1 File I/O

1.1 Stream-Klassen für R/W von Binärdateien

```
try (var in = new FileInputStream("t.bin")) {
  int b = in.read();
  for (int i = 0; b >= 0; i++) { // b=-1=Dateiende
    System.out.printf(" %02X", b); // b=geles. Byte
    b = in.read(); }}
```

```
try (var out = new FileOutputStream("test.bin")) {
  while (…) { byte b = …; out.write(b); }}
```

1.2 Reader-/Writer-Klassen für R/W von Textdateien

```
try (var reader = new FileReader("test.txt")) {
// für zeilenweise: BufferedReader(new
FileReader("test.txt"))
  int value = reader.read(); // oder readLine()
  while (value >= 0) { // oder (line !=null)
  char c = (char) value;  value = reader.read(); }}
```

```
try (var writer = new FileWriter("test.txt", true))
{ writer.write("test\n"); }
```

2 Serialisierung

Serialisierung: Mechanismus, bei dem Objekte in Byte-Folgen umgewandelt werden, um sie in ein File oder eine DB zu schreiben oder übers Netzwerk zu übertragen.
Deserialisierung: Mechanismus, bei dem Objekte aus Byte-Folgen wiederhergestellt werden.
In Java sind die beiden Mechanismen als **Serializable Interface** implementiert. Es sollte jedoch nicht verwendet werden, weil es nicht interoperabel mit anderen Programmiersprachen ist, Sicherheitsprobleme aufweist und Änderungen an der Implementation einer Klasse schwierig sind.

3 Generics

Der Begriff generische Programmierung steht synonym für «parametrisierte Typen». Die Idee dahinter ist, zusätzliche Variablen für Typen einzuführen. Diese Typ-Variablen repräsentieren zum Zeitpunkt der Implementierung unbekannte Typen, erst bei der Verwendung werden diese Typ-Variablen durch konkrete Typen ersetzt. Man garantiert zur Laufzeit, dass nur bestimmte Datentypen in der Datenstruktur gespeichert sind.

3.1 Type Erasure Funktionsweise & Auswirkung

Bei der Kompilierung wird generische Programmierung entfernt, sie ist im Bytecode nicht mehr vorhanden. Enschränkungen: kein instanceof T, Type-Casts zu T sind ungeprüft, kein new T(), keine primitiven Typen wie z.B. int als Typ-Argumente ()

```
Stack<String> s;                    Stack s;
s = new Stack<String>();    →       s = new Stack();
s.push("Hi!");                      s.push("Hi!");
String x = s.pop();                 String x = (String)s.pop();
```

```
public static <T extends Comparable<T>> int
findFirstGreaterThan(T[] a, T b) {}
```

↓

```
public static int findFirstGreaterThan(Comparable[]
a, Comparable b) {}
```

3.2 Generic Classes, Interfaces & Methods erstellen & verw.

```
class ReverseMap<L, R> extends Map<L, R> {
  List<L> leftList; List<R> rightList;
  void put(L left, R right) {} }
<T extends Comparable<T>> T findLargest(Iterable<T>
iterable){
  T largest = iterable.iterator().next();
  for(T t : iterable){ if (largest.compareTo(t)<1) {
    largest = t; }} return largest; }
```

3.3 Untersch. Fälle generischer Varianz erkl./einsetzen

	Typ	Kompatible Typ-Argumente	Lesen	Schreiben
Invarianz	C<T>	T	✓	✓
Kovarianz	C<? extends T>	T und Subtypen	✓	✗
Kontravarianz	C<? super T>	T und Basistypen	✗	✓
Bivarianz	C<?>	Alle	✗	✗

Schreiben = new Class()

```
<T> void move(Stack<? extends T> from, Stack<? super
T> to) {
  while (!from.isEmpty()) { to.push(from.pop()); }}
```

4 Abstract Data Types

Ein abstrakter Datentyp (ADT) ist eine Abstraktion einer konkreten Datenstruktur. Das «Was» wird beschrieben, aber nicht das «Wie». Es ist als Interface realisiert.

Elemente: Attribute (z.B. Einträge in einer Liste), Operationen auf den Attributen (z.B. get(), put(),…), Ausnahmen und Fehler (Welche Exception wird wann geworfen?)

Beispiel: Der abstrakte Datentyp Stack beschreibt eine Reihe von Funktionen (z.B. pop(), push()). Diese können auf unterschiedliche Art und Weise (z.B. mit Array oder einer Liste) implementiert werden.

4.1 Array

Ein Array in Java ist ein Container, welcher eine feste Anzahl von Werten eines einzelnen Typs enthält. Bei der Deklaration wird von Beginn an ein konkreter Datentyp und die Länge festgelegt, beides kann später nicht mehr verändert werden. Jedes Element erhält einen eindeutigen Index, das ist eine positive Ganzzahl inkl. 0. Lesen=O(1), Add=O(n)
Array: Schneller wahlfreier Zugriff, RAM-effizient, feste Grösse, einfügen ist unter Umständen teuer
ArrayList: Schneller wahlfreier Zugriff, dynamische Grösse, einfügen ist u.U. teuer, weniger RAM-effizient als Arrays

