Inhalt - Java

1. Einleitung

Objektorientierte Programmierung (OOP) ist ein Programmierparadigma, dass eine saubere Trennung von Verantwortlichkeiten erlaubt, eine einheitliche Modellierung komplexer Sachverhalte ermöglicht und die Wartbarkeit und Erweiterbarkeit von Programmen gewährleistet. Dabei werden sämtliche Strukturteile in Klassen abstrahiert und ihre Beziehungen über einzelne Objekte realisiert. Die Kunst der objektorientierten Programmierung besteht nicht nur darin logische Abfolgen zu realisieren oder mathematisch-logische Probleme effektiv zu implementieren, sondern gleichzeitig darin, ein Problem in eine räumliche Struktur zu gliedern. Hierzu werden Klassen und während der Laufzeit eines Programms Objekte verwendet. Auf die genauen Begrifflichkeiten wird im Folgenden weiter eingegangen.

2 Java-Grundlagen

Dieses Kapitel gibt eine Übersicht über grundlegende Sprachmittel in Java. Dazu gehören Kontrollstrukturen, die den Ablauf eines Programms steuern, Datenstrukturen (Arrays und das Collections-Framework) sowie Exception-Handling.

2.1 Kontrollstrukturen

Kontrollstrukturen steuern, in welcher Reihenfolge Anweisungen in einem Programm ausgeführt werden. Sie bilden eine logische Komponente ab, die grundlegende Funktionalitäten bereitstellen. Die beiden Hauptstrukturen sind Verzweigungen und Schleifen.

2.1.1 Verzweigungen

Mit if-else-Konstrukten werden Entscheidungen im Programm umgesetzt. Bedingungen müssen dabei immer einen booleschen Wert (`true` oder `false`) zurückgeben.

```
int x = 12;
if (x > 10) {
    System.out.println("größer als 10");
} else if (x == 10) {
    System.out.println("gleich 10");
} else {
    System.out.println("kleiner als 10");
}

Ausgabe:
größer als 10
```

2.1.2 Switch

Das switch-Konstrukt eignet sich, wenn ein Wert gegen mehrere Möglichkeiten geprüft werden soll. Diese Struktur ermöglicht praktisch viele If-Else-Abfragen kompakt darzustellen und umzusetzen. Ein case beschreibt hierbei eine mögliche Wertebelegung und entsprechende Programmausgabe.

```
int wochentag = 2;
switch (wochentag) {
    case 1 -> System.out.println("Montag");
    case 2 -> System.out.println("Dienstag");
    case 3, 4 -> System.out.println("Mittwoch oder Donnerstag");
    default -> System.out.println("Wochenende oder unbekannt");
}

Ausgabe:
Dienstag
```

2.1.3 Schleifen

Schleifen wiederholen Anweisungen solange eine Bedingung erfüllt ist oder bis eine bestimmte Anzahl an Durchläufen erreicht ist (while- bzw. for- Schleifen).

```
// klassische for-Schleife
for (int i = 0; i < 3; i++) {
    System.out.print(i + " ");
}

// foreach (enhanced for)
int[] arr = {10, 20, 30};
for (int v : arr) {
    System.out.print(v + " ");
}

// while-Schleife
int n = 3;
while (n > 0) {
    System.out.print(n + " ");
    n--;
}

// do-while-Schleife
int m = 0;
do {
    System.out.print("läuft mindestens einmal ");
} while (m > 0);

Ausgabe:
0 1 2
10 20 30
3 2 1
läuft mindestens einmal
```

2.2 Datenstrukturen

Eine Datenstruktur bestimmt, in welcher Art und Weise Daten verwaltet bzw. angeordnet und gespeichert werden. In Java gibt es zwei große Kategorien von Datenstrukturen; Arrays und das Collections-Framework. Arrays sind einfach und sehr schnell, aber unflexibel in der Größe (statisch). Collections sind dynamisch, anpassbar und decken eine Vielzahl von Anwendungsfällen ab, sind jedoch meist nicht direkt.

2.2.1 Arrays

Ein Array ist eine Sammlung von Elementen gleichen Typs. Die Länge eines Arrays wird beim Anlegen festgelegt und kann nicht mehr geändert werden. Bei der Initialisierung eines Arrays wird hinter den Datentypen bzw. Klasse ein `[]` angefügt.

```
int[] nums = new int[3];
nums[0] = 5; nums[1] = 9; nums[2] = 2;
System.out.println(nums.length);   // 3
System.out.println(nums[1]);       // 9
```

Wichtige Eigenschaften:

- Feste Länge
- Indexzugriff in O(1)
- Homogen (nur ein Datentyp erlaubt)

Arrays sind für sehr große Datenmengen effizient, aber unflexibel. Für dynamische Sammlungen greift man auf Collections zurück. Collection selbst ist eine Java ein Interface, das grundlegende Methoden für die Verwaltung von Objekten und den Zugriff auf diesen Datensatz gewährleistet.

2.2.2 Listen (`List`) – Beispiel `ArrayList`

Eine ArrayList ist eine dynamische Liste, die intern ein Array nutzt, dessen Größe bei Bedarf automatisch vergrößert wird.

```
import java.util.ArrayList;

List<String> namen = new ArrayList<>();
namen.add("Ada");
namen.add("Linus");
namen.add("Ada"); // Duplikate erlaubt
System.out.println(namen.get(1));  // Linus
System.out.println(namen.size());  // 3
```

Wichtige Methoden von `List`:

`add(E e)`	fügt ein Element hinzu
`get(int index)`	liest Element per Index
`remove(int index) / remove(Object o)`	entfernt Element
`size()`	Anzahl der Elemente
`contains(Object o)`	prüft Existenz

`List` ist die am häufigsten genutzte Collection in Java, da sie sich sehr flexibel verhält und Duplikate zulässt.

2.2.5 Queues und Stacks

Eine `Queue` ist eine Warteschlange (FIFO = First-In-First-Out), ein `Stack` ist ein Stapel (LIFO = Last-In-First-Out). Stack ist in Java eine eigene Klasse, kann also direkt implementiert werden. Queue hingegen ist ein Interface. Queues lassen sich jedoch in Form einer einfach verketteten Liste implementieren. Die Liste wird über Referenzen aufgebaut, d.h. jedes Element kennt seinen unmittelbaren Nachfolger (und sich selbst). Das Hinzufügen und Entfernen von Elementen kann über diese Verkettung gesteuert werden. Der Zugriff selbst erfolgt meist über ein Zeigerelement, also eine Zeigervariable in der jeweils die Referenz des aktuell betrachteten Elements verwaltet wird. Auf diese Weise kann der Zeiger z.B. über die Liste iterieren. Durch Einbindung in die Datenstrukturen (Queue) wird lediglich die Funktionalität eingeschränkt, um die Funktion der Datenstruktur gewährleisten zu können.

```
import java.util.*;

// Queue (FIFO)
Queue<String> q = new ArrayDeque<>();
q.offer("A"); q.offer("B");
System.out.println(q.poll()); // A
System.out.println(q.peek()); // B

// Stack (LIFO)
Deque<Integer> st = new ArrayDeque<>();
st.push(1); st.push(2); st.push(3);
System.out.println(st.pop());  // 3
System.out.println(st.peek()); // 2
```

