1 Input und Output

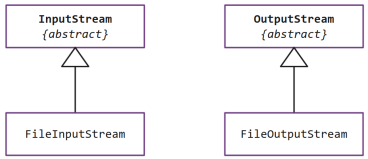


Abbildung 1: Klassenhierarchie von Input und Output

1.1 Input

1.1.1 File-Reader

```
try (var reader = new FileReader("quotes.txt")) {
  int value = reader.read();
  while (value >= 0) {
    char c = (char) value;
    // use character
    value = reader.read();
  }
}
```

```
new FileReader(f);
// ist äquivalent zu
new InputStreamReader(new FileInputStream(f));
```

1.1.2 Zeilenweises Lesen

```
try (var reader = new BufferedReader(new FileReader("quotes.txt")) {
  String line = reader.readLine();
  while (line != null) {
    System.out.println(line);
    line = reader.readLine();
  }
}
```

Info: `FileReader` liest einzelne Zeichen, `BufferedReader` liest ganze Zeilen.

1.2 Output

1.2.1 File-Writer

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

1.3 Zusammenfassung

- Byte-Stream: Byteweises Lesen von Dateien
- FileInputStream, FileOutputStream
- Character-Stream: Zeichenweises Lesen von Dateien (UTF-8)
- FileReader, FileWriter

2 Serialisierung

Das `Serializable`-Interface implementieren (Marker-Interface). Ohne Marker-Interface wird eine `NotSerializableException` geworfen. Jedes Feld, das serialisiert werden soll, muss ebenfalls `Serializable` implementieren (Transitive Serialisierung).

```
class Person implements Serializable {
  private static final long serialVersionUID = 1L;
  private String firstName;
  private String lastName;
  // …
}
```

Das kann dann vom ObjectOutputStream verwendet werden, um Data Binär zu serialisieren:

```
try (var stream = new ObjectOutputStream(new
FileOutputStream("serial.bin"))) {
  stream.writeObject(person);
}
```

Um ein Objekt aus einem Bytestrom zu deserialisieren, wird der ObjectInputStream verwendet:

```
try (var stream = new ObjectInputStream(
  new FileInputStream("serial.bin"))) {
  Person p = (Person) stream.readObject();
  // …
}
```

2.1 Serialisierung mit Jackson

```
Employee e = new Employee(1, "Frieder Loch");
String jsonString = mapper.writeValueAsString(e);
var writer = new PrintWriter(FILE_PATH);
writer.println(jsonString);
writer.close();
```

Output:

```
{"id":1,"name":"Frieder Loch"}
```

2.1.1 Beeinflussung der Serialisierung

```
public class WeatherData {
  @JsonProperty("temp_celsius")
  private double tempCelsius;
}
```

```
@JsonPropertyOrder({"name", "id"})
public class Employee{
  public int id;
  public String name;
}
```

`@JsonIgnore`, `@JsonInclude(Include.NON_NULL)` (nur nicht-null-Werte), `@JsonFormat(pattern = "dd-MM-yyyy")`

```
@JsonRootName(value="user")
public class Customer {
  public int id;
  public String name;
}
// ...
var mapper = new ObjectMapper().enable(
  SerializationFeature.WRAP_ROOT_VALUE
);
```

Output:

```
{
"user": {
"id": 1,
"name": "Frieder Loch"
}
}
```

2.1.2 JsonGenerator

```
var generator = new JsonFactory().createGenerator(
  new FileOutputStream("employee.json"), JsonEncoding.UTF8);
jsonGenerator.writeStartObject();
jsonGenerator.writeFieldName("identity");
jsonGenerator.writeStartObject();
jsonGenerator.writeStringField("name", company.name);
jsonGenerator.writeEndObject();
```

2.1.3 Deserialisierung

```
String json = "{\"name\":\"Max\", \"alter\":30}";
ObjectMapper mapper = new ObjectMapper();
Benutzer benutzer = mapper.readValue(json, Benutzer.class); // throws
JsonMappingException
```

Deserializer:

```
public class CompanyJsonDeserializer extends JsonDeserializer {
  @Override
  public Company deserialize(JsonParser jP, DeserializationContext dC)
throws IOException {
    var tree = jP.readValueAs(JsonNode.class);
    var identity = tree.get("identity");
    var url = new URL(tree.get("website").asText());
    var nameString = identity.get("name").asText();
    var uuid = UUID.fromString((identity.get("id").asText()));
    return new Company(nameString, url, uuid);
  }
}
```

