Mentorierte Arbeit
Fachwissenschaftliche Vertiefung mit pädagogischem Fokus Informatik A und B

Thema:
objektorientierte Programmierung (OOP)

Autor: Vincenzo Parisi

10.04.2024

Inhaltsverzeichnis

1 Konzeption

1.1 Vorwissen

Im 12. Schuljahr am Gymnasium haben die Lernenden 5 Lektionen pro Woche Informatik als Ergänzungsfach, in diesem Gefäss wird die objektorientierte Programmierung eingeführt. In den vergangenen Lektionen haben sich die Lernenden mit den Grundlagen der Programmierung befasst, dies beinhaltete vorwiegend das Konzept der Variablen, der Selektion, der Schleifen und der Funktionen. In diesem Zusammenhang wurde auch kurz das Thema Mouse-Event und Keyboard-Event angesprochen, um eine kleine Anwendung mit «gpanel» im TigerJython zu programmieren. Python ist die einzige Programmiersprache, welche die Lernenden kennengelernt haben.
Einfache Listen wurden in Python als Datenstruktur behandelt, aber mehr im Sinne eines dynamischen Arrays. Dies bedeutet, dass sie die verketteten Listen nicht kennen, was wir als Beispiel der objektorientierten Programmierung einführen werden.

1.2 Analyse des Stoffs

In dieser Arbeit wird die objektorientierte Programmierung (OOP) mit der Programmiersprache Java eingeführt. Java wird gewählt, da sie in der Praxis einen relevanten Wert hat und da in Java die OOP in meinen Augen sauber umgesetzt wird.
Ein weiteres Ziel ist die Nutzung von UML, um einen Softwareentwurf oder genauer gesagt, um die Softwarearchitektur zu erstellen, ohne vorher eine Zeile Code schreiben zu müssen. In UML werden wir aber vor allem die Klassendiagramme benutzen, um die Abhängigkeiten bei der Vererbung oder der Aggregation darzustellen. Weitere Diagramme werden nicht behandelt.

Dies ergibt folgenden Inhalten:

- Die Konzepte und Strukturen von Python in Java einführen
- Idee der OOP, Darstellung der Beziehungen mit UML
- Klassen und Objekte mit Überladen und Überschreiben von Methoden
- Vererbung und Polymorphismus
- OOP-Anwendungen: einfach verkettete Liste für eine Datenverwaltung

Abgrenzungen:
Bei der OOP sind Interface und abstrakte Klasse ein wichtiges Konzept. Da diese Themen sehr umfangreich sind und somit den Rahmen dieser Arbeit sprengen würden, werden sie trotz der Wichtigkeit weggelassen. Ein weiteres wichtiges Thema für die Strukturierung der Klassen sind Pakete (`package`), dieses Thema wird auch nicht behandelt werden. Die Lernenden sollen einen Einblick in die Konzepte der OOP erhalten, damit sie im weiteren Studium einen einfachen Einstieg haben werden.

Die Analyse der Fachliteratur zu OOP führt zu der folgender Concept Map:

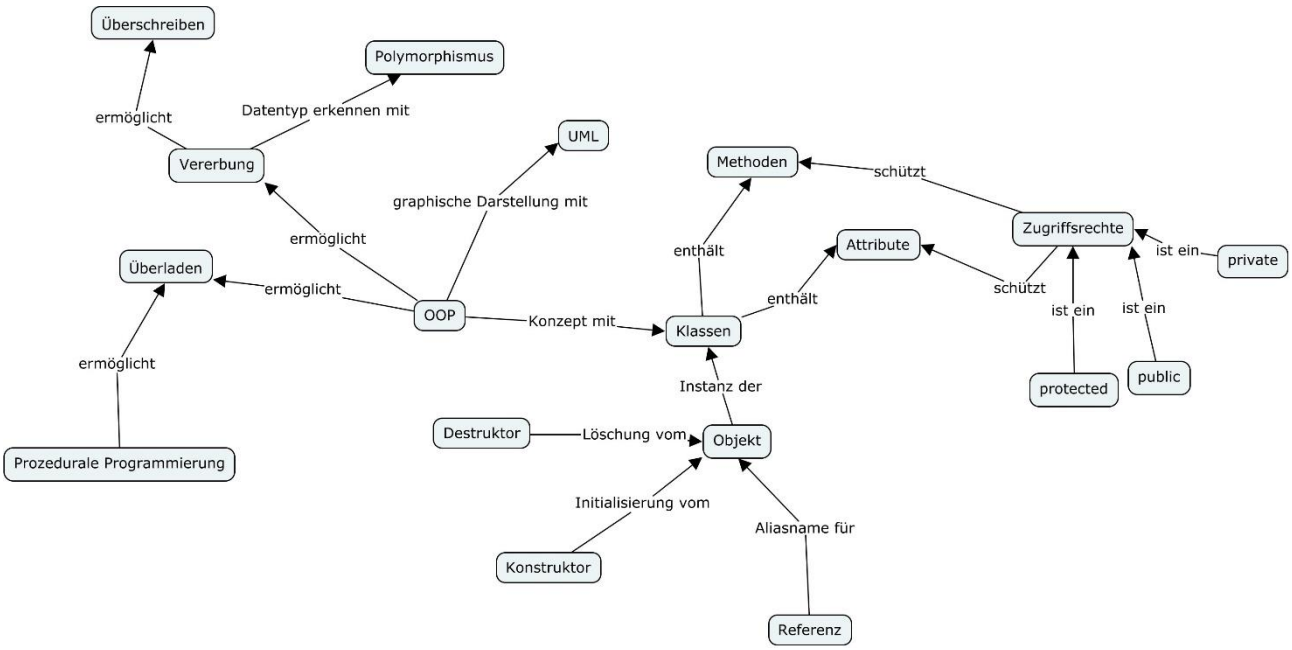

Abbildung 1: Concept Map OOP

1.3 Neue Begriffe

In diesem Abschnitt werden die Begriffe aus der Concept Map (Abbildung 1) als Einstieg kurz erklärt. Die ausführlichen Erklärungen erfolgen im Laufe der Arbeit in den entsprechenden Kapiteln.

1.3.1 OOP

Objektorientierte Programmierung hat als Idee, dass man Daten, welche zusammengehören als Einheit halten kann. Diese Sammlung von Daten entspricht der Struktur des Objektes.
Als Daten können Standarddatentypen oder wiederum andere Objekte verwendet werden. Ein Objekt ist somit eine Verallgemeinerung einer Variablen, um Informationen zu speichern. Der Zustand des Objektes, d.h. die Daten, welche im Speicher abgelegt sind, entspricht dann dem Variablenwert.

1.3.2 Klassen und Objekte

Der Unterschied von Klassen und Objekt verwirrt zu Beginn die Lernenden, deshalb werden diese Begriffe sehr genau eingeführt. Es soll klar gezeigt werden, dass die Klasse eine Vorlage für die Instanziierung eines Objektes ist.

1.3.3 Methode

Zu jedem Standarddatentyp wie z.B. Integer (`int`) oder Gleitkommazahl (`double`), gehört eine Menge von Operationen. Die Methoden beschreiben die Operationen für eine Klasse, welche einen neuen Datentyp definiert. Mit diesen Methoden ist es möglich über die Argumente den Zustand des Objektes zu verändern.

1.3.4 Attribute

Mit den Attributenwerten kann der Zustand eines Objektes gespeichert werden. Methoden und Attribute werden auch sehr genau behandelt, da diese einen entscheidenden Teil des Konzeptes der OOP sind.

1.3.5 Vererbung

Das Prinzip der Vererbung ist die Wiederverwendung eines Code- oder Programmfragmentes, ohne diesen zu kopieren. Damit kann erreicht werden, was wir im Ansatz bei Funktionen bereits gesehen haben, nämlich dass doppelten Code vermieden werden soll.