Wichtige Methoden Queue:

`offer(E e)`	Element am Ende der Queue einfügen (`true` bei Erfolg)
`poll`	erstes Element entfernen und zurückgeben (`null`, wenn leer)
`peek`	erstes Element zurückgeben, ohne es zu entfernen (`null`, wenn leer)
`isEmpty()`	prüft, ob die Queue leer ist
`size()`	Anzahl der Elemente in der Queue

Wichtige Methoden Stack:

`push(E e)`	Element oben auf den Stack legen

`pop()`	oberstes Element entfernen und zurückgeben (Exception, wenn leer)
`peek()`	oberstes Element zurückgeben, ohne es zu entfernen (`null`, wenn leer)
`isEmpty()`	prüft, ob der Stack leer ist
`size()`	Anzahl der Elemente im Stack

2.2.3 Sets (`Set`) – Beispiel `HashSet`

Ein Set ist ebenfalls eine Collection, aber gleichzeitig ein eigenes Interface und somit eine Art Erweiterung bzw. Spezialisierung. Sets speichern nur eindeutige Werte, lassen also keine Duplikate zu. Ein Häufig verwendetes Set ist das HashSet, dass für die Sicherstellung der Eindeutigkeit eine Hashfunktion nutzt.

```
import java.util.Set;
import java.util.HashSet;

Set<Integer> zahlen = new HashSet<>();
zahlen.add(3); zahlen.add(3); zahlen.add(1);
System.out.println(zahlen.size());      // 2
System.out.println(zahlen.contains(1)); // true
```

Wichtige Methoden von `Set`:

`add(E e)`	fügt Element hinzu
`remove(Object o)`	entfernt Element
`contains(Object o)`	prüft Existenz
`size()`	Anzahl der Elemente

Damit eigene Klassen korrekt in einem Set funktionieren, müssen `equals()` und `hashCode()` überschrieben werden.

2.2.4 Maps (`Map`) – Beispiel `HashMap`

Map ist eine Datenstruktur die Schlüssel-Wert-Paare speichert. In Java ist Map ebenfalls als Interface implementiert. Meistens wird konkret die Klasse HashMap verwendet, welche für die interne Speicherung ein Array von Buckets nutzt, in denen Schlüssel mithilfe der hashCode()-Methode auf eine bestimmte Speicherposition (Bucket) abgebildet werden. Der Key wird über hashCode() in einen Index übersetzt. Es ist möglich, dass zwei Datensätze den gleichen Hashwert zugewiesen bekommen, was zu einer Kollision führt. In diesem Fall wird innerhalb des Buckets, in dem mehrere Einträge Liegen, eine Liste zu Verwaltung dieser angelegt. Der Interne Zugriff bzw. Abgleich nach dem gesuchten Datensatz wird dann über die equals()-Methode bestimmt.

```
import java.util.*;

Map<String, Integer> punkte = new HashMap<>();
punkte.put("Alice", 10);
punkte.put("Bob", 15);
punkte.put("Alice", 12); // überschreibt alten Wert
```

```
System.out.println(punkte.get("Alice")); // 12
System.out.println(punkte.containsKey("Bob")); // true
```

Wichtige Methoden von `Map`:

`put(K key, V value)`	fügt Key-Value-Paar hinzu oder überschreibt es
`get(Object key)`	Wert zu Schlüssel abrufen
`remove(Object key)`	entfernt Paar
`containsKey(Object key)`	prüft, ob Key existiert
`keySet()`	Menge aller Schlüssel
`values()`	Sammlung aller Werte
`entrySet()`	Sammlung aller Key-Value-Paare

2.3 Exception-Handling

Ein Laufzeitfehler ist ein Ereignis, bei dem zur Laufzeit an einem bestimmten Programmabschnitt ein Fehler auftreten würde und das Programm zum Abstürzen bringen würde. Durch den Einbau einer Exception kann ein solcher Fehler abgefangen und in das Programm eingebunden werden, d.h. es kommt nicht zu einem Absturz und dem Fehler kann auf Programmeben entgegengewirkt werden. Es handelt sich hierbei um eine Unchecked Exception, da der Fehler behoben bzw. behandelt werden kann, aber nicht zwingend muss. Es kann im speziellen auch nach konkreten Fehlermeldungen gefiltert werden. Im Folgenden eine Übersicht, die die gängigsten Exceptions umfasst:

Exception	Auslöser / Ursache	Beispielcode
NullPointerException	Zugriff auf Objekt, das null ist	`String s = null; s.length();`
ArrayIndexOutOfBoundsException	Index außerhalb der Array-Grenzen	`int[] a = new int[3]; a[3];`
StringIndexOutOfBoundsException	Ungültiger Index bei String-Operation	`"Hallo".charAt(10);`
IllegalArgumentException	Ungültiges Argument in Methode	`Thread.sleep(-5);`
ArithmeticException	Mathematisch ungültige Operation	`int x = 1/0;`

Exception	Auslöser / Ursache	Beispielcode
NumberFormatException	String kann nicht in Zahl konvertiert werden	`Integer.parseInt("abc");`

2.3.1 Try-Catch-Finally

Mittel Try-Catch-Finally können Exceptions abgefangen und behandelt werden. Dazu wird zunächst der Code im try-Block regulär ausgefüllt. Es können beliebig viele catch-Blöcke für unterschiedliche Exception eingebaut werden. Sollte einer der aufgeführten Exceptions auftreten wird der Code im zugehörigen Block ausgeführt. Der Finally Block ist optional und wird in jedem Fall ausgeführt.

```
try {
    int z = Integer.parseInt("42");
    System.out.println(z);
} catch (NumberFormatException ex) {
    System.out.println("Ungültige Zahl");
} finally {
    System.out.println("Aufräumen abgeschlossen.");
}

Ausgabe:
42
Aufräumen abgeschlossen.
```

2.3.2 Eigene Exceptions und `throw`

Mittels throw können in einem Programm neue Exceptions geworfen werden.