@JacksonInject:

```
public class Book {
  public String name;
  @JacksonInject
  public LocalDateTime lastUpdate;
}
InjectableValues inject = new InjectableValues.Std()
.addValue(LocalDateTime.class, LocalDateTime.now());
Book[] books = new ObjectMapper().reader(inject)
.forType(new TypeReference<Book[]>(){}).readValue(jsonString);
```

3 Generics

3.1 Iterator

```
for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
  String s = it.next();
  System.out.println(s);
}
```

3.1.1 Iterable und Iterator

```
interface Iterable<T> {
  Iterator<T> iterator();
}
```

```
interface Iterator<T> {
  boolean hasNext();
  T next();
}
```

Klassen, die `Iterable` implementieren, können in einer enhanced `for`-Schleife verwendet werden:

3.2 Generische Methoden

```
public static <T> Stack<T> multiPush(T value, int times) {
  var result = new Stack<T>();
  for(var i = 0; i < times; i++) {
    result.push(value);
  }
  return result;
}
```

Typ wird am Kontext erkannt:

```
Stack<String> stack1 = multiPush("Hallo", 3);
Stack<Double> stack2 = multiPush(3.141, 3);
```

Generics mit Type-Bounds verwenden immer `extends`, kein `implements`.

Vorsicht:

```
private static <T extends Comparable<T>> T majority(T x, T y, T z) {
  // ...
}
// ...
Number n = majority(1, 2.4232, 3); // Compilerfehler
Main.<Number>majority(1, 2.4232, 3); // Eigentlich OK, aber Number hat
keine Comparable-Implementierung
```

Die JVM hat keine Typinformationen zur Laufzeit ⟶ Non-Reifiable Types, Type-Erasure.

3.3 Wildcards

```
public static void printAnimals(List<? extends Animal> animals) {
  for (Animal animal : animals) {
    System.out.println(animal.getName());
  }
}
public static void main(String[] args) {
  List<Animal> animalList = new ArrayList<>();
  printAnimals(animalList);
  List<Cat> catList = new ArrayList<>();
```

```
  printAnimals(catList);
}
```

3.4 Variance

	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	×	×

3.5 Generics vs ArrayList

```
ArrayList<String> stringsArray = new ArrayList<>();
ArrayList<Object> objectsArray = stringsArray; // Compilerfehler

String[] stringsArray = new String[10];
Object[] objectsArray = stringsArray; // OK
objectsArray[0] = Integer.valueOf(2); // Exception
```

Kompiliert nicht mit Subtypen:

```
Object[] objectsArray = new Object[10];
String[] stringsArray = objectsArray; // Compilerfehler
```

3.5.1 Kovarianz

```
Stack<? extends Graphic> stack = new Stack<Rectangle>();
stack.push(new Graphic()); // nicht erlaubt
stack.push(new Rectangle()); // auch nicht erlaubt
```

⟶ Kovariante generische Typen sind **readonly**.

3.5.2 Kontravarianz

```
public static void addToCollection(List<? super Integer> list, Integer i)
{
  list.add(i);
}

List<Object> objects = new ArrayList<>();
addToCollection(objects, 1); // OK
```

Lesen aus Collection mit Kontravarianz ist nicht möglich:

```
Stack<? super Graphic> stack = new Stack<Object>();
stack.add(new Object()); // Nicht OK, Object ist kein Graphic
stack.add(new Circle()); // OK
Graphic g = stack.pop(); // Compilerfehler
```

3.5.3 PECS

> Producer Extends, Consumer Super

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

3.5.4 Bivarianz

Schreiben nicht möglich, Lesen mit Einschränkungen:

```
static void appendNewObject(List<?> list) {
  list.add(new Object()); // Compilerfehler
}
public static void printList(List<?> list) {
  for (Object elem: list) {
    System.out.print(elem + " "); // OK
  }
  System.out.println();
}
```

4 Annotations und Reflection

Beispiele für Annotations:

- `@Override`
- `@Deprecated`
- `@SuppressWarnings(value = "unchecked")`
- `@FunctionalInterface`

4.1 Implementation von Annotations

```
@Target(ElementType.METHOD) // oder TYPE, FIELD, PARAMETER, CONSTRUCTOR
@Retention(RetentionPolicy.RUNTIME) // oder SOURCE, CLASS
public @interface Profile { }
```

4.2 Reflection

```
Class c = "foo".getClass();
Class c = Boolean.class;
```

Wichtige Methoden von `Class`:

- `public Method[] getDeclaredMethods() throws SecurityException`
- `public Constructor<?>[] getDeclaredConstructors() throws SecurityException`
- `public Field[] getDeclaredFields() throws SecurityException`

4.2.1 Methoden

- `public String getName()`
- `public Object invoke(Object obj, Object... args)`

4.2.2 Auswahl annotierter Methoden

```
for (var m : methods) {
  if(m.isAnnotationPresent(Profile.class)) {
    PerformanceAnalyzer.profileMethod(testFunctions, m, new Object[]
{array});
  }
}
```

4.2.3 Aufruf und Profiling der Methoden

```
public class PerformanceAnalyzer {
  public static void profileMethod(Object object, Method method, Object[]
args) {
    long startTime = System.nanoTime();
    try {
      method.invoke(object, args);
    } catch (IllegalAccessException | InvocationTargetException e) {
      e.printStackTrace();
    }
    long endTime = System.nanoTime();
    long elapsedTime = endTime - startTime;
     System.out.println(method.getName() + " took " + elapsedTime + "
nanoseconds to execute.");
  }
}
```

5 Arrays und Listen

5.1 Sortieren

5.1.1 Platz finden und Platz schaffen

Beispiel (Highscore-Liste):

- Iteration vom Ende zu Beginn
- Neuer Score grösser als Score an `position - 1`?
- Ja: Kopiere `position - 1` an `position`
- Nein: Iteration abbrechen
- Eintrag an `position` speichern

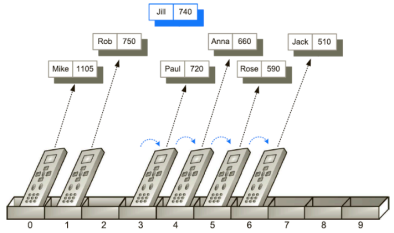


Abbildung 2: Leaderboard

```
public void add(GameEntry entry) {
  int newScore = entry.getScore();
  if(isHighscore(newScore)) {
    if(numEntries < board.length) {
      numEntries++;
    }
    int j = numEntries - 1;
    for(; j > 0 && board[j - 1].getScore() < newScore; j--) {
      board[j] = board[j - 1]
      j--;
    }
    board[j] = entry;
  }
}
```

5.1.2 Insertion Sort

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

5.2 Linked List

5.2.1 Einfügen am Anfang

1. Neuen Knoten mit altem Kopf verketten
2. `head` auf neuen Knoten setzen

5.2.2 Einfügen am Ende

1. Neuen Knoten auf `null` zeigen lassen
2. Früheren Endknoten mit neuem Knoten verketten
3. `tail` auf neuen Knoten setzen

5.3 Doubly Linked List

5.3.1 Einfügen eines Knotens am Anfang

```
public void addFirst(T element) {
  DoublyLinkedNode<T> newNode = new DoublyLinkedNode<>(element, null,
null);
  DoublyLinkedNode<T> f = header.getNext();
  header.setNext(newNode);
  newNode.setNext(f);
  size++;
}
```

5.3.2 Entfernen eines Knotens am Ende

```
public T removeLast() {
  DoublyLinkedNode<T> oldPrevNode
  = trailer.getPrev();
  DoublyLinkedNode<T> prevPrevNode
  = oldPrevNode.getPrev();
  trailer.setPrev(prevPrevNode);
  prevPrevNode.setNext(trailer);
  oldPrevNode.setPrev(null);
  oldPrevNode.setNext(null);
```

```
  size--;
  return oldPrevNode.getElement();
}
```

6 Algorithmenparadigmen

Definition: **Endliches**, **deterministisches** und **allgemeines** Verfahren unter Verwendung **ausführbarer**, **elementarer** Schritte.

6.1 Set-Covering Problem

Beispiel: Alle Staaten mit möglichst wenigen Radiosendern abdecken.