4.2 List

Elem. linear in Reihe, jedes Elem. hat genau 1 Vorgänger

4.2.1 Linked List

Das ist eine dynamische Datenstruktur, in der Datenelemente geordnet gespeichert sind. Es gibt keine maximale Anzahl Elemente und es muss nicht zwingend ein konkreter Datentyp angegeben werden bei der Erstellung. Die Liste besteht aus Knoten, welche wiederum aus Daten und einem Zeiger auf das nächste Element bestehen. Eine doppelt verkettete Liste hat zusätzlich einen Zeiger aufs vorherige Element. Bei der einfachen Liste ist irgendwo der head (erstes Element) definiert, bei der doppelt veketteten zusätzlich noch der trailer (letztes Element). Lesen=O(n), Add am Anfang/Ende=O(1)
LinkedList: dynam. Grösse, einfügen/löschen am Anfang ist schnell, langsamer Zugriff, weniger RAM-effizient als Arrays
DoublyLinkedList: dynamische Grösse, einfügen und löschen am Anfang und Ende ist schnell, Rückwärtsdurchlauf möglich, langsamer Zugriff, weniger RAM-effizient als Arrays.

```
void addOnBeginning(T data) {
  Node<T> newNode = new Node<>(data);
  newNode.next = head; head = newNode; }
void addLast(T data) {
  Node<T> newNode = new Node<>(data);
  if (head == null) { head = newNode; }
  else { Node<T> current = head;
    while (current.next != null) {
      current = current.next; }
    current.next = newNode; }}
```

```
boolean remove(T data) {
  if (head == null) { return false; }
  if (head.data.equals(data)) {
    head = head.next; return true; }
  Node<T> current = head;
  while (current.next != null) {
    if (current.next.data.equals(data)) {
      current.next = current.next.next;
      return true; }
    current = current.next; } return false; }
```

4.2.2 Doubly Linked List

```
class Node<T> {
  T data; Node<T> next; Node<T> prev; }
class DoublyLinkedList<T> {
  Node<T> head; Node<T> trailer;
  void addFirst(T item) {
    if (head == null) {
      head = new Node<>(item); trailer = head; }
    else { Node<T> newNode = new Node<>(item);
      newNode.next = head; head.prev = newNode;
      head = newNode; }}
  void addLast(T item) {
    if(trailer == null) {
      trailer = new Node<>(item); head = trailer; }
    else { Node<T> newNode = new Node<>(item);
      newNode.prev = trailer; trailer.next =
      newNode; trailer = newNode; }}
  void removeFirst() {
    if (head == null) { throw new NoSuchElemExc(); }
    else if (head == trailer) {
      head = null; trailer = null; }
    else { head.next.prev=null; head = head.next; }}
```

4.3 STACK

Collection von Elementen mit zwei Operationen: Push um ein Element hinzuzufügen, Pop um das zuletzt hinzugefügte Element zu entfernen, LIFO-Prinzip (Last In First Out)

```
class ListStack<T> implements Stack<T> {
  Node<T> top; int size;
  public ListStack(int max) { this.max = max; }
  int size() { return size; }
  boolean isEmpty() { return size == 0; }
  T top() { return top.getElement(); }
  void push(T element) {
    Node<T> oldTop = this.top;
    this.top = new Node<>(element);
    this.top.setNext(oldTop); size++; }
  T pop() {
    T oldTop = top.getElement();
    this.top = top.getNext();
    size--; return oldTop; }}
```

4.3.1 Algorithmus zur Span-Berechnung mit Stack

```
Algorithm spans2(X, n)
  S = new array of n integers
  A = new empty stack
  for i = 0 to n − 1 do
    while (¬A.isEmpty() ∧ X[A.top()] ≤ X[i]) do
      A.pop()
    if A.isEmpty() then
      S[i] = i + 1
    else
      S[i] = i - A.top()
    A.push(i)
  return S
```

4.4 QUEUE

ähnlich wie der Stack, jedoch FIFO-Prinzip (First In First Out)

```
class ArrayQueue<T> implements Queue<T> {
  private T[] array;
  private int size;
  private int front;
  public ArrayQueue() { array=(T[]) new Object[2]; }
  int size() { return size; }
  boolean isEmpty() { return size == 0; }
  T front() throws EmptyQEx {
    if (isEmpty()) { throw new EmptyQEx("q empt"); }
    return array[front]; }
  void enqueue(T element) {
    if (size == array.length) { enlargeArray(); }
    array[(front + size) % array.length] = element;
    size++; }
  T dequeue() throws EmptyQEx {
    if (isEmpty()) { throw new EmptyQEx("q empt"); }
    T oldFront = array[front];
    array[front] = null;
    front = (front + 1) % array.length;
    size--;
    if ((size>=2)&&(size==array.length/2)) {
reduceArray(); }
    return oldFront; }
  void enlargeArray() {
    T[] prev = array;
    array = (T[]) new Object[array.length * 2];
    copyToArray(prev);
    front = 0; }
  void reduceArray() {
    T[] prev = array;
    if (prev.length % 2 == 0) {
      array = (T[]) new Object[prev.length / 2];
    } else {
      array=(T[]) new Object[prev.length / 2 + 1]; }
    copyToArray(prev);
    front = 0; }
  void copyToArray(T[] src) {
    int rear = (front + size) % src.length;
    if (front < rear) {
      System.arraycopy(src, front, array, 0, size);
    } else {
      int frontToEndLength = src.length - front;
      System.arraycopy(src, front, array, 0,
frontToEndLength);
      System.arraycopy(src, 0, array,
frontToEndLength, size - frontToEndLength); }}
```