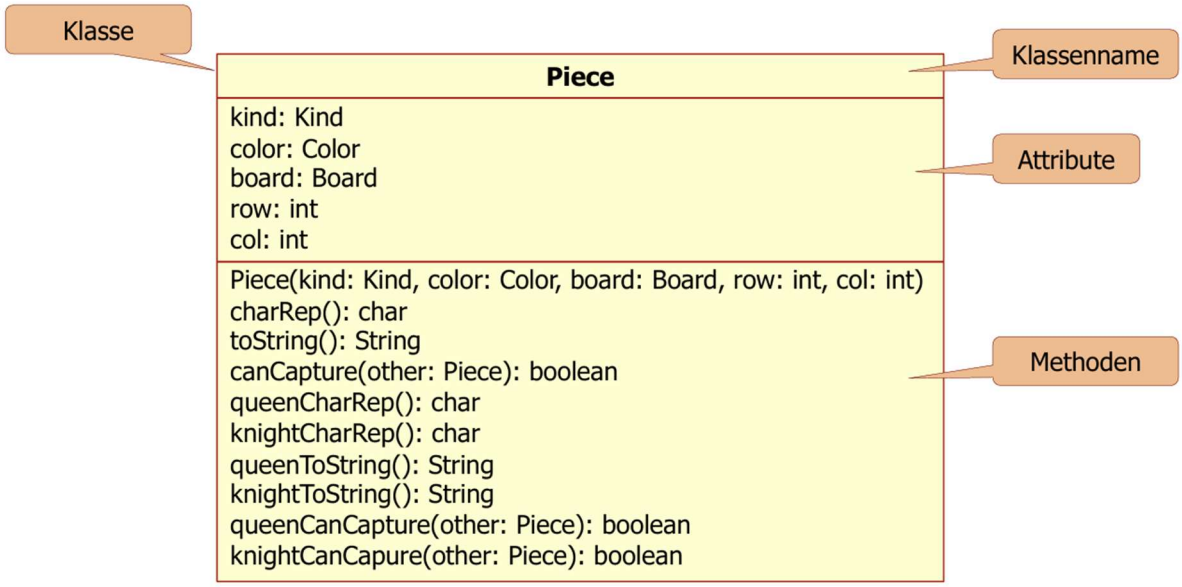
```
class NegativeValueException extends RuntimeException {
    public NegativeValueException(String msg) { super(msg); }
}

static int sqrtInt(int x) {
    if (x < 0) throw new NegativeValueException("negativer Wert: "
+ x);
    return (int)Math.sqrt(x);
}
```

3. Grundlagen der Objektorientierung in Java - **Klassen, Methoden, Attribute**

3.1 Klassen und Objekte

Eine Klasse kann im Abstrakten Sinn als ein Bauplan verstanden werden, der genutzt wird, um beliebig Viele Objekte mit gleichen Eigenschaften und Funktionalitäten zu erzeugen. Sie legt die grundlegende Struktur und übergeordnet den Verwendungszweck von Objekten der Klasse im Vorfeld fest. Eine Klasse setzt sich aus Folgenden Bestandteilen zusammen:



Ein Objekt ist eine konkrete Instanz einer solchen Klasse. Im Rahmen eines vollumfänglichen Programms wird die Wechselwirkung unter Klassen ausschließlich zwischen Objekten verschiedener Klassen ausgetragen. Die Klasse stellt insofern eine Schnittstelle zwischen den Objekten dar und beschreibt „was man mit Objekten machen kann“. Zur Laufzeit eines Programms wird das gesamte Programm selbst ausschließlich über Objekte realisiert.

3.2 Attribute

Attribute sind übergeordnet Variablen einer zugehörigen Klasse, die im Vorfeld deklariert werden und bestimmte Eigenschaften repräsentieren. Das Bedeutet, dass ein Attribut innerhalb einer Klasse für alle Operationen innerhalb der Klasse sichtbar ist und verwendet werden kann. Um die Zugriffsberechtigung zu steuern, werden Attribute mit Sichtbarkeitsmodifikatoren versehen, die im Rahmen der Kapselung eine essentielle Rolle spielen, diese sind:

`public`	Von überall sichtbar
`private`	Nur innerhalb der Klasse selbst sichtbar
`protected`	Zusätzlich sichtbar in Unterklassen und im selben Paket
`(ohne)`	Nur im selben Paket sichtbar (Package-Private)

Die erstmalige Wertbelegung von Attributen wird auch Initialisierung genannt. Solche Attribute die für jedes Objekt einer Klasse individuell belegt werden können, werden Instanzattribute genannt. Darüber hinaus gibt es globale Klassenattribute, die für eine Klasse einmalig sind. Diese werden mit dem Modifikator `static` versehen und sind von konkreten Objekten losgelöst. Der Zugriff via Punktnotation erfolgt entsprechend über die Klasse und nicht über ein Objekt (oder `this`). Äquivalent gilt dies für Methoden.

3.2.2 Datentypen

Ein Datentyp beschreibt, wie bestimmte Daten im Speicher interpretiert werden. Auf 8-Bit Ebene steht unter einer bestimmten Adresse im Speicher beispielsweise der Wert 01000001. Als Ganzzahl (`Integer`) interpretiert, also aus dem Binären ins Dezimal übersetzt liefert 01000001 den Wert 65. Auf Basis von ASCII würde 01000001 als Zeichenkette (`String`) interpretiert „A“ ergeben. Die häufigsten verwendeten grundlegende Datentypen sind:

`Integer`	Ganzzahlen
`Double`	Gleitkommazahlen – 64 Bit
`Float`	Gleitkommazahlen – 32 Bit
`Character`	Zeichen
`String`	Zeichenkette
`Boolean`	Wahrheitswert (true/false)

In Java wird zwischen primitiven Datentypen und Referenzdatentypen unterschieden. Der Unterschied ergibt sich aus dem, was konkret gespeichert wird. Ein primitiver Datentyp (`int, double, boolean, char, …`) speichert einen konkreten Wert, der direkt hinterlegt wird, wie z.B. eine Ganzzahl. Darüber hinaus stehen keine weiteren Methoden etc. zur Verfügung und es wird lediglich der konkrete Wert wiedergegeben.

Ein Referenzdatentyp speichert eine Adresse im Speicher (Heap), hinter der dann der gesamte Datensatz liegt und je nach Datentyp entsprechend interpretiert werden kann.

Dabei handelt es sich um ein Objekt einer bestimmten Klasse, für das auch ggf. Methoden bereitstehen. Jeder primitive Datentyp stellt zudem eine so genannte Wrapperklasse zur Verfügung, mit der ein konkreter Wert als Objekt dieses Datentyps „verpackt“ werden kann und zusätzlich eine große Palette nützlicher Funktionen bereitstellt. Im Folgenden wird ein Überblick über die grundlegenden Funktionen der Wrapperklassen `Integer`, `Double`, `Char` und `Boolean` gegeben.