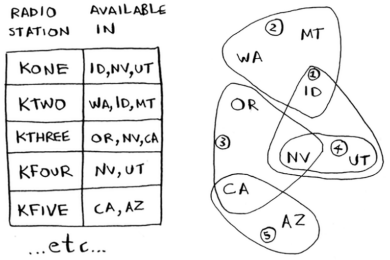


Abbildung 3: Set-Covering Problem

Optimaler Algorithmus:

- Teilmengen der Stationen aufzählen
- Minimale Anzahl Stationen wählen
- Problem: 2^n mögliche Kombinationen

Greedy Algorithmus:

- Immer Sender wählen, der die meisten neuen Staaten hinzufügt

```
public static void calculateSolution(HashSet<String> statesNeeded,
HashMap<String, HashSet<String>> stations) {
  var finalStations = new HashSet<String>();
  while (!statesNeeded.isEmpty()) {
    String bestStation = "";
    var statesCovered = new HashSet<String>();
    for (String station : stations.keySet()) {
      var covered = new HashSet<String>(statesNeeded);
      covered.retainAll(stations.get(station));
      if (covered.size() > statesCovered.size()) {
        bestStation = station;
        statesCovered = covered;
      }
    }
    statesNeeded.removeAll(statesCovered);
    finalStations.add(bestStation);
  }
  System.out.println(finalStations);
}
```

6.2 Binary Search

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

6.3 Backtracking

- Ziel erreicht:
- Lösungspfad aktualisieren
- **True** zurückgeben
- Wenn (x, y) bereits Teil des Lösungspfades:
- **False** zurückgeben
- (x, y) als Teil des Lösungspfades markieren
- Vorwärts in X-Richtung suchen: →
- Keine Lösung: In Y-Richtung abwärts suchen: ↓
- Keine Lösung: Zurück in X-Richtung suchen: ←
- Keine Lösung: Aufwärts in Y-Richtung suchen: ↑
- Immer noch keine Lösung: (x, y) aus Lösungspfad entfernen und **Backtracking**
- **False** zurückgeben

Vorgehensmodell:

```
fn backtrack(k: Konfiguration)
if [k ist Lösung] then
[gib k aus]
else
for each [direkte Erweiterung k' von k]
backtrack(k')
```

6.3.1 Sudoku

```
public boolean solve(int row, int col) {
  // Lösung gefunden?
  if (row == 8 && col == 9) return true;
  if (col == 9) {
    row++;
    col = 0;
  }
  // Feld schon befüllt?
  if (sudokuArray[row][col] != 0) {
    // Wenn ja: Nächstes Feld
    return solve(row, col + 1);
  } else {
    for (int num = 1; num <= 9; num++) {
      // Ergänzung regelgerecht?
      if (checkRow(row, num) && checkCol(col, num) && checkBox(row, col,
num)) {
        // Neue Teillösung erstellen und ergänzen
        sudokuArray[row][col] = num;
        if (solve(row, col + 1)) return true;
      }
    }
    // Backtracking
    sudokuArray[row][col] = 0;
    return false;
  }
}
```

6.3.2 Knight-Tour

```
boolean knightTour(int[][] visited, int x, int y, int pos) {
  visited[x][y] = pos;

  if (pos >= N * N) {
    return true;
  }

  for (int k = 0; k < 8; k++) {
    int newX = x + row[k];
    int newY = y + col[k];

    if (isValid(newX, newY) && visited[newX][newY] == 0) {
      if (knightTour(visited, newX, newY, pos + 1)) {
        return true;
      }
    }
  }

  visited[x][y] = 0;
  return false;
}
```

6.4 Dynamische Programmierung

```
public static long fibonacci(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];
}
```

7 Algorithmenanalyse

7.1 Theoretische Analyse

- Atomare Operationen
- In Pseudocode identifizierbar
- Annahme:
- Benötigen konstante Zeit
- Summe der primitiven Operationen bestimmt die Laufzeit

7.2 Big-O Notation

$f(n)$ ist $O(g(n))$, falls reelle, positive Konstante $c > 0$, Ganzzahlkonstante $n_0 \geq 1$, so dass $f(n) \leq c \cdot g(n)$ für $n \geq n_0$

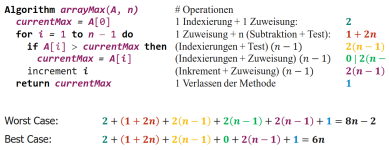


Abbildung 4: Primitive Operationen zählen

8 Sortieralgorithmen

8.1 Selectionsort

Beim Selectionsort wird immer das grösste/kleinste Element gesucht und an der nächsten Stelle in einer zweiten Liste eingefügt. Alternativ kann auch geswappt werden.