1.3.6 Überladen

Mit dem Konzept des Überladens kann eine neue Definition einer Methode mit demselben Namen implementiert werden, jedoch müssen die Parameter variieren. Je nach Anzahl oder Typ der Parameter, wird die entsprechende Methode ausgewählt.

1.3.7 Polymorphismus / Überschreiben

Polymorphismus wird in dieser Arbeit nicht stark vertieft, wir werden aber aufzeigen, dass der Datentyp eines Objektes für den Zugriff auf die Methoden und Attributen erst zur Laufzeit ermittelt wird. In diesem Zusammenhang wird dann auch das Überschreiben einer Methode eingeführt. Beide Konzepte sind grosse Stärken der OOP, aber für eine ausführliche Behandlung zu umfangreich für diese Arbeit.

1.4 Ziele

1.4.1 Leitidee

Beim objektorientierten Ansatz geht es darum, zusammen gehörende Informationen in einem Objekt zu halten. Man könnte dies auch verwenden, um eine Abstraktion oder ein Abbild eines realen Objektes im Speicher zu halten. Als Beispiel könnte man sich die Funktionalität eines Weckers vorstellen. Beim Softwareentwurf wird eine Klasse Wecker mit der gesamten Funktionalität und der gesamten Informationen erstellt. Die Daten, welche den Zustand des Weckers beschreiben sind die Tageszeit und die Weckzeit, welche wiederum Objekten sind. Die Zeiten bestehen aus Stunden, Minuten und Sekunden. Die Methoden der Klasse dienen dazu die Uhrzeit anzuzeigen, den Wecker zu stellen, den Alarm auszulösen und das Licht einzuschalten. Sowie auch Methoden, um die Zeiten zu setzen, also kann damit der Zustand des Objektes im Speicher verändert werden. Im Gegensatz dazu wäre der prozedurale Weg diese Funktionen zu schreiben, welche nacheinander je nach Anforderung aufgerufen werden. Der OOP-Ansatz ermöglicht dann weiter einen Wecker mit zwei Zeitzonen zu erstellen, was mit Vererbung ohne grossen Aufwand möglich ist. (Dies wird aber später genauer erklärt.) In der Softwareentwicklung ist diese Art von Programmierung weit verbreitet und deshalb sollte es als erweiterte Grundlage vermittelt werden.
Aus dieser kurzen Einführung ergibt sich folgende **Leitidee**:
Der Softwareentwurf OOP kann auf verschiedenen Aufgabenstellungen angewendet werden und der Lernende kennt die Vorteile der OOP im Vergleich zur prozeduralen Programmierung.

1.4.2 Dispositionsziel

- Im Anschluss an diese Unterrichtseinheit werden die Lernenden, bevor sie direkt mit dem Programmieren beginnen, sich überlegen, wie die Zusammenhänge der Anforderungen sind, damit sie sich einen OOP-Ansatz für die Lösung überlegen können.
- Beim OOP-Ansatz werden die Lernenden versuchen die Gemeinsamkeiten in einer Basisklasse zu vereinigen, um danach die spezialisierten Klassen für die spezifischen Anforderungen zu entwickeln.

1.4.3 Operationalisierte Ziele

- Die Lernenden können den Unterschied zwischen einer prozeduralen Funktion und einer objektorientierten Methode erklären, ohne dabei die Unterlagen zu verwenden.
- Sie kennen die Vor- und Nachteile von OOP und können sie einem Kollegen mit Beispielen wiedergeben, ohne dabei im Skript nachzulesen. Vor allem können sie den Sinn einer Klasse genau erklären.
- Der Einsatz von OOP können sie begründen, indem sie aufzeigen, dass die Klasse und somit auch das Objekt eine gewisse «Intelligenz» in Form von Operationen hat und deshalb es selbst weiss, welche Aufgabe es erledigen kann.
- Weiter kennen die Lernenden die Programmiersprache Java und können diese für viele Anwendungen einsetzen.

2 Von Python zu Java

Es wird vorausgesetzt, dass eine Entwicklungsumgebung für die Programmiersprache Java bereits installiert ist. Geeignet sind Eclipse oder VisualCode. Ferner muss beachtet werden, dass eine aktuelle Java Umgebung (jdk: java development kit) installiert ist, damit die Beispiele fehlerfrei gestartet werden können.

In diesem Kapitel werden v.a. die Strukturen von Python in Java beschrieben. Es sind nicht vollständige Programme, da Java dafür eine Klasse benötigt, was wir zu diesem Zeitpunkt noch nicht kennen.

2.1 Einführung in Java

Ein wichtiger Unterschied ist, dass in Java jede Anweisung mit einem «;» Semikolon abgeschlossen werden muss. Weiter muss der Code, welcher zu einer Struktur gehört, z.B. zu **`if`** oder zu **`while`**, nicht eingerückt werden. Er wird mit einem Block markiert, welcher mit geschweiften Klammern umrandet ist.

Kommentare
In Java können Kommentare mit **`//`** markiert werden, z.B.: **`//Das ist ein Kommentar`**.
Falls man mehrere Zeilen als Block auskommentieren will, kann der Blockkommentar **`/*`** … **`*/`** benutzt werden, z.B.:

```
/* erste Zeile des Kommentars
Zweite Zeile des Kommentars
Weitere Zeilen */
```

Kommentare werden im Editor der Entwicklungsumgebung meistens grün angezeigt.

Wir starten gleich mit der Ausgabe eines Textes.

Beispiel 1

Python	Java
`print("Hello World!")`	`System.out.println("Hello World!");`

Nun werden wir die Konzepte der Variablen und der Strukturen (**`if`**… **`else`**, logische Verknüpfungen, **`while`** und **`for`**) aus dem Python Unterricht in die Programmiersprache Java portieren.

2.1.1 Variablen

Eine Variable in Java bedeutet dasselbe wie in Python. Der einzige Unterschied ist, dass der Interpreter nicht selbständig einen Datentyp einer Variable zuordnet, sondern man muss den Datentyp angeben. Will man eine Variable für eine ganze Zahl erstellen, so muss **`int a;`** geschrieben werden, wobei **`int`** für Integer (ganze Zahl) steht. Diese Angabe des Datentyps wird nur bei der Initiierung der Variable angegeben, danach ist die Variable **`a`** im gesamten Code als Integer definiert und muss nicht bei einer weiteren Verwendung wieder angegeben werden (siehe Beispiel 2).

Beispiel 2

Wir möchten eine ganze Zahl ausgeben.

Python	Java

`a = 5`
`print(a)`
`a = a + 2`
	`int a = 5;`
	`System.out.println(a);`
	`a = a + 2 ;`

In diesem Beispiel ist die Variable **a**, wie anfangs bereits erklärt, eine ganze Zahl, dies entspricht dem Datentyp Integer, deshalb steht im obigen Java Code **int a**. Die Idee eines Datentyps ist, dass im Speicher genügend Platz für die Speicherung der Variable reserviert wird, dass der Wertebereich der Variable **a** definiert ist und dass der Datentyp vom Compiler überprüft werden kann. Zudem bietet diese Notation eine gewisse Robustheit gegenüber Fehler, denn so können der Variable **a** nur ganze Zahlen zugewiesen werden, ohne dass der Compiler einen Fehler markiert. Gewollte Typumwandlungen sind möglich, dies wird im Beispiel 3 gezeigt, aber der Programmieren muss wissen, was er tut. Weiter ist in diesem Beispiel ersichtlich, dass einen weiteren Umgang mit der Variable **a** keine erneute Angabe des Datentyps erfordert.
Beachten Sie, wie bereits erwähnt, dass in Java jede Anweisung mit einem « **;** » (Semikolon) abgeschlossen wird.

Für die Auswahl des Datentyps, muss man sich vorher überlegen, welchen Datentyp für eine Anwendung geeignet ist, da der Datentyp den Wertebereich bestimmt (Hromkovič, Gallenbacher, Komm, Lacher, & Pierhöfer, 2023, S. 24) und dieser im Nachhinein nicht verändert werden kann. Dadurch ist auch die Grösse des benötigten Speichers definiert. In der untenstehenden Tabelle (Krüger, 2000) sind die häufigsten Datentypen mit dem Wertebereich angegeben.