Integer

Methode	Beschreibung
`Integer.valueOf(String)`	Wandelt String in Integer (wie parseInt, aber gibt ein Objekt zurück)
`Integer.parseInt(String)`	Wandelt String in int (primitive)
`intValue()`	Gibt den int-Wert eines Integer-Objekts zurück
`toString()`	Wandelt Integer in String
`compareTo(Integer)`	Vergleicht zwei Integer
`compare(int x, int y)`	Statischer Vergleich (-1, 0, 1)
`max(x, y), min(x, y)`	Maximum / Minimum
`Integer.toBinaryString(n)`	Binärdarstellung als String
`Integer.toHexString(n)`	Hexadezimaldarstellung als String

Double

Methode	Beschreibung
`Double.parseDouble(String)`	Wandelt String in double (primitive)
`Double.valueOf(String)`	Gibt ein Double-Objekt zurück
`doubleValue()`	Gibt double-Wert zurück
`toString()`	Wandelt Double in String
`isNaN(), isInfinite()`	Prüft auf Sonderfälle
`compare(double a, double b)`	Vergleich von zwei double-Werten

Character

Methode	Beschreibung
`Character.isLetter(c)`	Ist Buchstabe?
`Character.isDigit(c)`	Ist Ziffer?
`Character.isWhitespace(c)`	Ist Leerzeichen / Whitespace?
`Character.toUpperCase(c)`	Großbuchstabe
`Character.toLowerCase(c)`	Kleinbuchstabe
`charValue()`	Gibt den char zurück
`toString()`	Wandelt char in String um

Boolean

Methode	Beschreibung
`Boolean.valueOf(String)`	Gibt Boolean-Objekt (nur "true" ergibt true)
`Boolean.parseBoolean(String)`	Gibt primitiven boolean
`booleanValue()`	Gibt primitiven Wert zurück
`toString()`	"true" oder "false"
`compare(boolean x, boolean y)`	Vergleich (false < true)
`Boolean.TRUE`, `Boolean.FALSE`	Konstante Werte

3.3 Methoden und Konstruktoren

Methoden sind innerhalb einer Klasse definierte Funktionen. Sie stellen alle Funktionalitäten eines Objekts bereit. Der Konstruktor ist dabei die Schlüsselkomponente bei der erstmaligen Erzeugung eines Objekts. Durch den Aufruf des Konstruktors können Objekte einer Klasse erstellt werden. In der Regel werden bereits hier alle Charakterisierenden Merkmale bzw. spätere Attributwerte eines konkreten Objekts übergeben. Diese Übergabewerte werden in den Methodenkopf eingebunden und heißen Parameter. In Java ist jede Methode an einen Rückgabetypen gekoppelt, der von Methode entsprechend zurückgegeben wird. Das kann ein einfacher Datentyp, aber auch wiederum eine eigene Klasse sein. Für den Fall, dass nichts zurückgegeben wird ist dieser Rückgabetyp `void`. Des Weiteren können Methoden überladen werden. Das bedeutet, die gleiche Methode kann mit unterschiedlichen Eingaben mehrfach implementiert werden. Im Folgenden ein Beispielprogramm, bei dem eine eigene Methode `printA()entweder die Eingabe auf der Konsole ausgibt oder bei Aufruft ohne Eingabe den Text „no Input“.`

```
public void printA(String input){
  System.out.println(input);
}
public void printA(){
  System.out.println("no input");
}
//Aufruf
printA("");        //gibt leeren String aus
printA();          //gibt „no input“
printA("test");    //gibt „test“
```

3.3.1 Punktnotation und Zugriffsarten

Der Aufruf von Methoden oder der Zugriff auf Attribute, sofern dieser entsprechend freigestellt ist, eines Objekts erfolgt mittels Punktnotation. Es wird zunächst das Objekt angesprochen und über dieses mittels „Punkt“ (ObjektName.Methode/Attribut) entsprechende Methoden und Attribute aufgerufen bzw. abgefrag. Der Bezug auf sich selbst wird im Programmcode mit dem Aufruf this referenziert. Im Bereich der Objektorientierung ist nicht immer gewünscht, dass Eigenschaften eines Attributs bzw. im Allgemeinen Eigenschaften eines Objekts von außen unkontrolliert verändert werden können und ggf. die logische Funktion eines Programms beeinträchtigt wird. Hier greift das Prinzip der Abstraktion und Kapselung oder auch Single Responsibility. Das bedeutet, dass jede Klasse genau für eine Aufgabe entwickelt wird und dieser Aufgabe treu bleibt. Gleichzeitig bedeutet es, dass nur so viel von der Klasse nach außen preisgeben wird wie notwendig, um der Aufgabenerfüllung gerecht zu werden.

3.4 Abstraktion und Kapselung

Abstraktion beschreibt das Vorgehen, ein komplexes System auf seine wesentlichen Merkmale zu reduzieren. Angewendet auf die Programmierung werden einer Klasse nur die Eigenschaften und Methoden zugesprochen, die für einen bestimmten Zweck tatsächlich gebraucht werden. Dies verhindert eine Überladung von Klassen mit für die Aufgabenerfüllung unrelevanter Details.

Kapselung (Encapsulation) ergänzt die Abstraktion, indem Attribute und Methoden in einer Klasse gebündelt und mittels Zugriffsmodifikatoren (`public`, `private`, `protected`) nach außen abgeschirmt werden. In Java stehen **private** Attribute für die ideale Kapselung: Sie stellen sicher, dass kein anderer Programmteil die Klasse in einem nicht vorgesehenen Zustand hinterlässt oder auf intern kritische Daten zugreift. Methoden zum Setzen oder Auslesen der Attribute (**Setter** und **Getter**) regeln die Verwaltung nach außen. Hier können weiter bestimmte Bedingungen, wie z.B. zulässige Wertenbereiche implementiert werden.

Beispiel:

```
public class GeheimSafe {
    private String passwort;

    public GeheimSafe(String passwort) {
        this.passwort = passwort;
    }

    public boolean pruefePasswort(String eingabe) {
```

```
            return passwort.equals(eingabe);
        }
    }
```

Hier bleibt das Attribut `passwort` nach außen versteckt; die Klasse `GeheimSafe` steuert den Zugriff und erlaubt nur die Prüfung mittels `pruefePasswort(...)`.

Zusammenfassend bedeutet **Abstraktion und Kapselung bzw. Single Responsibility, dass jede Klasse nur für einen bestimmten Zweck ausgelegt ist und dabei nur die notwendigen Funktionen und Zugriffe zulässt und bereitstellt, die für die Erfüllung dieses Zwecks von Nöten sind.**

3.5 Referenzen

Während der Laufzeit eines Programms kommt es in der Regel zu verschiedenen Wechselwirkungen zwischen Objekten gleicher oder auch unterschiedlicher Klassen. Eine solche Wechselwirkung kann dabei der gegenseitige Zugriff beinhalten. Beispielsweise gibt es ein Objekt „Schachbrett“, dass die einzelnen „Spielfiguren-Objekte“ verwaltet und koordiniert. Hierbei muss einem Schachbrett-Objekt der entsprechende Zugriff auf die Spielfiguren-Objekte gewährt werden. Dies kann mittels Referenzen implementiert werden. Eine Referenz ist im abstrakten Sinn ein Zeiger, der auf eine bestimmte Position im Speicher des Computers zeigt. Bei der Konstruktion eines Objekts werden die zugehörigen Daten und Informationen wie Attributwerte und weitere charakteristische Eigenschaften in einer Speicherzelle des Hauptspeichers geschrieben. Die entsprechende Referenz, sprich die Adresse, wird in einer Referenzvariable abgelegt, über die das Objekt angefragt werden kann. Falls auf mehrere Objekte des gleichen Typs referenziert werden bietet sich für die Verwaltung eine entsprechende Datenstruktur wie eine Liste oder ein Array an. Eine solche Referenzierung von verschiedenen Klassen untereinander wird auch als Assoziation bezeichnet, die nach einem „wer kennt wen ?“-Prinzip realisiert wird.