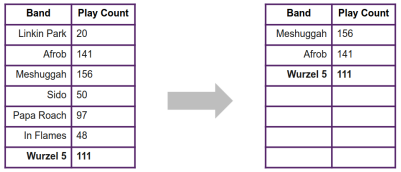


Abbildung 5: Selectionsort

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

Laufzeit: $O(n^2)$

8.2 Insertionsort

```
public static void insertionsort(Comparable[] a) {
  int n = a.length;
  for (int i = 1; i < n; i++) {
```

```
    for (int j = i; j > 0 && a[j] < a[j - 1]; j--) {
      swap(a, j, j - 1);
    }
  }
}
```

Laufzeit: $O(n^2)$

- Element entnehmen und an der richtigen Stelle in sortierter Liste einfügen
- Gut bei teilweise sortierten Arrays

8.3 Bubblesort

Array von links nach rechts durchgehen

- Wenn Element grösser als rechter Nachbar: tauschen

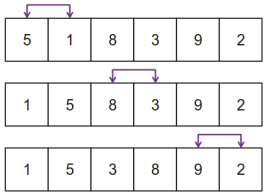


Abbildung 6: Bubblesort

```
void bubblesort(int[] a) {
  for (int n = array.length; n > 1; n --) {
    for (int i = 0; i < n - 1; i++) {
      if (a[i] > a[i + 1]) {
        swap(a, i, i + 1);
      }
    }
  }
}
```

Laufzeit: $O(n^2)$

9 Recursion

9.1 Schlüssel suchen (iterativ)

1. Lege einen Haufen Schachteln zum Durchsehen an
2. Nimm eine Schachtel vom Haufen und sieh sie durch
3. Wenn du eine Schachtel findest, lege sie auf den Haufen, um sie später zu durchsuchen
4. Wenn du einen Schlüssel findest, bist du fertig
5. Gehe zu Schritt 2

```
def look_for_key(main_box):
  pile = main_box.make_a_pile_to_look_through()
  while pile is not empty:
    box = pile.grab_a_box()
    for item in box:
      if item.is_a_box():
        pile.append(item)
      elif item.is_a_key():
        print("Found the key")
```

9.2 Schlüssel suchen (rekursiv)

1. Sieh die Schachtel durch
2. Wenn Schachtel gefunden: Gehe zu Schritt 1
3. Wenn Schlüssel gefunden: Fertig

```
def look_for_key(box):
  for item in box:
    if item.is_a_box():
      look_for_key(box)
    elif item.is_a_key():
      print("Found the key")
```

9.3 Array umkehren

```
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;
}
```

Umwandlung in einen iterativen Algorithmus:

```
int[] reverseArrayIteratively(int[] a, int i, int j) {
  while (i < j) {
    int temp = a[j];
    a[j] = a[i];
    a[i] = temp;
    i += 1;
    j += 1;
  }
  return a;
}
```

9.4 Endrekursion

Summe (nicht end-rekursiv):

```
int recsum(int x) {
  if (x == 0) {
    return 0;
  } else {
    return x + recsum(x - 1);
  }
}
```

Summe (end-rekursiv):

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

10 Stack & Queue

10.1 Array-basierter Stack

Push:

```
void push(E element) {
  if (size() == data.length) {
    resize();
  }
  data[t++] = element;
}

void resize() {
  int oldSize = data.length;
  int newSize = oldSize * 2;
  E[] temp = (E[]) new Object[newSize];
  for (int i = 0; i < oldSize; i++) {
    temp[i] = data[i];
  }
  data = temp;
}
```

Pop:

```
public E pop() {
  if (isEmpty()) {
    throw new IllegalStateException ("Stack is empty!");
  }
  E element = data[t];
  data[t--] = null;
  return element;
}
```

10.2 Queue

- `enqueue(E)`: Element am Ende der Queue einfügen
- `E dequeue()`: Element vom Anfang der Queue entfernen und zurückgeben
- `E first()`: Liefert erstes Element, ohne es zu entfernen
- `int size()`: Anzahl gespeicherter Elemente
- `boolean isEmpty()`

10.2.1 Enqueue

- `storedElements = 0`
- `front = 0`

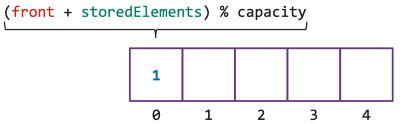


Abbildung 7: Enqueue

→ `storedElements = 1` (`front` bleibt 0)

```
void enqueue(E element) {
  if (storedElements == capacity) {
    throw new IllegalStateException();
  } else {
    int r = (front + storedElements) % capacity;
    data[r] = element;
    storedElements++;
  }
}
```

10.2.2 Dequeue

- `storedElements -= 1`
- `front = (front + 1) % capacity`

```
E dequeue() {
  if (isEmpty()) {
    return null;
  } else {
    E elem = data[front];
    front = (front + 1) % capacity;
    storedElements--;
    return elem;
  }
}
```