4.4.1 Ringbuffer implementieren

```
class CircularBuffer<E> {
  int capacity = 0; int size = 0;
  int head = 0; int tail = -1;
  E[] array;
  public CircularBuffer(int capacity) {
    this.capacity = capacity;
    array = new E[capacity]; }
  void add(E element) throws Exception {
    int index = (tail + 1) % capacity;
    size++;
    if (size == capacity) { throw new
Exception("Buffer Overflow"); }
    array[index] = element;
    tail++; }
  E get() throws Exception {
    if (size == 0) { throw new Exception("Empty
Buffer"); }
    int index = head % capacity;
    E element = array[index];
    head++;
    size--;
    return element; }
  E peek() throws Exception {
    if (size == 0) { throw new Exception("Empty
Buffer"); }
    int index = head % capacity;
    E element = array[index];
    return element; }
  boolean isEmpty() { return size == 0; }
  int size() { return size; }
```

4.4.2 Priority queue

In eine Priority Queue können beliebige Einträge mit Key (Priorität) und Value eingefügt werden. Beim Holen von Elementen aus der Queue wird jeweils der Eintrag mit der niedrigsten Priorität zurückgegeben. Die Einträge können entweder beim Einfügen sortiert werden oder erst bei der Ausgabe.
Erweiterung der Priority Queue mit remove(), replaceValue() und replaceKey(). Ein Beispiel einer Adaptable Priority Queue ist ein Trading System, bei dem es zwei Queues für Buy und Sell gibt, bei den Entries gilt K=Preis, V=Anzahl Aktien. Ein Buy-Auftrag wird ausgeführt, wenn $k_s < k_b$ & $a_s > a_b$ und ein Sell-Auftrag wenn $k_s < k_b$ & $a_s > a_b$. Die User können noch Anpassungen an ihren Orders machen.

```
class PriorityQueue<K, V> implements IPQ<K, V> {
  final LinkedList<Entry<K, V>> list;
  final Comparator<K> comparator;
  public PriorityQueue(Comparator<K> comparator) {
    this.comparator = comparator;
    list = new LinkedList<>(); }
  int size() { return list.size(); }
  boolean isEmpty() { return size() == 0; }
```

```
  Entry<K, V> min() throws Exception {
    if(size()==0){ throw new Exception("q empty"); }
    return list.first(); }
  Entry<K,V> insert(K key, V value) {
    Entry<K,V> newest = new PQEntry<>(key, value);
    Position<Entry<K,V>> walk = list.last();
    while(walk!=null && compare(newest,
walk.getElement()) < 0)
    walk = list.before(walk);
    if (walk == null) {list.addFirst(newest);
    } else { list.addAfter(walk, newest); }
    return newest; }
  Entry<K,V> removeMin() {
    if (list.isEmpty()) return null;
    return list.remove(list.first()); }
  void print() {
    for (var element : list) {
      System.out.println(element.toString()); }}}
```

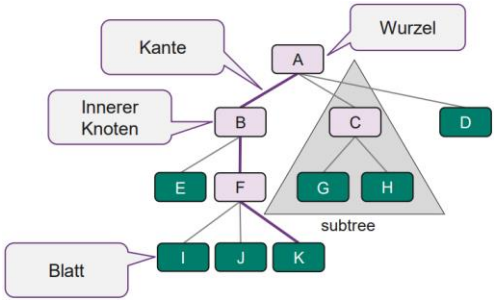
4.4.2.1 *Adaptable Priority Queue*

```
class APQ<K,V> implements IAPQ<K,V> {
  Entry<K, V> root;
  final Comparator<K> comparator;
  int count = 0;
  // Methoden wie in Priority Queue
  Entry<K, V> remove(Entry<K, V> e) {
    Entry<K, V> current = root;
    Entry<K, V> previous = null;
    if (current != null && root.equals(e)) {
      root = root.getNext();
      return current; }
    while(current!=null && !current.equals(e)) {
      previous = current;
      current = current.getNext(); }
    if (current == null) { return null; }
    previous.setNext(current.getNext());
    return current; }
  boolean replaceValue(Entry<K, V> e, V value) {
    Entry<K, V> current = root;
    while(current!=null && !current.equals(e)) {
      current = current.getNext(); }
    if (current == null) { return false; }
    current.setValue(value);
    return true; }
  boolean replaceKey(Entry<K, V> entry, K key) {
    Entry<K, V> replacedEntry = remove(entry);
    if (replacedEntry == null) { return false; }
    replacedEntry.setKey(key);
    replacedEntry.setNext(null);
    sortNewNode(replacedEntry);
    return true; }}
```

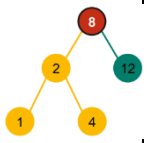
4.5 Trees

4.5.1 Merkmale/Komponenten Baum

Umgekehrter Baum, zweidimensionale Datenstruktur, stellt hierarchische Beziehungen dar, Komponenten:



Vorgängerknoten: Eltern, Grosseltern, …
Tiefe eines Knotens: Anzahl Vorgänger
Höhe des Baums: Maximale Tiefe der Knoten
Geschwister: Knoten mit selben Eltern
Binärer Baum: Knoten haben immer höchstens 2 Kinder, Kinder sind von links nach rechts geordnet
Binärer Suchbaum (BST): jeder Knoten hat Key, Keys des linken Teilbaums sind kleiner als die Wurzel, Keys des rechten Teilbaums sind grösser als die Wurzel