Datentyp	Länge in Byte	Wertebereich	Standardwert
`boolean`	1	`true, false`	`false`
`char`	2	Alle Unicode-Zeichen	\u0000
`byte`	1	$-2^7 \ldots 2^7-1$	0
`short`	2	$-2^{15} \ldots 2^{15}-1$	0
`int`	4	$-2^{31} \ldots 2^{31}-1$	0
`long`	8	$-2^{63} \ldots 2^{63}-1$	0
`float`	4	$+/-3.40282347 * 10^{38}$	0.0
`double`	8	$+/-1.79769313486231570 * 10^{308}$	0.0

Tabelle 1: Datentypen

Der Standardwert (letzte Spalte in der obigen Tabelle) wird vergeben, falls die Variable nicht mit einem anderen Wert initialisiert wird.
Ein sogenannter Typecast bewirkt, dass bei einer Variable eines bestimmten Datentyps einen Wert oder Variable eines anderen Datentyps zugewiesen werden kann. Falls man einen **double** aus einer Methode als Rückgabewert erhält und man sich sicher ist, dass einen **float** genügen würde, dann kann der Datentyp umgewandelt werden. Beachten Sie aber, dass der Wert als **double** gekürzt wird. Von einem «kleineren» Datentyp wandelt Java den Typ automatisch um, z.B. **int** zu **double**, da es in diesem Fall keinen Datenverlust zur Folge hat. Im folgenden Beispiel werden verschiedene Typecast gezeigt und dessen Auswirkungen, falls man von einem «grösseren» Datentyp in einen «kleineren» umwandelt.

Beispiel 3

Anhand dieses Beispiels wird die Problematik einer Typumwandlung gezeigt.

```
//automatische Datenkonvertierung
System.out.println("automatischer Typecast ohne Datenverlust:");
short a = 5;
int b = 13;
System.out.println(a);
System.out.println(b);
b = a;
System.out.println(b);
```

Obiger Code ergibt die untenstehende Ausgabe, man sieht, dass es keinen Datenverlust gibt, bei der Zuweisung **`b = a`**, hat die Variable **`b`** denselben Wert wie **`a`**, da **`a`** der kleinere Datentyp ist. Mit kleineren Datentyp ist gemeint, dass der Wertebereich des Datentyps kleiner ist.

Ausgabe:

```
automatischer Typecast ohne Datenverlust:
5
13
5
```

```
//Typecast mit Datenverlust
byte c = 16;
a = 30000;
// Typecast
System.out.println("erzwungener Typecast mit Datenverlust:");
System.out.println(c);
c = (byte)a;
System.out.println(a);
System.out.println(c);
```

Nach Ausführung des zweiten Teils erscheint die folgende Ausgabe:

```
erzwungener Typecast mit Datenverlust:
16
30000
48
```

Im zweiten Teil mit einem erzwungenen Typecast, **`c = (byte)a`**, steht in der Variable **`c`** nicht derselbe Wert wie in der Variable **`a`**, da die Variable **`c`** vom Typ **`byte`** ist, und der maximale Wert von **`byte`** 127 ist, kann in der Variable **`c`** den Wert 30'000 der Variable **`a`** vom Typ **`short`** nicht gespeichert werden. Bei dieser Umwandlung werden die oberen Bits einfach abgeschnitten.

Was auf der Bit-Ebene geschieht, werden wir als kleinen Einschub darstellen, binäre Zahlen sind aber nicht das Thema dieser Arbeit.

Wert: 30'000	`short` (16 Bit)	`byte` (8 Bit)
Als Binäre Zahl:	0111 0101 **0011 0000**	**0011 0000**
Als Dezimalzahl	30'000	48

Aufgabe 1

Gegeben ist folgender Codeausschnitt.

```
int a = 10990;
long b = 200223;
System.out.println(a);
System.out.println(b);
b = a;
System.out.println(b);
```

Bestimmen Sie welche Ausgabe die obigen Zeilen generieren.

Lösung:
Es gibt keinen Datenverlust, da `long` ein grösserer Datentyp als `int` ist, somit ist die Ausgabe:

```
10990
200223
10990
```

Aufgabe 2

Ergänzen im folgenden Codeausschnitt die Datentypen, damit es die angegebene Ausgabe erzeugt. Erklären Sie wie es zu dieser Ausgabe kommt.

```
???? c = 0;
???? s = 350;
???? = (????)????;
System.out.println(s);
System.out.println(c);
```

Ausgabe:

```
350
94
```

Lösung:

```
byte c = 0;
short s = 350;
c = (byte)s;
System.out.println(s);
System.out.println(c);
```

350 wird einer Variable **c** zugewiesen, dessen Wert danach kleiner ist, d.h. es wurde eine Typumwandlung erzwungen mit Datenverlust. Die ausgegebenen Werte können in der Darstellung mit binären Zahlen erklärt werden.

Wert: 350	**short** (16 Bit)	**byte** (8 Bit)
Als Binäre Zahl:	0000 0001 **0101 1110**	**0101 1110**
Als Dezimalzahl	350	94

Aufgabe 3

Sie sollen ohne arithmetische Operationen nur mit einer Typumwandlung in einem Codeausschnitt die folgenden Ausgaben erzeugen bei gegebenen Eingaben.

Eingabe:
98596
Ausgabe:
36

Eingabe:
12.84
Ausgabe:
12

Lösung:
36 ist kleiner als 127, somit könnte der Datentyp **byte** sein.

```
long l = 98596;
byte b = (byte)l;
System.out.println(b);
```

Die Kommastellen werden abgeschnitten, d.h. es liegt einen Datentyp einer ganzen Zahl vor, also **byte**, **short**, **int**, **long** wären alles korrekte Antworten.

```
double d = 12.84;
int i = (int)d;
System.out.println(i);
```

Aufgabe 4

Schreiben Sie einen Ausschnitt eines Java-Programmes, welches die Lösung einer quadratische Gleichung

$$a \cdot x^2 + b \cdot x + c = 0$$

berechnet.
Geben Sie die Koeffizienten **a**, **b** und **c** zu Beginn als fixe Werte ein, einmal als ganze Zahlen, z.B. **a = 3**, **b = 3** und **c = -3** und einmal als Kommazahlen, z.B. **a = 1.4**, **b = 2.4** und **c = -0.4** an. Berechnen Sie die Nullstellen mit der bekannten Formel:

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4 * a * c}}{2 * a}$$

Achten Sie dabei auf einen geeigneten Datentyp bei der Variablendeklaration.

Hinweis:
Die Wurzel kann mit **Math.sqrt(b*b - 4*a*c)** berechnet werden. Passen Sie auf, dass mit **** in Java nicht das Potenzieren gemeint ist! In Java müssen Sie a^b mit **Math.pow(a, b)** berechnen.

Lösung:

```
/* int a = 3;
int b = 3;
int c = -3; */

double a = 1.4;
double b = 2.4;
double c = -0.4;

double res;
//b^2 ist bitweise xor!
res = -b + Math.sqrt(b*b - 4*a*c) / (2*a);
System.out.println(res);
res = -b - Math.sqrt(b*b - 4*a*c) / (2*a);
System.out.println(res);
```

Aufgabe 5

Gegeben ist folgendes Programmfragment:

```
int i_a = 10;
int i_b = 3;
int i_res = i_a / i_b;
System.out.println(i_res);

double d_a = 10;
double d_b = 3;
double d_res = d_a / d_b;
System.out.println(d_res);
```

Gehen Sie die Zeilen im Kopf durch und beurteilen Sie die zwei Lösungen.