10.3 Ringbuffer

```
synchronized void add(E element) throws Exception {
  int index = (tail + 1) % capacity;
  size++;

  if(size == capacity) {
    throw new Exception("Buffer Overflow");
  }

  data[index] = element;
  tail++;
}
synchronized E get() throws Exception {
  if (size == 0) {
    throw new Exception("Empty Buffer");
  }

  int index = head % capacity;
  E element = data[index];

  head++;
  size--;
  return element;
}
```

11 Trees

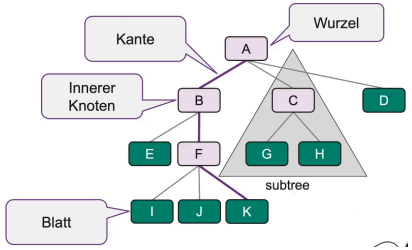


Abbildung 8: Tree

- Tiefe eines Knotens: Anzahl Vorgänger
- Höhe eines Baums: Maximale Tiefe der Knoten eines Subtree
- Subtree (Unterbaum): Baum aus einem Knoten und seinen Nachfolgern

Java Tree Interface:

```
interface Tree<E> extends Iterable<E> {
  Node<E> root();
  Node<E> parent(Node<E> p);
  Iterable<Node<E>> children(Node<E> p);
  int numChildren(Node<E> p);
  boolean isInternal(Node<E> p); // Node
  boolean isExternal(Node<E> p); // Leaf
  boolean isRoot(Node<E> p);
}
```

Binary Tree in Java:

```
interface BinaryTree<E> extends Tree<E> {
  Node<E> left(Node<E> p);
  Node<E> right(Node<E> p);
  Node<E> sibling(Node<E> p);
  Node<E> addRoot(E e);
  Node<E> addLeft(Node<E> p, E e);
  Node<E> addRight(Node<E> p, E e);
}
```

11.1 Arten von Bäumen

11.1.1 Binärer Suchbaum

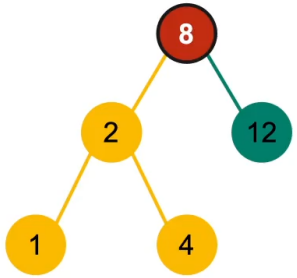


Abbildung 9: Binary Search Tree

- Jeder Knoten trägt einen Schlüssel
- Alle Schlüssel im linken Teilbaum sind kleiner als die Wurzel des Teilbaums
- Alle Schlüssel im rechten Teilbaum sind grösser als die Wurzel des Teilbaums
- Die Unterbäume sind auch binäre Suchbäume

11.2 Algorithmen

11.2.1 Tiefe

```
int depth(Node<E> p) {
  if (isRoot(p)) {
    return 0;
  } else {
    return 1 + depth(parent(p));
  }
}
```

11.2.2 Höhe des Trees

```
int height(Node<E> p) {
  int h = 0;
  for (Node<E> c : children(p)) {
    h = Math.max(h, 1 + height(c));
  }
  return h;
}
```

11.2.3 Sibling

11.3 Traversierungen

11.3.1 Preorder (W - L - R)

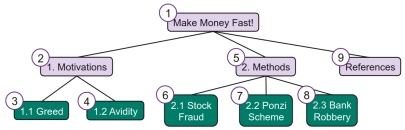


Abbildung 10: Preorder

```
algorithm preOrder(v)
  visit(v)
  for each child w of v
    preOrder(w)
```

11.3.2 Postorder (L - R - W)

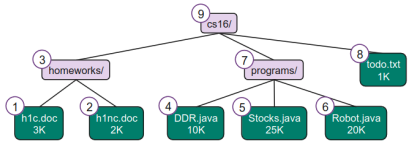
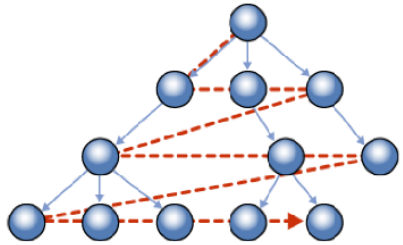


Abbildung 11: Postorder

```
algorithm postOrder(v)
  for each child w of v
    postOrder(w)
visit(v)
```