4.5.2 Tiefe/Höhe berechnen

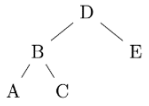
```
Algorithmus depth(T, v):
  if v ist Wurzel von T then
    return 0
  else
    return 1 + depth(T, v.parent())
```

```
Algorithmus height(T, v):
  h = 0
  for all children w of v
    h = max(h, 1 + height(T, w))
  return h
```

4.5.3 Anwendungen Baum beschreiben

- Darstellung arithmetischer Ausdrücke
- Heapsort
- HTML Document Object Model
- Binärsuche
- Verzeichnisbaum Filesystem
- Inhaltsverzeichnis

4.5.4 Baum traversieren



Pre-Order (W-L-R): D-B-A-C-E
Post-Order (L-R-W): A-C-B-E-D
In-Order (L-W-R): A-B-C-D-E
Breadth-First: D-B-E-A-C

4.6 Set

Set: Sammlung gleichartiger Objekte, keine Duplikate, keine Reihenfolge, entspricht Map ohne Values, Methoden: add(e), remove(e), contains(e), size(), isEmpty()
Multiset: Set mit erlaubten Duplikaten, Methoden: get(e), contains(e), count(e), remove(e), remove(e,n)

4.7 Map

Map: speichert Key-Value Paare, Schlüssel sind einzigartig, genau ein Wert pro Schlüssel, Methoden: get(k), put(k,v), remove(k), size(), isEmpty(), keySet(), values(), entrySet()
Multimap: Mehrere Values pro Key, Methoden: get(k), put(k,v), remove(k,v), removeAll(k)

5 Algorithmen

Ein Algorithmus hat ein Problem als Input und eine oder mehrere Lösungen als Output. Er muss endlich, deterministisch (immer das gleiche Ergebnis bei gleicher Eingabe) und allgemein sein. Er verwendet ausführbare und elementare Einzelschritte.

5.1 Pseudocode Beispiel

```
Algorithm arrayMax(A, n)
  Input array A of n integers
  Output maximum element of A
  currentMax = A[0]
  for i = 1 to n - 1 do
    if A[i] > currentMax then
      currentMax = A[i]
  return currentMax
```

5.2 Empirische Analyse

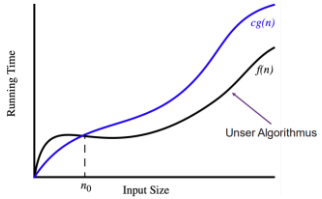
Bei der empirischen Analyse wird der Algorithmus implementiert, mit unterschiedlichen Eingaben ausgeführt, Laufzeiten gemessen und die Ergebnisse verglichen.

```
import org.apache.commons.lang3.time.StopWatch;
StopWatch s = new StopWatch(); s.reset(); s.start();
/* Algorithmus */ s.stop();
```

Nachteile: Eingaben beeinfl. Ergebnis, Störvariablen wie HW/SW/Systembelastung, Algorithmus muss impl. sein

5.3 Big-O Notation

Simplifizierte Analyse der Effizienz eines Algorithmus basierend auf der Inputgrösse n, es ist machine-independent und zählt die primitiven Oerationen im Worst-Case, es wird die Zeit und Memory-Usage berechnet, Zugehörigkeit zu einer Komplexitätsklasse gezeigt



$f(n) = O(g(n))$ *falls* $f(n) \leq c \cdot g(n)$ für $n \geq n_0$

Konstanten werden ignoriert: $5n \rightarrow O(n)$

Terme werden Termen höherer Ordung überschrieben:
$O(1) < O(\log(n)) < O(n) < O(n \log(n)) < O(n^2) < O(n^3) < O(2^n)$

$7n-2$ ist $O(n)$

Gesucht: $c > 0$ und $n_0 \geq 1$, so dass $7n-2 \leq c*n$ für $n \geq n_0$

Lösung: $c = 7$ und $n_0 = 1$

5.3.1 Primitive Operationen zählen

Algorithm arrayMax(A, n)	# Operationen	
currentMax = A[0]	1 Indexierung + 1 Zuweisung:	2
for i = 1 to n − 1 do	1 Zuweisung + n (Subtraktion + Test):	$1+2n$
if A[i] > currentMax then	(Indexierungen + Test) $(n-1)$	$2(n-1)$
currentMax = A[i]	(Indexierungen + Zuweisung) $(n-1)$	$0 \mid 2(n-1)$
increment i	(Inkrement + Zuweisung) $(n-1)$	$2(n-1)$
return currentMax	1 Verlassen der Methode	1

Worst Case: $2+(1+2n)+2(n-1)+2(n-1)+2(n-1)+1 = 8n-2$

Best Case: $2+(1+2n)+2(n-1)+0+2(n-1)+1 = 6n$

5.4 PARADIGMEN

5.4.1 Greedy

Dieser Algorithmus trifft immer die aktuell beste Entscheidung. Dadurch ist er sehr schnell, liefert aber möglicherweise nicht die optimale Lösung. Zum Beispiel nimmt er bei der Zusammenstellung von Münzen für die Rückgabe immer die grösste Münze, die in den Restbetrag passt, möglicherweise ist die Anzahl Münzen dann aber nicht minimal am Schluss.

```
static HashSet<String>
calculateSolution(HashSet<String> statNeed,
HashMap<String, HashSet<String>> availStat) {
  var solution = new HashSet<String>();
  while(!statNeed.isEmpty()) {
    String bestStation = "";
    var bestProbStatCov = new HashSet<String>();
    for (String probStat, availStat.keySet()) {
     var probStatCov = new HashSet<>(statNeed);
     probStatCov.retainAll(availStat.get(probStat));
     if (probStatCov.size()>bestProbStatCov.size()){
      bestStation = probStat;
      bestProbStatCov = probStatCov; } }
   statNeed.removeAll(bestProbStatCov);
   solution.add(bestStation); } return solution; }
```