Lösung:
Bei der ersten Division wird eine ganzzahlige Division berechnet, bei der zweiten eine exakte Division. Da bei der ersten Varianten die zwei Variablen **`i_a`** und **`i_b`** vom Datentyp Integer sind, berechnet Java immer eine ganzzahlige Division und das Resultat ist wiederum eine ganze Zahl.

Aufgabe 6

Ändern Sie nun die obige Aufgabe, sodass die erste Division mit zwei **`float`** Werten gerechnet wird. Welchen Unterschied beim Resultat erwarten Sie?

Lösung:

```
float f_a = 10;
float f_b = 3;
float f_res = f_a / f_b;
System.out.println(f_res);

double d_a = 10;
double d_b = 3;
double d_res = d_a / d_b;
System.out.println(d_res);
```

Erwartung:
3.3333333
3.3333333333333335
Mit dem Datentyp **`double`** werden mehrere Stellen angezeigt, da **`double`** eine grössere Genauigkeit hat. Der Datentyp **`double`** hat 15 signifikante Stellen und der Datentyp **`float`** nur 7.

2.2 Selektion (if...else)

Die Selektion ist in Java fast gleich wie in Python, deshalb wird weiter unten direkt ein Beispiele gezeigt. Die allgemeine Struktur der Selektion in Python und in Java ist in der untenstehenden Tabelle dargestellt:

Python	Java

`if Bedingung:`
`  Anweisung`
`elif Bedingung:`
`  Anweisung`
`else:`
`  Anweisung`
	`if (Bedingung){`
	`  Anweisung;`
	`}else if (Bedingung){`
	`  Anweisung;`
	`}else{`
	`  Anweisung;`
	`}`

Anstatt des Doppelpunktes nach der Bedingung steht in Java eine geschweifte Klammer (**{**). Die öffnende geschweiften Klammern definieren in Java den Beginn eines Blockes und die schliessende (**}**) das Ende des Blockes. Würde man keinen Block verwenden, so würde nur die erste Anweisung der Selektion zugeordnet werden. Mit einem Block können mehrere Befehle einer Struktur zusammengefasst werden. Wir werden in dieser Arbeit immer Blöcke (**{**…**}**) verwenden und empfehlen auch danach mit Blöcken zu arbeiten, damit es keine Verwirrungen gibt. Die Einrückungen spielen in Java nur für die visuelle Darstellung eine Rolle, bei der Übersetzung des Codes werden sie nicht berücksichtigt.

Aufgabe 7

Erarbeiten Sie die Unterschiede der Variante in Python und in Java.

Lösung:
Python benötigt keine runden Klammern für die Bedingung. Dafür wird der Abschluss des `if`'s mit einem Doppelpunkt markiert und die Anweisungen, welche zum `if` gehören sind eingerückt. Bei Java ist die Bedingung in Klammern und die Anweisungen sind in einem Block bestehend aus geschweiften Klammern (**{ }**). Dies gilt auch für `else if` und `else`.
In Python ist `elif` das Schlüsselwort für ein «sonst falls» und in Java `else if`.

Beispiel 4

Wir werden in diesem Beispiel prüfen, ob ein Jahr ein Schaltjahr ist.
Die Regel für die Berechnung lautet:

1. Ist die Jahreszahl durch vier teilbar, aber nicht durch 100, ist es ein Schaltjahr.
2. Ist die Jahreszahl durch 100 teilbar, aber nicht durch 400, ist es kein Schaltjahr.
3. Ist die Jahreszahl durch 400 teilbar, dann ist es ein Schaltjahr.

Die Lösung für diese Problem in **Python (TigerJython)**:

```
year = 2100

if year % 400 == 0:
    print(str(year) + " ist ein Schaltjahr")
elif year % 100 == 0:
    print(str(year) + " ist kein Schaltjahr")
elif year % 4 == 0:
    print(str(year) + " ist ein Schaltjahr")
else:
    print(str(year) + " ist kein Schaltjahr")
```

Der relevante Code für die Lösung in **Java** wird unten gezeigt.

```
int year = 2100;

if (year % 400 == 0){
    System.out.println(year + " ist ein Schaltjahr");
} else if (year % 100 == 0){
    System.out.println(year + " ist kein Schaltjahr");
} else if (year % 4 == 0){
    System.out.println(year + " ist ein Schaltjahr");
} else{
    System.out.println(year + " ist kein Schaltjahr");
}
```

Aufgabe 8

In der Aufgabe 4 haben wir eine quadratische Gleichung gelöst. Die Zahlen waren vorgegeben, damit es sicher zwei Lösungen gab. Nun sollen Sie den obigen Programmausschnitt erweitern, sodass Sie aufgrund der Diskriminante herausfinden, wie viele Lösungen die Gleichung hat.
Es gilt:

$$D = b^2 - 4 * a * c$$

$D > 0$: es existieren zwei Lösungen
$D = 0$: es existiert nur eine Lösung
$D < 0$: es existiert keine Lösung
Erst danach berechnen Sie die Lösungen, falls es welche gibt.

Lösung:

```
//Koeffizienten für zwei Lösungen
/* int a = 3;
int b = 3;
int c = -3; */

//Koeffizienten für eine Lösung
/* int a = 1;
int b = 4;
int c = 4; */
```

```
//Koeffizienten für keine Lösung
int a = 2;
int b = 4;
int c = 4;

/* double a = 1.4;
double b = 2.4;
double c = -0.4; */

double res;
//b^2 ist xor!
double D = b*b - 4*a*c;
if (D == 0){
    System.out.println("Es existiert nur eine Loesung");
    res = -b;
    System.out.println(res);
}else if (D > 0){
    System.out.println("Es existieren zwei Loesungen");
    res = -b + Math.sqrt(D) / (2*a);
    System.out.println(res);
    res = -b - Math.sqrt(D) / (2*a);
    System.out.println(res);
}else{
   System.out.println("Es existieren keine Loesungen");
}
```

2.3 Fallunterscheidung (switch)

Eine Fallunterscheidung mit dem Schlüsselwort **`switch`** gibt es in Python nicht, dies wird in Python mit mehreren **`elif`**'s gelöst. Betrachten wir folgendes Beispiel in Java.

Beispiel 5

Wir möchten eine Ziffern zwischen 0 und 9 als Text ausgeben. Ein Variante ist die Aufgabe, wie wir es in Python gewohnt sind, mit **`else if`** (Python: **`elif`**) zu lösen.
Anbei ist der Java Code aufgelistet:

```
int ziffer = 3;

if (ziffer == 0){
    System.out.println("Null");
}else if (ziffer == 1){
    System.out.println("Eins");
}else if (ziffer == 2){
    System.out.println("Zwei");
}else if (ziffer == 3){
    System.out.println("Drei");
}else if (ziffer == 4){
    System.out.println("Vier");
}else if (ziffer == 5){
    System.out.println("Fünf");
}else if (ziffer == 6){
    System.out.println("Sechs");
}else if (ziffer == 7){
    System.out.println("Sieben");
}else if (ziffer == 8){
    System.out.println("Acht");
}else if (ziffer == 9){
    System.out.println("Neun");
}else{
    System.out.println("falsche Ziffer");
}
```

Betrachtet man den Code, so fällt auf, dass wir immer dieselbe Variable **`ziffer`** testen. In dieser Situation können wir in Java eine andere Struktur anwenden, nämlich **`switch`** … **`case`**. Eine allgemeine Darstellung der Struktur wird folgend dargestellt.

```
switch(var){
case 0:
     Anweisung;
     break;
case 1:
     Anweisung;
     break;
   :
   :
   :
default:
     Anweisung;
}
```

Dabei wird die Variable **var** mit allen **case** Werten verglichen und der Code im zutreffenden **case** wird bis zum **break** ausgeführt. Würde man **break** weggelassen, so werden alle folgenden Anweisungen auch ausgeführt, bis ein anderer **break** folgt oder bis der **switch**-Block beendet wird. Meistens will man aber nur die Anweisungen des zutreffenden **case** ausführen, in diesem Fall ist der **break** erforderlich.