Beispiel:

```
public class Farmer {
    private int sheeps;
    private int cows;
    public Farmer(int sheeps, int cows) {
        this.sheeps = sheeps;
        this.cows = cows;
    }
    public int getSheeps() {
        return sheeps;
    }
    public int getCows() {
        return cows;
    }
```

```
    public static void main(String[] args){
        Farmer farmer1 = new Farmer(10, 10);
        Farmer farmer2 = new Farmer(20, 20);
        System.out.println("Farmer1: " + farmer1 + " | Sheeps: " +
farmer1.getSheeps() + ", Cows: " + farmer1.getCows());
        System.out.println("Farmer2: " + farmer2 + " | Sheeps: " +
farmer2.getSheeps() + ", Cows: " + farmer2.getCows());
        farmer2 = farmer1;
        System.out.println("Farmer2: " + farmer2 + " | Sheeps: " +
farmer2.getSheeps() + ", Cows: " + farmer2.getCows());
    }
}
```

Ausgabe:

```
Farmer1: carpool.Farmer@27716f4 | Sheeps: 10, Cows: 10
Farmer2: carpool.Farmer@452b3a41 | Sheeps: 20, Cows: 20
Farmer2: carpool.Farmer@27716f4 | Sheeps: 10, Cows: 10
```

Hierbei ist biespielsweise carpool.Farmer@27716f4 die konkrete Adresse im Speicher unter der sich, der Datensatz befindet. carpool ist dabei der zugehörige Package-Name.

Eine interessante Beobachtung ergibt sich bei farmer2 = farmer1;. Hier wird nämlich die Adresse, die in der Variable farmer2 gespeichert wird umgeschrieben auf die Adresse, die in farmer1 gespeichert ist. Das macht sich in der Ausgabe insofern bemerkbar, dass farmer1 und farmer2 dieselbe Adresse beinhalten und daher auf dasselbe Objekt verweisen. Sofern carpool.Farmer@452b3a41 durch keinen anderen Zeiger verwaltet wird, wird dieses Objekt im Speicher Nutzlos und zu einem bestimmten Zeitpunkt in der Laufzeit vom Garbage-Collector entfernt, da dieser Code nicht mehr verwendet werden kann und den Speicher unnötig blockiert.

Für den Fall, dass eine Übernahme derselben Werte gewünscht ist, muss entweder eine Kopie des konkreten Objekts erstellt werden oder die Werte manuell angepasst werden.

Für die Prüfung auf Gleichheit wird grundsätzlich die Methode equals() verwendet. Diese wird für eine eigne Klasse ggf. überschrieben und vergleicht entsprechende Attributwerte. Mit == wird, also z.B. farmer1 == farmer2 wird Referenzgleichheit geprüft. Diese ist nur gleich, wenn es sich um dasselbe Objekt handelt. Für primitive Datentypen hingegen funktioniert diese Operation.

4 Vererbung und Polymorphie

Vererbung (Inheritance) in Java bedeutet, dass eine Klasse (Unterklasse) Eigenschaften und Verhalten einer bereits existierenden Klasse (Oberklasse) erbt. Das bedeutet, dass eine Unterklasse zunächst alle Attribute und Methoden einer Oberklasse übernimmt geregelt durch den Zugriff ggf. bestimmte Veränderungen oder weitere Eigenschaften und Funktionen ergänzen kann. Die Unterklasse ist in dieser Hinsicht eine Art der Verfeinerung, die ein Problem auf einer unteren Ebene aufgreift. Durch dieses Prinzip lassen sich Gemeinsamkeiten zwischen Klassen herausarbeiten, sodass redundante Codepassagen minimiert werden und Änderungen an einem Ort gebündelt auftreten. Bei der Implementation einer Unterklasse muss zunächst immer der Konstruktor der Oberklasse aufgerufen werden. Dies geschieht über den Aufruf super() im Konstruktor der Unterklasse, bei dem alle für die Oberklasse benötigten Parameter übergeben werden müssen.
In Java existiert eine Oberklasse Object von der jede weitere Klasse automatisch erbt. Polymorphie (Polymorphism) beschreibt die Fähigkeit, dass verschiedene Klassen dieselbe Methode unterschiedlich implementieren können (Override). Die Hierarchie der Aufrufe arbeitet dabei von unten nach oben, das bedeutet welche Funktionalität beim Aufruf der gleichnamigen Methode umgesetzt wird hängt davon ab, welcher Klasse das konkrete angehört.

Im Folgenden ein Beispielprogramm:
Hier wird eine Baumstruktur mittels Knoten implementiert. Es existiert eine abstrakte Klasse Node von der die drei Unterklassen InnerNode, SingleNode und Leaf erben. Abstrakt bedeutet, dass von dieser Klasse ohne weiteres keine Objekte erzeugt werden können. Des Weiteren wird eine Methode doSomething implementiert. Diese ist in Node ebenfalls als abstrakt implementiert. Das bedeutet, dass diese Methode zwangsläufig in den Unterklassen mittels @Override implementiert werden muss. In diesem Fall realisiert diese Methode eine Traversierung durch den Baum, bei dem die Zahlenwerte der Teilbäume sowie der des aktuellen Knoten addiert und auf der Konsole ausgegeben werden. Für die Klassen InnerNode sowie SingelNode zieht dies einen rekursiven Aufruf mit sich. Für die Klasse Leaf kann der Wert direkt zurückgegeben werden. Eine Beispielaufruf wird in der Klasse Graph implementiert.

```
public abstract class Node {
    private Node left;
    private Node right;
    protected int value;
    public Node (Node left, Node right, int value) {
        this.left = left;
        this.right = right;
        this.value = value;
    }
    public int getValue() {
        return value;
    }
    public Node getLeft() {
        return left;
    }
    public Node getRight() {
        return right;
    }
    public abstract int doSomething();
}


public class InnerNode extends Node {

    public InnerNode(Node left, Node right, int value) {
        super(left, right, value);
    }

    @Override
    public int doSomething() {
        return           this.getLeft().doSomething()           +
this.getRight().doSomething() + this.value;
    }
}

public class SingleNode extends Node {

    public SingleNode(Node left, int value) {
        super(left, null, value);
    }

    @Override
    public int doSomething() {
        return this.getLeft().doSomething() + this.value;
    }
}

public class Leaf extends Node {

    public Leaf(int value) {
        super(null, null, value);
    }

    @Override
    public int doSomething() {
        return this.value;
    }
```

```
}

package Graph;

public class Graph {

    public Node root;
    public Node n1;
    public Node n2;
    public Node n3;
    public Node n4;
    public Node n5;

    public Graph() {
        this.n4 = new Leaf(4);
        this.n5 = new Leaf(1);
        this.n3 = new Leaf(6);
        this.n1 = new InnerNode(n3, n4, 8);
        this.n2 = new SingleNode(n5, 7);
        this.root = new InnerNode(n1, n2, 3);
    }

    public static void main (String[] args) {
        Graph g = new Graph();
        System.out.println(g.root.doSomething());
        System.out.println(g.n1.doSomething());
        System.out.println(g.n4.doSomething());
    }
}

Ausgabe von main() in Graph:
  29
  18
  4
```