5.4.2 Divide-and-Conquer/Teile-und-Herrsche

Verfahren wird auch Binärsuche genannt und eignet sich zur Suche in einem sortierten Array.

```
<T extends Comparable<T>> boolean searBin(List<T>
data, T target, int low, int high) {
  if (low > high) { return false; }
  int pivot = low + ((high - low) / 2);
  if (target.equals(data.get(pivot))){return true; }
  else if (target.compareTo(data.get(pivot))<0) {
    return searBin(data,target,low,pivot-1); }
  else{return searBin(data,target,pivot+1,high); }}
```

5.4.3 Backtracking

Beim Backtracking wird ein Weg gegangen, bis es nicht mehr geht. Wenn es nicht mehr geht, geht man zurück bis zur letzten Verzweigung und wählt dort eine andere Möglichkeit. Ein Beispiel eines Backtracking-Problems ist das Damenproblem, dabei müssen k Damen auf einem k*k Schachbrett platziert werden. Dabei darf keine Dame von einer anderen Dame bedroht werden (Damen können alle Figuren in ihrer Reihe, Kolonne und Diagonale bedrohen).

5.4.3.1 Knight Tour

```
bol knightTour(int[][] visited,int x,int y,int pos){
  visited[x][y] = pos; // Position markieren
  if (pos >= N * N) { return true; } // Abbruchbed.
  for (int k = 0; k < 8; k++) { // Alle möglichen
    int newX = x + row[k]; // Züge probieren und
    int newY = y + col[k]; // neue Koordinate berec.
  -  if(isValid(newX,newY)&&
visited[newX][newY]==0){
      if (knightTour(visited,newX,newY,pos+1)) {
        return true;  }}} // Prüfen, ob Feld
innerhalb Brett, nicht besucht & rekursiv positiv
  visited[x][y] = 0; return false; } // Backtrack
```

5.4.3.2 Sudoku

```
boolean solve(int row, int col) {
  if (row == 8 && col == 9) return true;
  if (col == 9) { row++; col = 0; }
  if (sudokuArray [row][col] != 0) {
    return solve(row, col + 1) }
  else {
    for (int num = 1; num < 10; num++) {
      if (checkRow(row, num) && checkCol(col, num)
      && checkBox(row, col, num)) {
        sudokuArray [row][col] = num;
        if (solve(row, col + 1)) return true; }}
  sudokuArray[row][col] = 0;
  return false; }
```

5.4.3.3 Maus-Labyrinth

```
public class RatMazeProblem {
  static int R;
  static int C;
  void displaySolution(int result[][]) {
    System.out.println("The resultant matrix is: ");
    for (int r = 0; r < R; r++) {
      for (int c = 0; c < C; c++)  {
        System.out.print(" " + result[r][c] + " ");}
      System.out.println();  }}
  boolean isValid(int maze[][], int r, int c) {
    return (r >= 0 && r < R && c >= 0 && c < C &&
maze[r][c] == 0); }
  boolean findPathMaze(int maze[][])  {
    int maz[][] = new int[R][C];
    for(int r = 0; r < R; r++) {
      for(int c = 0; c < C; c++) {
        maz[r][c] = 1; }}
    if (solveUtil(maze, 0, 0, maz) == false){
      System.out.print("Path doesn't exist");
      return false; }
    displaySolution(maz);
    return true; }
  boolean solveUtil(int maze[][],int r,int c,int
res[][]) {
    if (r == R - 1 && c== C - 1 && maze[r][c] == 0){
      res[r][c] = 0;
      return true; }
    if (isValid(maze, r, c) == true) {
      if (res[r][c] == 0)  { return false; }
      res[r][c] = 0;
      if(solveUtil(maze,r+1,c,res)) { return true; }
      if(solveUtil(maze,r,c+1,res)) { return true; }
      if(solveUtil(maze,r-1,c,res)) { return true; }
      if(solveUtil(maze,r,c-1,res)) { return true; }
      res[r][c] = 1; return false; }
    return false; }
  public static void main(String argvs[])  {
    RatMazeProblem r = new RatMazeProblem();
    int maze[][] = { { 0, 1, 1, 1 },
                     { 0, 0, 1, 0 },
                     { 1, 0, 1, 1 },
                     { 0, 0, 0, 0 } };
    R = maze.length; C = maze[0].length;
    r.findPathMaze(maze); }}
```

5.4.4 Dynamische Programmierung

Bei der dynamischen Programmierung fängt man klein an und berechnet iterativ die Lösungen für immer grössere Probleme aus den Lösungen der kleineren Probleme. Die Lösungen der kleineren Probleme werden dabei immer so lange gespeichert, wie sie noch für Berechnung grösserer Probleme benötigt werden. Beispiel Fibonacci-Folge: (folg. Zahl ergibt sich durch Addition der beiden vorherigen):

```
public static long dynamisch(int n) {
  long[] f = new long[n + 2];
  f[0] = 0; f[1] = 1;
  for (int i = 2; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2]; } return f[n]; }
```