Beispiel 6

Dieselbe Aufgabe wie im Beispiel 5, lösen wir nun mit einem **switch**.

```
int ziffer = 5;

switch (ziffer){
case 0:
    System.out.println("Null");
    break;
case 1:
    System.out.println("Eins");
    break;
case 2:
    System.out.println("Zwei");
    break;
case 3:
    System.out.println("Drei");
    break;
case 4:
    System.out.println("Vier");
    break;
case 5:
    System.out.println("Fünf");
    break;
case 6:
    System.out.println("Sechs");
    break;
```

```
case 7:
    System.out.println("Sieben");
    break;
case 8:
    System.out.println("Acht");
    break;
case 9:
    System.out.println("Neun");
    break;
default:
    System.out.println("falsche Ziffer!");
}
```

Aufgabe 9

Schreiben Sie ein Programmfragment, welches ihnen den Monat als Zahl in einen Text umwandelt. Bei der Zahl 1 wird «Januar» ausgegeben usw. Die Eingabe soll wiederum direkt im Code als fixen Wert angegeben werden.

Lösung:

```
int month = 10;
switch (month){
case 1:
    System.out.println("Januar");
    break;
case 2:
    System.out.println("Februar");
    break;
case 3:
    System.out.println("März");
    break;
case 4:
    System.out.println("April");
    break;
case 5:
    System.out.println("Mai");
    break;
case 6:
    System.out.println("Juni");
    break;
case 7:
    System.out.println("Juli");
    break;
case 8:
    System.out.println("August");
    break;
case 9:
    System.out.println("September");
    break;
```

```
case 10:
    System.out.println("Oktober");
    break;
case 11:
    System.out.println("November");
    break;
case 12:
    System.out.println("Dezember");
    break;
default:
    System.out.println("falscher Monat!");
}
```

2.4 Relationale Operatoren

Relationale Operatoren werden auch Vergleichsoperatoren genannt, sie sind in der untenstehenden Tabelle aufgeführt. Dabei ist ersichtlich, dass es sich um dieselben Operatoren wie in Python handelt.

Operator	**Beschreibung**	**Beispiel a=5; b=10**	**Resultat**
<	kleiner als	a < b	**true**
<=	kleiner gleich	a <= b	**true**
>	grösser als	a > b	**false**
>=	grösser gleich	a >= b	**false**
!=	ungleich	a != b	**true**
==	gleich	a == b	**false**

Aufgabe 10

Ermitteln Sie die Ergebnisse der Vergleiche in der folgenden Tabelle.

a	**b**	**Vergleich**	**Resultat**
5	6	a < b	
10	10	a > b	
10	10	a <= b	
5	5	a != b	
23	23	a >= b	
5	7	a != b	
15	15	a == b	

Lösung:

a	**b**	**Vergleich**	**Resultat**
5	6	a < b	**true**
10	10	a > b	**false**
10	10	a <= b	**true**
5	5	a != b	**false**
23	23	a >= b	**true**
5	7	a != b	**true**
15	15	a == b	**true**

Aufgabe 11

Schreiben Sie einen Programmausschnitt, welches eine ganzzahlige Division auf zwei ganze Zahlen ausführt, dabei wird die erste Zahl durch die zweite Zahl dividiert. Achten Sie darauf, dass eine Division durch 0 nicht erlaubt ist. In diesem Fall schreiben Sie eine Fehlermeldung auf die Konsole und führen die Berechnung nicht aus.

Lösung:

```
//erste Zahl
int a = 5;

//zweite Zahl
int b = 8;
if (b == 0){
    System.out.println("Division durch Null ist nicht erlaubt.");
}else{
    System.out.println(a + " / " + b + " = " + a/b);
}
```

2.5 Logische Operatoren

Um mehrere Bedingungen zu verknüpfen, benötigen wir logische Operatoren. Bitte beachten Sie, dass die bitweise Operationen nicht zum Inhalt dieser Arbeit gehören.

2.5.1 AND (UND) Verknüpfung

In Python: `s > 10 and s < 50` und
in Java: `s > 10 && s < 50`

`s > 10`	`s < 50`	**Ergebnis** `s > 10 && s < 50`
`false`	`false`	`False`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

2.5.2 OR (ODER) Verknüpfung

In Python: `s > 10 or s < 50` und
in Java: `s > 10 || s < 50`

`s > 10`	`s < 50`	**Ergebnis** `s > 10 || s < 50`
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

2.5.4 NOT (NICHT) Verknüpfung

In Python: **not (s > 10)** und
in Java: **!(s > 10)**

s > 10	Ergebnis !(s > 10)
false	true
false	true
true	false
true	false

Beispiel 7

Die Beispiele werden ab dieser Stelle nur noch in Java geschrieben.
Das Ergebnis des Ausdruckes soll **true** sein, falls eine Zahl **z** zwischen 10 und 100 liegt:

```
z >= 10 && z <= 100
```

Das Ergebnis des Ausdruckes soll **true** sein, falls eine Zahl z ausserhalb des Intervalls zwischen 10 und 100 liegt:

1. Variante:

```
!(z >= 10 && z <= 100)
```

2. Variante

```
z < 10 || z > 100
```

Aufgabe 12

Ergänzen Sie die folgende Tabelle:

s < 40	a < 10	Operator	Ergebnis
true	false	||	
false	false	&&	
true	true	||	
true	true	&&	
false	false	||	

Lösung:

s < 40	a < 10	Operator	Ergebnis
true	false	||	true
false	false	&&	false
true	true	||	true
true	true	&&	true
false	false	||	false

Aufgabe 13

Schreiben Sie ein Programmfragment, bei dem Sie prüfen, ob eine Zahl sich in einem Bereich befindet. Ist die Zahl **nicht** zwischen 3 und 10, so erscheint einen Fehler. Ist die Zahl korrekt, so wird sie ausgegeben.

Lösung:

```
int a = 6;
if (a > 3 && a < 10){
    System.out.println("Die Zahl ist korrekt: " + a);
}else{
    System.out.println("Fehler: Die Zahl ist nicht im gewünschten
Intervall.");
}
```

2.6 while-Schleifen

Die **while**-Schleife ist in Java sehr ähnlich wie in Python, deshalb folgt nur ein Beispiel.
Die Struktur der **while**-Schleife ist:

Python	Java

`while Bedingung:`
`  Anweisungen`
	`while(Bedingung){`
	`  Anweisungen;`
	`}`

Beispiel 8

Wir möchten die Quersumme einer Zahl berechnen. Da wir nicht wissen, wie gross die Zahl ist und somit wie viele Stellen sie hat, können wir das Problem nicht mit einer **for**-Schleife lösen.
Die Zahl selbst wird zur Vereinfachung als festen Wert im Code angegeben.

```
int zahl = 341;
int tmp = zahl;
int summe = 0;
while (tmp > 0){
    summe = summe + tmp % 10;
    tmp = tmp / 10;
}
System.out.println("Die Quersumme von "+zahl+" ist "+summe);
```

Die Unterschiede der **while**-Schleife in Java zu Python sind:

1. der fehlende Doppelpunkt, dafür markieren geschweifte Klammern den Rumpf (Block) der Schleife
2. und die Bedingung steht in Klammern

Aufgabe 14

Schreiben Sie einen Code, der alle Quadratzahlen bis 1000 ausgibt.

Lösung:

```
int a = 1;
int i = 1;
while (a < 1000){
    System.out.println(a);
    i = i + 1;
    a = i * i;
}
```

2.7 for-Schleifen

In Java existieren zwei **for**-Schleifen. Die Zählschleife, mit der eine bestimmt Anzahl Wiederholungen angegeben werden kann und die zweite ist eine sog. «for-each» -Schleife, das heisst, dass z.B. für jedes Element in einer Liste der Schleifenrumpf ausgeführt wird.
Die **for**-Schleife in Python entspricht eher einer «for-each»-Schleife, welche in Java wie im nächsten Beispiel gezeigt ist, implementiert werden kann. Das Array entspricht hier fast einer Liste in Python. Das Thema Array wird in dieser Arbeit im Kapitel 6.1 «Array» behandelt.