4.1 Interfaces

Ein Interface ist eine Schnittstelle, die festlegt, welche Funktionen eine Klasse bieten muss, ohne zu spezifizieren, wie diese Funktionen umgesetzt werden. Dazu werden abstrakte Methoden und Konstanten definiert, welche wiederum von der implementierenden Klasse implementiert werden müssen. Eine Klasse kann mehrere Interfaces implementieren, wodurch sie die in den Interfaces festgelegten Methoden implementieren muss. Dies ermöglicht Mehrfachvererbung von Fähigkeiten und die Abstraktion von Implementierungsdetails, sodass Klassen eine bestimmte Funktionalität oder eine bestimmte Eigenschaft aufweisen müssen, ohne dass die konkrete Umsetzung vorgegeben ist.

Beispielprogramm:

Es gibt die Klassen Wasserflugzeug, Auto, Boot sowie die Interfaces Fliegen, Schwimmen, Fahren. Über die Implementierung dieser Interfaces in den zuvor genannten Klasse kann also festgelegt werden, welche Klasse über welche Eigenschaften bzw. Fähigkeiten verfügt.

```
public interface Fahren {}
public interface Schwimmen {}
public interface Fliegen {}
public class Auto implements Fahren {}
public class Wasserflugzeug implements Schwimmen, Fliegen {}
public class Boot implements Schwimmen {}
```

5 Typecasting

Im Zusammenhang mit Vererbung spielt Typecasting eine wichtige Rolle. Analog zum Kapitel Datentypen wird beim Typecasting die Interpretation des Datensatzes verändert. Auf diese Weise kann ein Objekt in einen anderen Typ „umgewandelt“ werden. Im Rahmen der Vererbung besteht in Java eine Vererbungshierarchie, die auf diese Weise manipuliert werden kann. Es wird hierbei zwischen zwei grundlegenden Fällen unterschieden; Upcasting und Downcasting. Analog zu den vorgestellten Fällen funktioniert Typecasting grundsätzlich mit beliebigen Datentypen bzw. Klassen sofern sich diese logisch in das Programm einbinden lassen.

Upcasting beschreibt die Umwandlung von einer Unterklasse zur Oberklasse. Dieser Vorgang passiert meist automatisch und ist immer sicher, da jede Instanz der Unterklasse auch Instanz der Oberklasse ist.

Beispiel: `Node n = new Leaf(5);`

Downcasting beschreibt die Umwandlung von einer Oberklasse zur Unterklasse. Diese muss explizit erfolgen, da nicht jede Oberklassenreferenz tatsächlich ein Objekt der Unterklasse darstellt. Wird auf ein inkompatibles Objekt gecastet, entsteht eine ClassCastException. Eine sichere Prüfung zu Laufzeit erfolgt über den instanceof-Operator. Mit diesem kann geprüft werden, ob ein Konkreter Datensatz Instanz einer bestimmten Klasse ist bzw. ob das Typecasting ohne Fehler erfolgen kann.

Typecasting wird insbesondere dann notwendig, wenn über eine Oberklassenreferenz auf Methoden zugegriffen wird, die nur in einer speziellen Unterklasse existieren.

Das folgende Beispiel erweitert die Klasse Leaf um eine Methode printLeafInfo(), die ausschließlich für Blätter verfügbar ist:

```
public void printLeafInfo() {
        System.out.println("Leaf-Knoten mit Wert: " + this.value);
    }
```

In der main-Methode von Graph ergibt sich folgender Ablauf:

```
public static void main (String[] args) {
    Graph g = new Graph();
    Node someNode = g.n4;          // Upcasting: Leaf → Node
(automatisch)
    // allgemeiner Zugriff möglich
    System.out.println(someNode.doSomething());
    // spezieller Zugriff auf Leaf-Methode erfordert Downcasting
    if (someNode instanceof Leaf) {
        Leaf leafNode = (Leaf) someNode;  // explizites Downcasting
        leafNode.printLeafInfo();
    }
}
```

6 Generics

Generics in Java sind ein Sprachmittel, die es erlauben, Klassen, Interfaces und Methoden so zu definieren, dass sie mit **Typparametern** arbeiten können. Das bedeutet, dass sich eine Klasse oder Methode nicht auf einen konkreten Datentyp festlegen muss, sondern flexibel mit unterschiedlichen Typen genutzt werden kann. Auf diese Weise wird **Typsicherheit** gewährleistet, da zur Kompilierzeit geprüft wird, ob die Typen korrekt verwendet werden. Gleichzeitig entfällt die Notwendigkeit von Casts, was den Code übersichtlicher und weniger fehleranfällig macht.

Ein klassisches Beispiel ist die Java-Klasse ArrayList<T>. Der Platzhalter <T> steht für einen beliebigen Typ, der beim Anlegen einer Liste spezifiziert wird. Dadurch ist sichergestellt, dass nur Objekte dieses Typs in die Liste eingefügt werden dürfen.

Neben Klassen können auch Methoden generisch sein, wenn sie einen Typen nur innerhalb der Methode benötigen. Des Weiteren lassen sich Generics durch Bounds einschränken (z. B. `<T extends Number>`), sodass nur bestimmte Typen erlaubt sind.

Beispielprogramm:

```
// Generische Box-Klasse
public class Box<T> {
    private T content;

    public void setContent(T content) {
        this.content = content;
    }

    public T getContent() {
        return content;
    }
}
// Generische Methode
public class GenericMethodDemo {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }
}

// Testklasse
public class GenericTest {
    public static void main(String[] args) {
        // Beispiel 1: Box
        Box<String> stringBox = new Box<>();
        stringBox.setContent("Hallo Generics");
        System.out.println(stringBox.getContent());

        Box<Integer> intBox = new Box<>();
```

```
        intBox.setContent(42);
        System.out.println(intBox.getContent());

        // Beispiel 2: Generische Methode
        Integer[] zahlen = {1, 2, 3, 4};
        String[] worte = {"A", "B", "C"};

        GenericMethodDemo.printArray(zahlen);
        GenericMethodDemo.printArray(worte);
    }
}

Ausgabe
Hallo Generics
42
1 2 3 4
A B C
```

7 Patterns

Ein Pattern ist im abstrakten Sinne eine Art Blaupause, die ein Programm um eine bestimmte Funktionalität erweitert. Es kann sowohl als Erzeugungsmuster, als auch als strukturelles Entwurfsmuster in Erscheinung treten. Ein Pattern bietet also eine Lösung in einer Form, die es ermöglicht, diese Lösung unzählige Male anzuwenden, ohne sie zweimal auf genau dieselbe Weise umzusetzen. In dieser Hinsicht gibt es keine universalen festen Standards, aber durchaus wiederkehrende Konstrukte, von denen einige im Folgenden erläutert werden.