5.4.5 Rekursion

Bei einer rekursiven Methode wird die eigene Methode wieder aufgerufen und es gibt eine Abbruchkondition.

```
int recursiveFactorial(int n) {
  if (n == 0) { return 1; }
  else { return n * recursiveFactorial(n - 1); }}
```

5.4.5.1 Algo. mit linearer Rekursion implementieren

```
int[] reverseArray(int[] a, int i, int j) {
  if (i < j) { int temp = a[j];
    a[j] = a[i]; a[i] = temp;
    reverseArray(a, i + 1, j - 1); } return a; }
```

5.4.5.2 Algo. mit binärer/Mehrfacher Rekursion impl.

Funktion wird 2x (binär) oder mehrfach aufgerufen innerhalb der Funktion

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

5.4.5.3 *Endrekursive Algo. erkennen & in iterativ umw.*

Bei der Endrekursion wird das Zwischenergebnis beim Methodenaufruf als Argument mitgegeben.

```
int recursiveFactorial(int x, int total) {
  if (x == 0) { return total; }
  else { return recursiveFactorial(x-1, total*x); }}

int iterativeFactorial(int x) {
  int total = 1;
  for (int i = x; i > 0; i--) { total = total * i; }
  return total; }
```

5.5 Sortieralgorithmen

5.5.1 Insertionsort Funktionsweise/Laufzeit/Impl.

Beim Insertionsort geht man bei jedem Schritt eine Position nach vorne. Die aktuelle Position wird so lange nach vorne geschoben, bis sie einsortiert ist. Gut für teilweise sortierte Arrays. Best case $O(n)$, Worst case $O(n^2)$

```
<T extends Comparable<T>> void sort(T[] data) {
  for (int i = 1; i < data.length; i++) {
    T curr = data[i]; int j = i;
    for(;(j>0)&&(data[j-1].compareTo(curr)>0);j--) {
      data[j] = data[j - 1]; }
    data[j] = curr; }}
```

5.5.2 Selectionsort Funktionsweise/Laufzeit/Impl.

Beim Selectionsort geht man bei jedem Schritt eine Position nach vorne. Dann wird das ganze Array hinter der Position nach dem kleinsten Element durchsucht und dieses getauscht mit der aktuellen Position. Wenige Bewegungen im Array. Best & Worst Case $O(n^2)$

```
static void sort(int[] arr) {
  int n = arr.length;
  for (int i = 0; i < n - 1; i++) {
    int minimum = i;
    for (int j = i + 1; j < n; j++) {
      if (arr[j] < arr[minimum]) { minimum = j; }}
    swap(arr, i, minimum); }}
```

5.5.3 Shellsort Funktionsweise/Laufzeit/Implementation

Generalisierte Version des Insertion Sorts. Zuerst werden weit auseinanderliegende Werte verglichen und swapped wenn nötig. Die Werte liegen dann laufend näher. Best Case $O(n \cdot log(n))$ Worst Case $O(n \cdot log^2(n))$

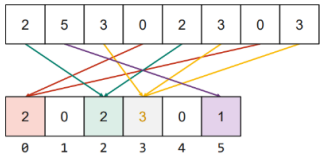
```
void sort(int[] arr) { int h = 1;
 while (h < arr.length/3) { h = 3 * h + 1; }
 while (h >= 1) {
  for (int i=h; i<arr.length; i++) {
   for (int j=i; j>=h && arr[j]<arr[j-h]; j=j-h) {
    swap(arr, j, j - h); }}
  h /= 3; }}
```

5.5.4 Bubblesort Funktionsweise/Laufzeit/Impl.

Gehe von links nach rechts durch, vergleiche Nachbaren & tausche wenn nötig. Best Case $O(n)$ Worst Case $O(n^2)$

```
void sort(int[] arr) {
 boolean isSwapped;
 for (int i = 0; i < arr.length - 1; i++) {
  isSwapped = false;
  for (int j = 0; j < arr.length - i - 1; j++) {
   if (arr[j] > arr[j + 1]) {
    int temp = arr[j]; arr[j] = arr[j + 1];
    arr[j + 1] = temp; isSwapped = true; }}
  if (!isSwapped) { break; }}}
```

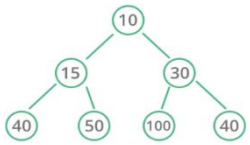
5.5.5 Counting Sort Funktionsweise/Laufzeit/Impl.