Im Beispiel 9 wird zuerst die Python ähnliche **for**-Schleife gezeigt und erst in einem weiteren Beispiel die zweite Variante.

Python	Java

`for element in arr:`
`    Anweisungen`
	`for (int element : arr){`
	`    Anweisungen;`
	`}`

Beispiel 9

```
// array declaration
int arr[] = { 10, 50, 60, 80, 90 };

for (int element : arr){
    System.out.print(element + " ");
}
```

Vielfach wird in Java die Zählschleife verwendet, welche wesentlich anders wie in Python funktioniert. Betrachten wir zuerst die Struktur dieser **for**-Schleife:

```
for(Initialisierung; Bedingung; Inkrement){
     Anweisungen
}
```

Die Initialisierung wird nur einmal beim Eintritt in die Schleife ausgeführt, nur falls die Bedingung erfüllt ist, werden die Anweisungen im Schleifenrumpf ausgeführt und zuletzt wird der Inkrement-Befehl ausgeführt.

Beispiel 10

Wir zeigen die Anwendung dieser Variante der **for**-Schleife an einem einfachen Beispiel, bei dem die Zahlen 1 bis 10 ausgegeben werden.

```
for (int i = 1; i <= 10;i++){
    System.out.println(i);
}
```

Daraus ist ersichtlich, dass die Initialisierung **int i = 1**, die Bedingung **i < 10** und das Inkrement **i++** ist. **i++** ist dasselbe wie **i = i + 1** oder **i += 1**. Sobald die Bedingung nicht erfüllt ist, bricht die Schliefe ab.

Aufgabe 15

Gegeben sei eine Zahl. Geben Sie auf der Konsole so viele Sterne (*) aus, wie der Wert der Zahl ist.
Hinweis: Mit **System.out.print()** wird in der Ausgabe keine neue Zeile erzeugt.

Lösung:

```
int n =7;
for (int i = 0; i < n; i++){
    System.out.print("*");
}
```

2.8 String

Mit **String** sind in Java Zeichenketten gemeint, also einen Text. Wir werden in diesem Skript den Datentyp **String** verwenden und möchten ihn deshalb sehr kurz vorstellen, damit die Beispiele verstanden werden.

```
String a = "Das ist ein String.";
String b = a;
b = "Nun wird der String überschrieben"
```

Der Inhalt der obigen drei Zeilen ist alles, was wir in dieser Arbeit benötigen werden.

3 Klassen und Objekte

3.1 Datentypen und Operationen

Wir haben im Kapitel 2.1.1 «Variablen» bereits über die verschiedenen Datentypen gesprochen. Auf diese Datentypen können Operationen ausgeführt werden, z.B. können ganze Zahlen (`int`) addiert (+), subtrahiert (-), multipliziert (*) und dividiert (/) werden. Anders gesagt, jedem Datentyp gehört eine Menge von Operationen, welche darauf angewendet werden können. Zudem existieren Funktionen aus Bibliotheken z.B. aus der Mathematik-Bibliothek (Math) mit denen man weitere Operationen ausführen kann, z.B. Wurzelziehen (`Math.sqrt(a)`), Potenzieren (`Math.pow(a, b)`) usw.

Beispiel 11

Angenommen wir möchten den Abstand zwischen zwei Punkte im Koordinaten System (x, y) berechnen, so können wir den Startpunkt mit zwei Variablen (`xStart, yStart`) definieren. Analog definieren wir den Endpunkt (`xEnd, yEnd`) und können den Abstand mit folgender Formel berechnen:

$$d = \sqrt{(xEnd - xStart)^2 + (yEnd - yStart)^2}$$

Das Programmfragment für die Berechnung in Java, sieht folgendermassen aus:

```
int xStart = 1;
int yStart = 2;
int xEnd = 4;
int yEnd = 5;

double len = Math.sqrt(Math.pow((xEnd - xStart),2) +
                       Math.pow((yEnd - yStart), 2));
```

Damit man sich bei der Programmierung nicht immer überlegen muss, welche Variablen zusammengehören und welche Operationen auf welchen Datentypen angewendet werden dürfen, könnte man einen eigenen Datentyp mit all seinen Operationen und Variablen als eine Einheit definieren, dafür verwenden wir den Begriff **Klasse** anstatt Einheit. Dies bedeutet, falls man mit ganzen Zahlen arbeiten möchte, verwendet man eine Klasse z.B. die Klasse **IntNumber**, in der man eine Zahl und die erlaubten Operationen definiert.

3.2 Klassen

Das Schlüsselwort, um eine Klasse zu erstellen heisst in Java **`class`**, danach folgt der Klassenname, welcher beliebig gewählt werden kann.

Beispiel 12

In diesem Beispiel definieren wir eine Klasse **`IntNumber`**, welche die oben beschriebene Klasse für eine ganze Zahl darstellt. Die Operationen wie z.B. die Addition, Subtraktion, Multiplikation und Division, usw. werden erst später in dieser Arbeit hinzugefügt.
Die Klasse enthält ein Feld (Element oder Komponente), welche eine ganze Zahl repräsentiert.

```
public class IntNumber {
    int number;
}
```

Damit haben wir einen neuen Datentyp **`IntNumber`** erstellt. Das Feld in der Klasse ist vom Typ **`int`**, da wir mit ganzen Zahlen arbeiten möchten und der Name des Feldes ist **`number`**. **`number`** wird auch **Attribut** der Klasse genannt.
Das Schlüsselwort **`public`** bedeutet, dass die Klasse von jeder anderen Klasse verwendet werden kann, die Zugriffsrechte werden weiter unten im Kapitel 3.7 «Zugriffsmodifikatoren» erklärt.

Wir haben nun mit einer Klasse einen eigenen Datentyp mit dem Namen **`IntNumber`** erstellt.

Aufgabe 16

Definieren Sie eine Klasse **`DecimalNumber`** für das Arbeiten mit einer Gleitkommazahl.

Lösung:

```
public class DecimalNumber {
    double decNumber;
}
```

Betrachten wir wieder die Klasse **`IntNumber`**, so fehlen die Operationen, wir können nicht mit der Zahl **`number`** der Klasse **`IntNumber`** arbeiten resp. rechnen. Dazu benötigen wir sog. Methoden, welche die Operationen beschreiben. Die Methoden werden erst im Kapitel 3.5 «Methoden» eingeführt, deshalb werden wir vorläufig direkt mit den Punkt-Operator auf die Felder einer Klasse zugreifen, damit wir bereits mit den bekannten Operatoren rechnen können.

3.3 Objekte

Möchten wir mit einer Klassen analog zu einem Datentyp arbeiten, so müssen wir zuerst ein Objekt erzeugen. Ein Objekt entspricht einer Variable. Eine Variable beinhaltet einen bestimmten Standarddatentyp, während ein Objekt aus einer Klasse erzeugt wird, was einem eigenen Datentyp entspricht. Somit ist die Klasse der Bauplan oder die Vorlage, wie das Objekt erstellt werden soll.

Objekte sind eine Art Verallgemeinerung der Variablen, die Programmzeile

```
int x = 3;
```

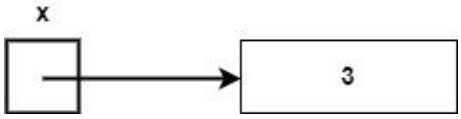

Abbildung 2: Integer-Variable im Speicher

erzeugt eine Variable, welche eine Speicherzelle referenziert, in der der Wert 3 steht (Abbildung 2). Also eine Art Verweis auf den Wert 3 im Speicher. Bei einem Objekt ist es analog.