7.1 Factory Method

Die Factory Method ist ein Erzeugungsmuster, mit dem die Erzeugung von Objekten gekapselt werden kann, um die Abhängigkeit von einem konkreten Klassentypen zu vermeiden. D.h. statt den Aufruf `new` direkt im Code zu verwenden, wird eine Fabrikmethode definiert, die konkrete Objekte erzeugen kann. Erzeugt eine Klasse eine neue Instanz mittels `new` wird sie fest an eine konkrete Implementierung gebunden. Durch die Nutzung einer Factory Methode kann die Objekterzeugung und Nutzung sauber voneinander getrennt werden.

7.2 Composite Pattern

Das Composite Pattern ist ein strukturelles Entwurfsmuster, das eine einheitliche Behandlung von Objektgruppen ermöglicht. Dieses Verfahren eignet sich besonders gut bei hierarchischen Strukturen wie Bäumen. Die einzelnen Elemente (Blätter) sowie Zwischenkomponenten (Knoten) werden dabei über ein gemeinsames Interface

angesprochen. Die Abarbeitung bzw. Traversierung erfolgt dann vollständig rekursiv, wobei nicht zwischen Blatt und Knoten unterschieden werden muss.
Das gemeinsame Interface beschreibt die Operationen, die auf beiden Typen erlaubt ist. Die Blatt-Klasse implementiert diese Operation direkt, die Composite-Klasse (hier Knoten) indem die Operation rekursiv an die Kindknoten weitergegeben werden.
Beispielprogramm:

```
public abstract class Node {
    protected int value;

    public Node(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
    // zentrale Operation (z. B. Berechnung, Traversierung)
    public abstract int doSomething();
}

public class Leaf extends Node {

    public Leaf(int value) {
        super(value);
    }

    @Override
    public int doSomething() {
        return this.value;
    }
}

public class CompositeNode extends Node {

    private List<Node> children = new ArrayList<>();

    public CompositeNode(int value) {
        super(value);
    }

    public void add(Node child) {
        children.add(child);
    }

    @Override
    public int doSomething() {
        int sum = this.value;
        for (Node child : children) {
            sum += child.doSomething();
        }
        return sum;
    }
}
```

```
public class Graph {

    public static void main(String[] args) {
        Leaf l1 = new Leaf(4);
        Leaf l2 = new Leaf(1);
        Leaf l3 = new Leaf(6);

        CompositeNode c1 = new CompositeNode(8);
        c1.add(l3);
        c1.add(l1);

        CompositeNode c2 = new CompositeNode(7);
        c2.add(l2);

        CompositeNode root = new CompositeNode(3);
        root.add(c1);
        root.add(c2);

        System.out.println(root.doSomething());
        System.out.println(c1.doSomething());
        System.out.println(l1.doSomething());
    }
}

Ausgabe:
29
18
4
```

7.3 Visitor Pattern

Das Visitor Pattern ist ein Verhaltensmuster, das es ermöglicht, neue Operationen auf einer Objektstruktur zu definieren, ohne die Klassen der Elemente zu ändern. Statt die Logik in die Klassen selbst einzubauen, wird ein Besucherobjekt über die Struktur geschickt. Die Klasse muss lediglich eine universale accept-Methode bereitstellen, die sämtliche Besucherobjekte akzeptieren kann. Entsprechend ist diese Methode generisch implementiert.

Für die Implementierung wird zunächst meist ein Visitor-Interface bereitgestellt in welchem definiert wird, welche Klassen ein konkrete Besucherklasse besuchen können muss, um die Programmlogik aufrecht zu erhalten. In der jeweiligen Besucherklasse wird die Logik für die entsprechende Operation definiert. Hier stellen sowohl die Klassen CompositeNode als auch Leaf eine accept-Methode bereit. Die Traversierung der Struktur wird weiterhin durch die Struktur via accept-Methode festgelegt, allerdings keine Logische Operation wie z.B. die Aufaddierung aller Knotenwerte. Dies wird in der Klasse SumVisitor implementiert. D.h. während der

Laufzeit akzeptiert jeder Knoten den Besucher bzw. ein Objekt der Klasse SumVisitor indem auf dem Besucher die visit-Methode aufgerufen wird. Als Instanz wird das besuchte Objekt selbst übergeben; Da die Visit Methode überladen ist, wird automatisch die passende Funktion im Visitor aufgerufen und ausgeführt. Das bedeutet auch, dass zur Laufzeit konkret ein SumVisitor-Objekt für einen Operationsdurchlauf bzw. einen Traversierungsdurchlauf gekoppelt ist.
Dieses Verfahren ist besonders nützlich, wenn man viele unterschiedliche Operationen auf derselben Datenstruktur benötigt.

```
public abstract class Node {
    protected int value;
    public Node(int value) { this.value = value; }
    public abstract void accept(Visitor v);
}

public class Leaf extends Node {
    public Leaf(int value) { super(value); }
    @Override
    public void accept(Visitor v) { v.visit(this); }
}

public class CompositeNode extends Node {
    private List<Node> children = new ArrayList<>();
    public CompositeNode(int value) { super(value); }
    public void add(Node child) { children.add(child); }
    @Override
    public void accept(Visitor v) {
        v.visit(this);
        for (Node child : children) {
            child.accept(v);
        }
    }
}

public interface Visitor {
    void visit(Leaf leaf);
    void visit(CompositeNode node);
}


// Beispiel-Visitor: Berechnet Summe
public class SumVisitor implements Visitor {
    private int sum = 0;
    @Override
    public void visit(Leaf leaf) { sum += leaf.value; }
    @Override
    public void visit(CompositeNode node) { sum += node.value; }
    public int getSum() { return sum; }
}

public class Graph {
    public static void main(String[] args) {
        Leaf l1 = new Leaf(4);
```

```
        Leaf l2 = new Leaf(1);
        CompositeNode c1 = new CompositeNode(3);
        c1.add(l1);
        c1.add(l2);

        SumVisitor visitor = new SumVisitor();
        c1.accept(visitor);
        System.out.println("Summe: " + visitor.getSum());
    }
}
Ausgabe: 8
```

7.4 Iterator Pattern

Das Iterator Pattern erlaubt, eine Sammlung von Objekten zu durchlaufen, ohne deren interne Struktur offenzulegen. Damit wird die Traversierung von Datenstrukturen entkoppelt und standardisiert. Besonders wichtig ist das bei komplexeren Strukturen wie Bäumen, da die Reihenfolge (Pre-Order, In-Order, Post-Order) flexibel implementiert werden kann. Dazu würde zunächst ein Interface implementiert werden, das eine hasNext() und next() Methode bereit stellt. Die hasNext() Methode gibt an, ob es in der Datenstruktur bzw. während der Traversierung noch ein weiteres Folgeobjekt gibt, welches wiederum über next() zurückgegeben wird. Die Logik zur Iteration selbst wird intern verwaltet. In Java wird ein solches generisches Interface Iterator<> bereitgestellt. Das Gegenstück bildet die Klasse bzw. die Struktur über die Iteriert wird. Diese kann zusätzlich das von Java bereitgestellte Interface Iterable<> implementieren. Dadurch kann die herkömmliche for-each Schleife für die Iteration verwendet werden. Im folgenden Beispielcode werden beide Versionen behandelt. Die Logik zur Iteration wird in der Methode next() implementiert. Zur Verwaltung der Knoten bzw. des aktuellen Knotens wird ein Stack verwendet.