Array mit Länge $höchster\ Wert + 1$ erstellen, Anzahl Elemente im Array notieren, Array mit originaler Grösse aufgrund Hilfsarray füllen, Best/Worst Case $O(n + grösstes\ Elem. - kleinstes\ Elem.$

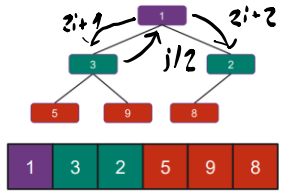
```
void sort(int arr[]) {
 int[] output = new int[arr.length];
 int max = arr[0];
 for (int i = 1; i < arr.length; i++) {
  if (arr[i] > max) { max = arr[i]; }}
 int[] count = new int[max + 1];
 for (int i = 0; i < arr.length; i++) {
  count[arr[i]]++; }
 for (int i = 1; i <= max; i++) {
  count[i] += count[i - 1]; }
 for (int i = arr.length - 1; i >= 0; i--) {
  output[count[arr[i]] - 1] = arr[i];
  count[arr[i]]--; }
 for (int i = 0; i < arr.length; i++) {
  arr[i] = output[i]; }}
```

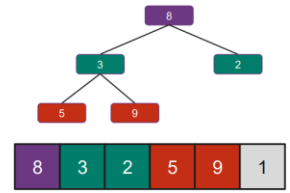
5.5.6 Binärer Heap / Heapsort



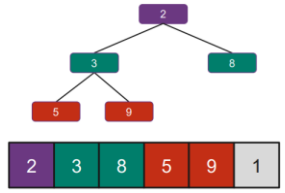
Beim binären Heap muss die Wurzel immer den kleinsten Wert von Wurzel, links und rechts enthalten. Links/rechts muss nicht sortiert sein.

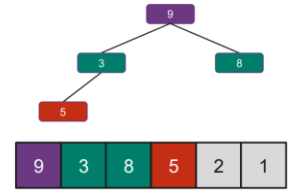
Ausgangssituat. für Heapsort ist ein Heap, man weiss, dass 1 die kleinste Zahl ist und tauscht sie mit dem Ende des Arrays:





Heap ist nicht mehr erfüllt & muss wiederhergestellt werden, 2 ist die kleinste & wird mit dem zweitletzten Index getauscht





Das wird nun immer wiederholt, bis das Array sortiert ist

```
void heapSort(Comparable[] arrayToSort) {
  heapifyMe(arrayToSort);
  for (int i = arrayToSort.length - 1; i > 0; i--) {
    swap(arrayToSort, 0, i);
    percolate(arrayToSort, 0, i - 1); }}

void percolate(Comparable[] arr, int startIndex, int
last) {
 int i = startIndex;
 while(hasLeftChild(i, last)) {
  int leftChild = getLeftChild(i);
  int rightChild = getRightChild(i);
  int exchangeWith = 0;
  if(arr[i].compareTo(arr[leftChild]) > 0) {
   exchangeWith = leftChild; }
  if(rightChild <= last &&
  arr[leftChild].compareTo(arr[rightChild]) > 0) {
   exchangeWith = rightChild; }
  if (exchangeWith == 0 ||
  arr[i].compareTo(arr[exchangeWith]) <= 0) {
   break; }
  swap(arr, i, exchangeWith);
  i = exchangeWith; }}
```

6 Design Patterns

Wiederverwendbare Lösungen für wiederkehrende Probleme, Vorlagen, die in konkreten Problemen angewandt werden, Pattern beschreibt Problem, abstrakte in neuen Kontexten anwendbare Lös., Anwendung & Konsequenzen

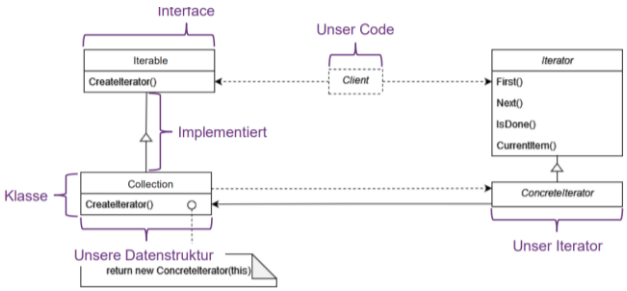
Erzeugungsmuster: abstrahieren Instanziierung, z.B. Factory, Singleton

Strukturmuster: Zusammensetzung von Klassen & Objekten zu grösseren Strukturen, z.B. Adapter, Fassade

Verhaltensmuster: Algorithmen und Verteilung von Verantwortung zwischen Objekten, z.B. Iterator, Visitor

6.1 Iterator Pattern

Ziel: Auf Elemente eines Aggregatobjekts sequentiell zugreifen, ohne zugrunde liegende Repräsentation offenzulegen.



Zur Implementation in einer ADT muss eine (innere) Klasse für die ADT erstellt werden, welche das Java Iterator Interface implementiert.

Lazy-Iterator: Iteration auf Original-Datenstruktur, O(1), Probleme bei unerwarteter Modifikation während Iteration

Snapshot-Iterator: Original-Datenstruktur beibehalten, O(n), es wird eine Kopie der Datenstruktur erstellt vor der Iteration

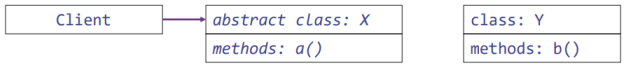
6.2 Visitor Pattern

Ziel: Algorithmen von Datenstrukturen auf denen Sie operieren trennen. Einfache Erweiterbarkeit, wenn neue Algorithmen hinzugefügt werden. Grosser Änderungsbedarf wenn neue Klassen ergänzt werden.