Die Zeile:

```
IntNumber myNumber;
```

definiert ein Objekt **myNumber**.
Anders als bei den Grunddatentypen wie **int**, **double** usw. wird hier keinen Speicher für die Attribute der Klasse **IntNumber** reserviert, sondern nur für die Objektvariable, resp. das Objekt. Das Objekt **myNumber** zeigt nach der obigen Zeile auf einen ungültigen Speicherbereich, nämlich auf **null**, da noch keine Instanz der Klasse erzeugt wurde (Abbildung 3).

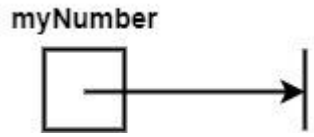

Abbildung 3: Objekt myNumber

Die Instanziierung der Klasse **IntNumber** wird mit dem **new**-Operator programmiert, erst danach zeigt das Objekt **myNumber** auf einen Speicherbereich und wir können mit den Feldern arbeiten (Abbildung 4).
Die folgende Zeile erzeugt einen Speicherbereich, auf den das Objekt **myNumber** zeigt.

```
myNumber = new IntNumber();
```

In diesem Speicherbereich können wir nun den Wert des Attributes **number** speichern, dies entspricht nun dem **Zustand** des Objektes im Speicher.
Bei der Speicherung einer Variable (**x**) liegt der Variablenwert (**3**) im Speicher (Abbildung 2). Bei einem Objekt wird der Zustand des Objektes gespeichert. Der Zustand von Objekten mit Standarddatentypen wird durch die Werten aller Attributen dieser Komponenten beschrieben. In unserem Fall wird der Zustand des Objektes **myNumber** durch den Wert des Attributes **number** definiert (Abbildung 4). Das Objekt selbst referenziert diesen Zustand im Speicher, deshalb spricht

man auch von Referenzen. Eine Referenz ist ein Verweis auf einen Speicherbereich analog zu einem Objekt, also ist es dasselbe.

Anders gesagt, alle Attributenwerten eines Objektes speichern den Zustand des Objektes ab, welcher mit Methoden (oder direkt mit dem Punkt-Operator) verändert oder gelesen werden kann.

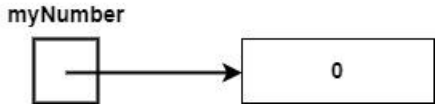

Abbildung 4:Das Objekt myNumber zeigt auf einen Speicherbereich.

Standardmässig initialisiert Java **`int`**-Werte mit **`0`**. In der untenstehenden Tabelle finden Sie die restlichen Default-Werten bei einer Initialisierung in Java:

Typ	Default-Wert
boolean	false
char	"\u0000"
byte	0
short	0
int	0
long	0
float	0.0
double	0.0

Tabelle 2: Default Werte bei der Initialisierung

Nach dem Instanziieren des Objektes **`myNumber`** enthält somit das Attribut **`number`** den Wert **`0`**. Mit dem Objekt **`myNumber`** und mit dem **`.`**-Operator (Punkt-Operator) kann das Feld **`number`** danach bearbeitet werden. Die Zeile

```
myNumber.number = 3;
```

weist den Wert 3 dem Attribut **`number`** zu.

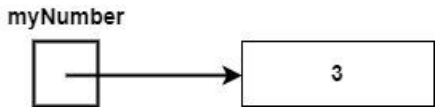

Abbildung 5: Neuer Zustand von myNumber im Speicher

Dadurch haben wir einen neuen Zustand des Objektes **`myNumber`** im Speicher.

Betrachten wir, bevor wir weitere Überlegungen anstellen, den Fall bei der Zuweisung einer Integer-Variable zu einer anderen Integer-Variable:

```
int x = 3;
int y;
y = x
```

Hier wird zuerst eine Variable **`y`** erzeugt und dann den Wert von **`x`** kopiert, **`y`** zeigt auf eine andere Speicherzelle als **`x`** (Abbildung 6).

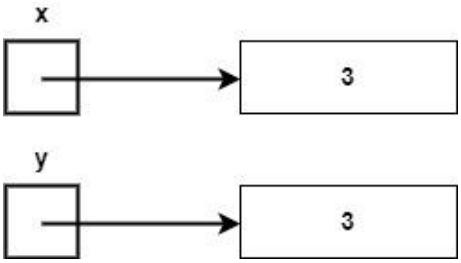

Abbildung 6: Kopie der Variable x nach y

Leicht unterschiedlich zu den Variablen eines Grunddatentyps verhalten sich die Objekten.

```
IntNumber myNumber1 = new IntNumber();
IntNumber myNumber2 = myNumber1;
```

Bei der Zuweisung **`myNumber2 = myNumber1`** muss man vorsichtig sein, denn das Objekt **`myNumber2`** verweist nun auf denselben Speicherbereich wie **`myNumber1`**. Alle Änderungen, welche über **`myNumber2`** gemacht werden, beeinflussen auch das Objekt **`myNumber1`** (Abbildung 7). Beide Objekten haben denselben Zustand! Es wird keine neue Instanz der Klasse in einem eigenen Speicherbereich erzeugt, sondern nur ein Objekt **`myNumber2`**, welches auf denselben Speicherbereich wie **`myNumber1`** zeigt.

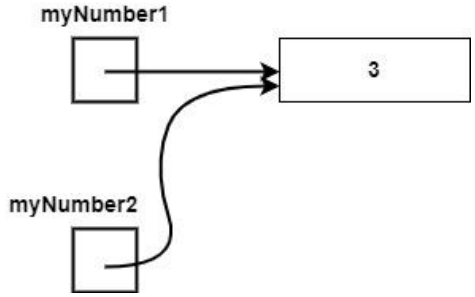

Abbildung 7: Objekt myNumber2 zeigt auf denselben Speicherbereich wie myNumber1

In diesem Fall würde die Zeile

```
myNumber2.number = 5;
```

den Zustand des Objektes **`myNumber1`** ändern (Abbildung 8), was bei einer echten Kopie nicht möglich wäre.

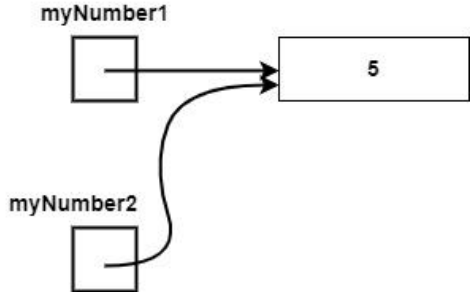

Abbildung 8: Neuer Zustand über Objekt myNumber2

Möchte man eine Kopie des Zustandes des Objektes **myNumber1** erhalten, so muss zuerst für das Objekt **myNumber2** einen eigenen Speicherbereich für seinen Zustand erzeugt werden und danach müssen die Werte einzeln kopiert werden (Abbildung 9). Mit der Notation **myNumber1.number** erhält man vom Objekt **myNumber1** das Feld **number**. In dem wir **myNumber2.number = myNumber1.number** schreiben, können wir den Wert **number** des Objektes **myNumber1** dem Wert **number** des Objektes **myNumber2** zuweisen.

```
myNumber2 = new IntNumber();
myNumber2.number = myNumber1.number;
```

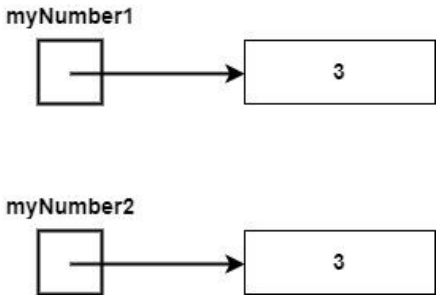

Abbildung 9: Zwei Speicherbereiche mit demselben Zustand

Nun haben wir zwei Objekte mit demselben Zustand an zwei verschiedenen Speicherbereichen.

Bemerkung:
In der Literatur findet man häufig die Bezeichnung «eine Referenz auf ein Objekt», was nicht ganz korrekt ist, aber damit ist dasselbe wie oben beschrieben gemeint, das Objekt ist die Referenz und verweist auf einen Speicherbereich mit seinem Zustand.