11.3.3 Breadth-First / Level-Order

Beispiel: Sudoku (Welcher Zug soll als nächstes gewählt werden).



```
algorithm breadthFirst()
  // Q enthält ROot
  while Q not empty
v = Q.dequeue()
    visit(v)
for each child w in children(v)
Q.enqueue(w)
```

11.3.4 Inorder (L - W - R)

Beispiel: Arithmetische Ausdrücke

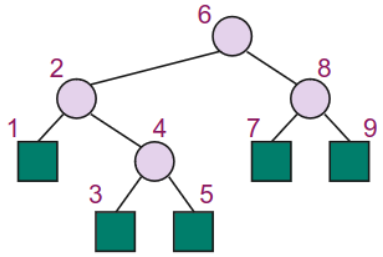
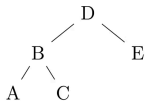


Abbildung 13: Inorder

```
algorithm inOrder(v)
  if hasLeft(v)
inOrder(left(v))
  visit(v)
if hasRight(v)
inOrder(right(v))
```

11.3.5 Übersicht



- **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

Abbildung 14: Traversierungen (Übersicht)

11.4 Heapsort

Binärer Baum mit folgenden Eigenschaften:

- Baum ist vollständig (alle Blätter haben dieselbe Tiefe)
- Schlüssel jedes Knotens kleiner als oder gleich wie der Schlüssel seiner Kinder

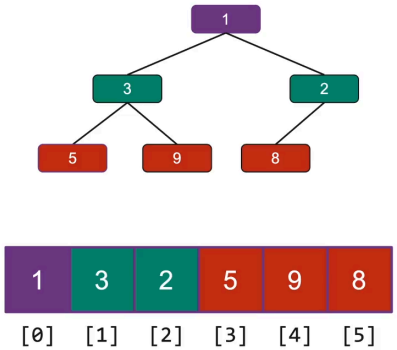


Abbildung 15: Heap als Array

11.4.1 Ablauf

1. Nehme Root-Element aus dem Heap heraus und lege es in Array (Root ist immer das kleinste Element)
2. Letztes Element im Heap in Root (→ Heap-Eigenschaft wird verletzt)
3. Root-Element im Tree versenken (percolate)
4. Zurück zu Schritt 1.

Percolate-Algorithmus:

```
void percolate(Comparable[] array, int startIndex, int last) {
  int i = startIndex;
  while (hasLeftChild(i, last)) {
```

```
    int left = getLeftChild(i);
    int right = getRightChild(i);
    int exchangeWith = 0;

    if (array[i].compareTo(array[left]) > 0) {
      exchangeWith = left;
    }
    if (right <= last && array[left].compareTo(array[right]) > 0) {
      exchangeWith = right;
    }
    if (exchangeWith == 0 || array[i].compareTo(array[exchangeWith]) <=
0) {
      break;
    }
    swap(array, i, exchangeWith);
    i = exchangeWith;
  }
}
```

Heap-Sort-Funktion:

```
void heapSort(Comparable[] array) {
  int i;
  heapifyMe(array);
  for (i = array.length - 1; i > 0; i--) {
    swap(array, 0, i); // Erstes Element mit letztem tauschen
    percolate(array, 0, i - 1); // Heap wiederherstellen
  }
}
```

12 Design Patterns

12.1 Arten

- Erzeugungsmuster: Abstrahieren Instanziierung (Factory, Singleton)
- Strukturmuster: Zusammensetzung von Klassen & Objekten zu grösseren Strukturen (Adapter, Fassade)
- Verhaltensmuster: Algorithmen und Verteilung von Verantwortung zwischen Objekten (Iterator, Visitor)

12.1.1 Template-Method

- Rumpf (Skelett) eines Algorithmus definieren, Teilschritte in Subklasse spezifizieren
- Subklasse definiert nur Teilschritte, die Struktur des Algorithmus bleibt bestehen
- Entscheidung: Welche Teile sind unveränderlich, welche können angepasst werden?