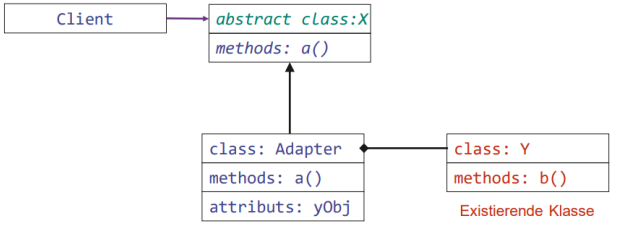
```
class Book implements ItemElement {
  int accept(ShopCartVisito visitor) {
    return visitor.visit(this); }}

class Fruit implements ItemElement {
  int accept(ShopCartVisito visitor) {
    return visitor.visit(this); }}

class ShopCartVisitoImp implements ShopCartVisito {
  int visit(Book book) { ... }
  int visit(Fruit fruit) { ... }}

class ShoppingCartClient {
  static int calculatePrice(ItemElement[] items) {
    ShopCartVisito v = new ShopCartVisitoImp();
    int sum=0;
    for(ItemElement i : items){sum += i.accept(v); }
    return sum; }}
```

6.3 Adapter/Wrapper Pattern

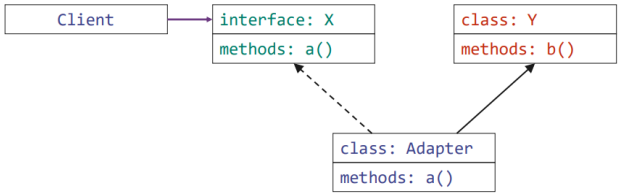
Ziel: Existierende Klasse verwenden, deren Schnittstelle nicht der benötigten Schnittstelle entspricht.



Objekt-Adapter:



Klassen-Adapter:



6.4 Template-Method Pattern

Ein Algorithmus wird als Skelett mit Defaults für Operationen (abstract class) definiert und die Child-Klassen implementieren die Details dazu und können optional Defaults überschreiben. Dadurch können Teile des Algorithmus verändert werden, ohne dass die ganze Struktur geändert wird.

7 Hashing

Eine Hash-Funktion bildet Schlüssel auf Indexwerte im Intervall [0; N-1] ab und bestimmt die Position des Elements im Feld. Die hashCode() Methode ist in Object impl. und existiert somit bereits für alle Klassen. x.equals(y) → x.hashCode()==y.hashCode()(umgekehrt nicht zwingend)

Kompressionsfunktionen können auf Hashfunktionen ausgeführt werden, um möglichst zufällig verteilte Hashes in ein fixes Intervall zu transformieren.

Das Ziel ist, mit konstantem Aufwand in einer Collection zu suchen. Die Hashfunktion muss konsistent sein, also für den gleichen Input den gleichen Hash ausgeben.

7.1 Arten

- **Divisionsrestverfahren**: $mod\ (Size\ Zielintervall)$ auf zu hashende Zahl ausführen
- **Integer Cast**: Speicheradresse des Objekts auslesen, streut gut
  `byte[] b=key.getBytes(StandardCharsets.UTF_16); ByteBuffer wrapped = ByteBuffer.wrap(b); return wrapped.getInt();`
- **Komponentensumme**: Schlüssel in Komponenten fester Länge unterteilen, Kompontenten summieren, Overflow ignorieren

```
int hash = 0;
for (int i = 0; i < s.length(); i++) {
  hash += s.charAt(i); }
```

- **Polynom-Akkumulation**: ähnlich wie Komponentensumme, Komponenten werden unterschiedlich gewichtet, Default bei Strings
  `s[0] * 31^(n-1) + s[1] * 31^(n-2) + ... + s[n-1]`

7.2 Hashsets/Hashmaps

Man möchte möglichst schnell einen Eintrag in einer Hashmap finden mithilfe von Hashing.

7.3 Kollisionsbehandlung

Bei einer Kollision haben zwei unterschiedliche Schlüssel den gleichen Hash. Es gibt zwei Arten zur Behandlung:

7.3.1 Geschlossene Adressierung

Behälter sind verkettete Listen, Platz nicht begrenzt, prinzipiell keine Überläufer, zusätzliche Datenstruktur und Speicherbedarf, $Average\ O(\frac{n=Einträge\ der\ Tabelle}{N=Buckets\ in\ der\ Tabelle})$

7.3.2 Offene Adressierung

Für Überläufer wird in anderen Behältern Platz gesucht, Sondierungsfolge bestimmt Weg zum Speichern und Wiederauffinden der Überläufer, Lineare Sondierung: immer 1 nach hinten/vorne im Array wandern, bis ein Index frei ist:

```
int findAvailableSlot(int indexInHashTable, K key) {
  int probedI = indexInHashTable;
  do {
    if (isAvailable(probedI)) { return probedI; }
    else if (table[probedI].getKey().equals(key)) {
      return probedI; }
    probedI= (probedI + 1) % capacity;
    } while (probedI != indexInHashTable);
    return -1; }

V remove(K key) {
  int i=Math.abs(key.hashCode() % capacity);
  int indexInHashMap=probeForDeletion(i, key);
  if (indexInHashMap == -1) { return null; }
  V answer = table[indexInHashMap].getValue();
  table[indexInHashMap] = DELETED;
  return answer; }
```

Es gibt auch noch die Option quadratische Sondierung (+1, +4, +9, …), doppeltes Hashing oder dynamisches Hashing.

Beim dynamischen Hashing werden für den Hash nur eine gewisse Anzahl Bits verwendet, die notwendig ist, um Collisions zu vermeiden. Objekte dürfen nicht physisch gelöscht werden, sondern müssen als gelöscht markiert werden. Dies würde die Sondierungsfolge anderer Objekte unterbrechen.