1 Algorithmus

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

2 Input und Output

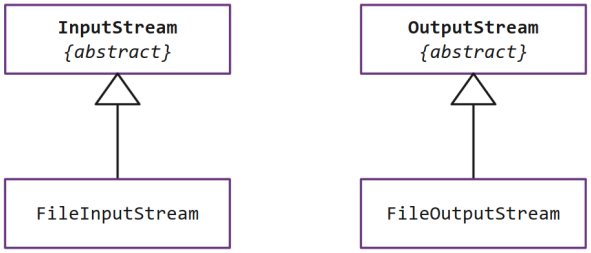

Abbildung 1: Klassenhierarchie von Input und Output

2.1 Input

2.1.1 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));
```

2.1.2 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.

2.2 Output

2.2.1 File-Writer

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

2.3 Zusammenfassung

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

3 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();
  // …
}
```

3.1 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"}
```

3.1.1 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"
  }
}
```

3.1.2 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();
```

3.1.3 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);
```

4 Generics

4.1 Iterator

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

4.1.1 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:

4.2 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
```

Erstellung eines Type T geht nicht:

```
T t = new T(); // Compilerfehler
T[] array = (T[]) new Object[10]; // Funktioniert
```

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

So laufen:
- Alte, nicht generische Programme auf neuen JVMs
- Neue, generische Programme auf alten JVMs
- Alter, nicht generischer Code kompiliert mit neuen Compilern

4.3 Unterschied Comparable

```
<T extends Comparable<T>> T max(T x, T y) {
  return x.compareTo("lmaooo") > 0 ? x : y; // Compilerfehler
}
<T extends Comparable> T max(T x, T y) {
  return x.compareTo("lmaooo") > 0 ? x : y; // OK
}
```

4.4 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);
}
```

4.5 Variance

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

4.6 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
```

4.6.1 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**.

4.6.2 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
```

4.6.3 PECS

> Producer Extends, Consumer Super

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

4.6.4 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();
}
```

5 Annotations und Reflection

Beispiele für Annotations:
- `@Override`
- `@Deprecated`
- `@SuppressWarnings(value = "unchecked")`
- `@FunctionalInterface`

5.1 Implementation von Annotations

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

5.2 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`

5.2.1 Methoden

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

5.2.2 Auswahl annotierter Methoden

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

5.2.3 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.");
  }
}
```