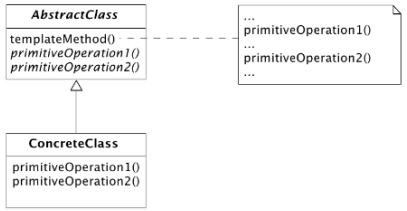


Abbildung 16: Template-Method

Das Template-Method-Pattern wird oft in Frameworks benutzt, im Sinne des Holywood-Prinzips: *Don't call us, we call you.*

Euler-Tour:

- Generische Traversierung binärer Bäume
- Jeder Knoten wird drei mal besucht:
- Von links (preorder)
- Von unten (inorder)
- Von rechts (postorder)

```
public abstract class EulerTour<E> {
  protected abstract void visitLeaf(Node<E> node);

  protected abstract void visitLeft(Node<E> node);

  protected abstract void visitBelow(Node<E> node);

  protected abstract void visitRight(Node<E> node);

  public void eulerPath(Node<E> node, Node<E> parent) {
    if (node == null) {
      return;
    }
    if (node.isLeaf()) {
      visitLeaf(node);
    } else {
      visitLeft(node);
      eulerPath(node.getLeft(), node);
      visitBelow(node);
      eulerPath(node.getRight(), node);
      visitRight(node);
    }
  }
}
```

12.1.2 Visitor-Pattern

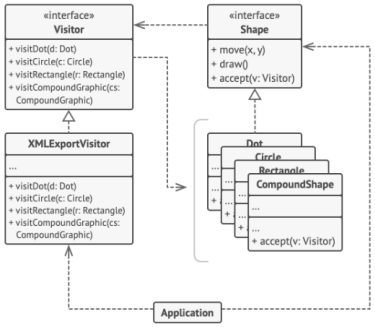


Abbildung 17: Visitor-Pattern

13 Sets, Maps, Hashing

13.1 Multiset

- Set mit erlaubten *Duplikaten*
- Was ist ein Duplikat?
- `equals()` (Standard) vs ==

Add:

```
public int add(E element, int occurrences) {
  if (occurrences < 0) {
    throw new IllegalArgumentException("Occurrences cannot be negative.");
  }
  int currentCount = elements.getOrDefault(element, 0);
  int newCount = currentCount + occurrences;
  elements.put(element, newCount);
  return newCount;
}
```

13.1.1 Divisionsrestverfahren

- $h(x) = x \bmod 10$: Nach Kriterien gute Hashfunktion, jedoch ist bei jeder Zahl nur die letzte Ziffer relevant
- $h(x) = x \bmod 2^2$: Die letzten beiden Ziffern in der Binärrepräsentation mappen immer auf dieselben Hashwerte
- → Ungerade Zahlen (oder besser Primzahlen) verteilen sich besser
- Wird oft als Kompressionsfunktion nach dem Hash verwendet

13.1.2 Komponentensumme

1. Schlüssel in Komponenten unterteilen
2. Komponenten summieren
3. Overflow ignorieren

→ Problematisch, da z.B. bei Strings die Reihenfolge der Chars keine Rolle spielt

13.1.3 Polynom-Akkumulation

- Komponentensumme, mit gewichteten Komponenten:
- $p(z) = a_0 + a_1 z + a_2 z^2 + \cdots + a_{n-1} z^{n-1}$
- Gut für Strings
- Mit $z = 33$ maximal 6 Kollisionen bei 50'000 englischen Wörtern

13.1.4 Rezept für Benutzerdefinierte Typen

```
@Override public int hashCode() {
  int result = 17;
  result =
  31 * result + (name != null ? name.hashCode() : 0);
  result = 31 * result + age;
  return result;
}
```

13.2 Geschlossene Adressierung

- Behälter sind verkettete Listen
- Platz nicht begrenzt, keine Überläufer

Falls es zu einer Kollision kommt, wird in demselben Bucket der neue Wert als verlinkte Liste angehängt. Die Komplexitat beträgt $O(\frac{n}{N})$, wobei:

- n: Einträge in der Tabelle
- N: Buckets

13.3 Offene Adressierung

- Für überläufer in anderen Behältern Platz suchen
- Sondierungsfolge bestimmt Weg zum Speichern und Wiederauffinden der Überläufer

13.3.1 Sondieren

- Lineare Sondierung: Überläufer ins nächste freie Bucket schreiben
- $s(k, 1) = h(k) + 1$
- $s(k, 2) = h(k) + 2$
- Quadratische Sondierung
- $s(k, 1) = h(k) + 1^2$
- $s(k, 2) = h(k) + 2^2$
- $s(k, 3) = h(k) + 3^2$

Löschen von Werten: Datensätze nicht physisch löschen, nur als gelöscht markieren

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

13.3.2 Dynamisches Hashing

- Hashtabelle kann nur mit Aufwand vergrössert werden, um Kollisionen zu verringern
- Reorganisation bei vielen Kollisionen können nur durch Reservieren eines sehr grossen Speicherbereichs zu Beginn verhindert werden
- Wie könnte ein flexibleres Hashverfahren aussehen?