List<Customer> customerListe = new ArrayList<>();

customerListe.add(new Customer("1337", "Maersu", "Gretsch"));

customerListe.add(new Customer("1234", "Neeemo", "Hallo"));

customerListe.add(new Customer("9999", "Raphij", "Schilling"));

System.out.println(customerListe);

System.out.println("-----------------");

customerListe.sort((p1,p2)->p1.getNachname().compareTo(p2.getNachname()));

for (Customer s : customerListe) {

System.out.println(s);

}

// Mit Comparator (anonyme Klasse) V1

Collections.sort(customerListe, new Comparator<Customer>() {

@Override

public int compare(Customer p1, Customer p2) {

return p1.getNachname().compareTo(p2.getNachname());

}

});

for (Customer s : customerListe) {

System.out.println(s);

}

// Mit Comparator (anonyme Klasse) V2

customerListe.sort(new Comparator<Customer>() {

@Override

public int compare(Customer p1, Customer p2) {

return p2.getNachname().compareTo(p1.getNachname());

}

});

for (Customer s : customerListe) {

System.out.println(s);

}

}

List<Benutzer> benutzerListe = new ArrayList<>();

benutzerListe.add(new Customer("1337", "Maersu", "Gretsch"));

benutzerListe.add(new Customer("1234", "Neeemo", "Hallo"));

benutzerListe.add(new Customer("9999", "Raphij", "Schilling"));

List<Benutzer> jungeBenutzer = benutzerListe.stream()

.filter(u -> u.getAlter() <= 30)

.collect(Collectors.toList());

List<Benutzer> alteSchillingBenutzer = benutzerListe.stream()

.filter(u -> u.getNachname().equals("Schilling"))

.collect(Collectors.toList());

List<Benutzer> alleGretschSortiert = benutzerListe.stream()

.filter(u -> u.getNachname().equals("Gertsch"))

.sorted((p1, p2) -> p1.getVorname().compareTo(p2.getVorname()))

.collect(Collectors.toList());

List<Benutzer> keineDuplikate = benutzerListe.stream()

.filter(u -> u.getAlter() > 30)

.sorted((p1, p2) -> p1.getVorname().compareTo(p2.getVorname()))

.map(x -> x.getNachname())

.distinct

.map(name -> name.toUpperCase())

.collect(Collectors.toList());