Variante 1:

```
public interface MyIterator<Node> {
    boolean hasNext();
    T next();
}

public class NodeIterator implements MyIterator<Node> {
    private Stack<Node> stack = new Stack<>();
    public NodeIterator(Node root) {
        if (root != null) stack.push(root);
    }
    @Override
    public boolean hasNext() { return !stack.isEmpty(); }
    @Override
    public Node next() {
        Node current = stack.pop();
```

```
        if (current instanceof CompositeNode) {
            CompositeNode c = (CompositeNode) current;
            List<Node>          children          =          new
ArrayList<>(c.getChildren());
            Collections.reverse(children); // richtige Reihenfolge
            for (Node child : children) {
                stack.push(child);
            }
        }
        return current;
    }
}

public class Graph {
    public static void main(String[] args) {
        Leaf l1 = new Leaf(4);
        Leaf l2 = new Leaf(1);
        CompositeNode root = new CompositeNode(3);
        root.add(l1);
        root.add(l2);

        NodeIterator it = new NodeIterator(root);
        while (it.hasNext()) {
            Node n = it.next();
            System.out.println("Node-Wert: " + n.value);
        }
    }
}
Node-Wert: 3
Node-Wert: 4
Node-Wert: 1
```

Variante 2:

```
public class NodeIterator implements Iterator<Node> {
    private Stack<Node> stack = new Stack<>();
    public NodeIterator(Node root) {
        if (root != null) stack.push(root);
    }
    @Override
    public boolean hasNext() { return !stack.isEmpty(); }
    @Override
    public Node next() {
        Node current = stack.pop();
        if (current instanceof CompositeNode) {
            CompositeNode c = (CompositeNode) current;
            for (Node child : c.children) {
                stack.push(child);
            }
        }
        return current;
    }
}

public abstract class Node implements Iterable<Node> {
    protected int value;
    public Node(int value) { this.value = value; }
```

```
    public abstract void accept(Visitor v);
    @Override
    public Iterator<Node> iterator() {
        return new NodeIterator(this);
    }
}

public class Graph {
    public static void main(String[] args) {
        Leaf l1 = new Leaf(4);
        Leaf l2 = new Leaf(1);
        CompositeNode root = new CompositeNode(3);
        root.add(l1);
        root.add(l2);

        NodeIterator it = new NodeIterator(root);
        while (it.hasNext()) {
            Node n = it.next();
            System.out.println("Node-Wert: " + n.value);
        }
    }
}
Node-Wert: 3
Node-Wert: 1
Node-Wert: 4
```

7.5 State Pattern

Das State Pattern ist ein Verhaltensmuster, das es ermöglicht, das Verhalten eines Objektes abhängig von seinem internen Zustand zu ändern. Anstatt große Verzweigungen oder switch-Konstrukte einzusetzen, wird jeder Zustand als eigene Klasse modelliert. Das Kontextobjekt delegiert seine Methodenaufrufe an das aktuell gesetzte Zustandsobjekt. Auf diese Weise bleibt das Programm strukturiert und neue Zustände können einfach durch Hinzufügen weiterer Klassen ergänzt werden, ohne die bestehende Logik verändern zu müssen. Ein klassisches Anwendungsbeispiel ist ein Automat, der je nach Zustand unterschiedlich reagiert. Entsprechend kann ein Zustandsübergangsdiagramm angefertigt werden. Das folgende Beispielprogramm verdeutlicht einen klassischen Getränkeautomaten, der vereinfacht (via Textoutput) zunächst auf eine Eingabe und eine Münze wartet und schließlich das Produkt ausgibt. Dabei führt der Automat intern lediglich die Funktion request() aus; Die Logik bzw. Ausgabe wird über den aktuellen State über die Methode handle() verwaltet. Der Wechsel erfolgt über die Methode setState(). Die States selbst sind ein Interface gebunden, das festlegt, welche Funktionen ein neuer State für die Aufrechterhaltung der Logik des Programms implementieren muss.

```
// State-Interface
public interface State {
    void handle();
}
// Konkrete Zustände
public class IdleState implements State {
    @Override
    public void handle() {
        System.out.println("Automat wartet auf Produktauswahl.");
    }
}

public class InputState implements State {
    @Override
    public void handle() {
        System.out.println("Produkt    erkannt.    Bitte    Münze
einwerfen.");
    }
}

public class DispensingState implements State {
    @Override
    public void handle() {
        System.out.println("Produkt wird ausgegeben...");
    }
}
// Kontextklasse
public class VendingMachine {
    private State state;

    public VendingMachine() {
        this.state = new IdleState(); // Startzustand
    }

    public void setState(State state) {
        this.state = state;
    }

    public void request() {
        state.handle();
    }
}
// Testprogramm
public class TestMachine {
    public static void main(String[] args) {
        VendingMachine machine = new VendingMachine();

        machine.request(); // Idle
        machine.setState(new InputState());
        machine.request(); // Coin
        machine.setState(new DispensingState());
        machine.request(); // Dispense
    }
}
Ausgabe:
Automat wartet auf Produktauswahl.
Produkt erkannt. Bitte Münze einwerfen.
```

```
Produkt wird ausgegeben...
```

Im Rahmen eines automatisierten Wechsels der States können die States auch direkt als Abstrakte Klasse innerhalb der Kontextklasse implementiert werden und ein zusätzliches Attribut hinzugefügt werden, das den aktuellen State verwaltet. Der Wechsel selbst erfolgt dann innerhalb der State Klasse über dieses Attribut.

```
public class VendingMachine {
    //Interne Klasse State
    private abstract class State {
        abstract void handle();
    }
    private final State idleState = new State() {
        @Override
        void handle() {
            System.out.println("Idle");
            state = inputState;
        }
    };
    private final State inputState = new State() {
        @Override
        void handle() {
            System.out.println("Coin");
            state = dispensingState;
        }
    };
    private final State dispensingState = new State() {
        @Override
        void handle() {
            System.out.println("Product");
            state = idleState;
        }
    };
    //Verwalten des aktuellen States
    State state = idleState;

    public void request(){
        state.handle();
    }
    //Test
    public static void main(String[] args) {
        VendingMachine vm = new VendingMachine();
        vm.request();
        vm.request();
        vm.request();
    }
}

Ausgabe:
Idle
Coin
Product
```