Aufgabe 17

Erzeugen Sie zwei Objekte der Klasse **DecimalNumber** aus der Aufgabe 16 für zwei unterschiedliche Dezimalzahlen. Das erste Objekt soll den Wert der Zahl **π** (3.14) und das zweite Objekt den Wert der Eulerzahl **e** (2.71) je auf zwei Kommastellen gerundet enthalten.

Lösung:

```
DecimalNumber pi = new DecimalNumber();
DecimalNumber e = new DecimalNumber();

pi.decNumber = 3.14;
e.decNumber = 2.71;
```

Wir haben nun einen neuen Datentyp durch das Definieren einer Klasse erzeugt, damit können wir vorläufig die bekannten Operatoren des Datentyps **int** und **double** verwenden, um die Zustände der Objekten zu verrechnen. Dies wird im folgenden Beispiel gezeigt.

Beispiel 13

In diesem Beispiel berechnen wir den Umfang eines Kreises mit Radius **`r = 3.5`**. Für den Radius verwenden wir die Klasse **`DecimalNumber`**. Wir betrachten vorerst nur den Codeausschnitt, ohne ihn ausführen zu können, um zu zeigen, wie wir bis zu diesem Zeitpunkt mit unseren Objekten rechnen können.

```
DecimalNumber pi = new DecimalNumber();
DecimalNumber r = new DecimalNumber();
DecimalNumber u = new DecimalNumber();
r.decNumber = 3.5;
pi.decNumber = 3.14;

u.decNumber = 2*r.decNumber * pi.decNumber;
System.out.println("Der Umfang ist: " + u.decNumber);
```

3.4 Wozu Klassen?

Der interessierte Leser, frägt sich nun, weshalb wir einen neuen Datentyp erstellt haben, der nicht mehr als ein Grunddatentyp kann. Diese Bemerkung ist berechtigt, und wir möchten deshalb zeigen, welche Ergänzungen die Vorteile einer Klasse mit sich bringen.

Dazu betrachten wir das Beispiel 11 nochmals, da wurde der Abstand zwischen zwei Punkte gerechnet. Da die Variablennamen gut gewählt waren, war es leicht im Programm festzustellen, welche Werte zum Startpunkt und welche zum Endpunkt gehören. Wäre dies nicht der Fall, wird es schwieriger sein die Zugehörigkeit zu bestimmen. Genau diese Zugehörigkeit der Daten zu einer Einheit ist der Vorteil einer Klasse, welche wir nun im folgenden Beispiel darstellen werden.

Beispiel 14

Wir definieren nun eine Klasse **`Point`**, welche einen Punkt in der Ebene darstellt.

```
public class Point {
    double x;
    double y;
}
```

Und rechnen damit den Abstand zwischen zwei Punkte.

```
Point startPoint = new Point();
Point endPoint = new Point();

startPoint.x = 1;
startPoint.y = 2;
endPoint.x = 4;
endPoint.y = 5;

double len = Math.sqrt(Math.pow((endPoint.x - startPoint.x),2) +
             Math.pow((endPoint.y - startPoint.y), 2));
System.out.println(len);
```

Nun ist ersichtlich, welche Koordinaten zu welchen Punkt gehören.
Das Objekt **`startPoint`** referenziert nun einen Speicherbereich mit seinem Zustand, was hier den Koordinaten des Punktes entsprechen (Abbildung 10).

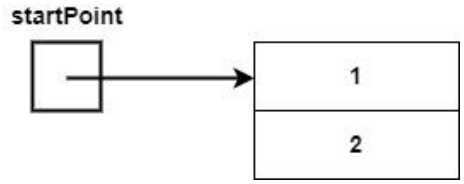

Abbildung 10: Objekt startPoint

Aufgabe 18

Erstellen Sie eine Klasse **`Date`**, um ein Datum zu speichern. Ein Datum besteht aus einem Tag von Typ Integer (**`int`**), Monat vom Typ **`String`** und einem Jahr vom Typ Integer.

Lösung:

```
public class Date {
    int day;
    String month;
    int year;
}
```

Aufgabe 19

Erzeugen Sie zwei Objekte der Klasse **`Date`** aus der Aufgabe 18 für zwei unterschiedliche Daten. Dem ersten Objekt setzen Sie das Datum ihres Geburtstages und dem zweiten Objekt das heutige Datum.
Erstellen Sie eine Skizze der zwei Objekten mit deren Zustände im Speicher.

Lösung:

```
Date date1 = new Date();
Date date2 = new Date();

date1.day = 11;
date1.month = "September";
date1.year = 1969;

date2.day = 29;
date2.month = "Dezember";
date2.year = 2023;
```

Skizze:

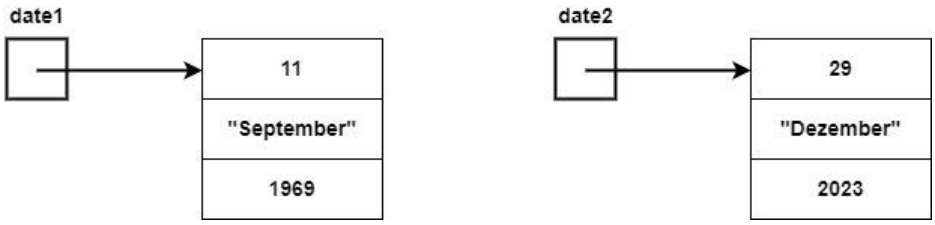

Abbildung 11: Objekt date1 und date2 mit den Zuständen im Speicher

Bis hier haben wir Klassen mit Attributen von Standarddatentypen erzeugt. Da die Klasse selbst auch ein Datentyp ist, könnten wir auch ein Attribut mit den neu erzeugten Datentypen, also unseren Klassen, definieren. In diesem Fall resultieren Felder, welche Objekte einer Klasse sind. Dazu betrachten wir folgendes Beispiel.

Beispiel 15

Wir erstellen die Klasse **`Point3d`** und analog dazu eine Klasse **`Vector`**.

```
public class Point3d {
    double x;
    double y;
    double z;
}
```

```
public class Vector {
   double x;
   double y;
   double z;
}
```

In der Physik wird die Kraft an einem Angriffspunkt mit einem Vektor für die Richtung der Kraft in diesem Punkt modelliert. Dies bedeutet für uns, wir können eine Klasse Kraft (Force) erstellen, welche einen Punkt und einen Vektor enthält. Dies ist in der Klasse **`Force`** mit den Attributen **`p`** vom Typ **`Point3d`** und **`dir`** vom Typ **`Vector`** implementiert.

```
public class Force {
  Point3d p;
  Vector dir;
}
```

Wir haben nun hier einen verschachtelten Datentyp erstellt, welcher uns ermöglicht eine Kraft als einen neuen Datentyp zu verwenden. Diese Struktur enthält alle Elementen, welche zu einer Kraft gehören. Diese Zusammengehörigkeit von Informationen ist ein weiterer Vorteil einer Klasse und ermöglicht auch einen modularen Aufbau eines neuen Datentyps resp. einer Klasse.

Möchten wir nun die Schwerkraft modellieren, welche auf eine Kugel an der Position (0.0, 5.0, 10.0) wirkt (Abbildung 12), so können wir die Klasse **`Force`** dafür verwenden.

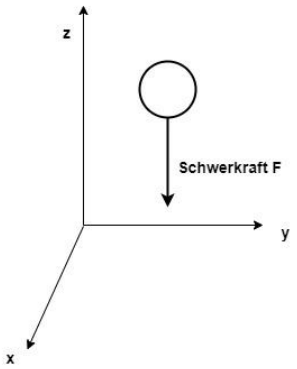

Abbildung 12: Krafteinwirkung auf eine Kugel

Der untenstehenden Codeausschnitt zeigt diese Anwendung.

```
Force F = new Force();
Point3d p = new Point3d();
p.x = 0;
p.y = 5;
p.z = 10;
Vector dir = new Vector();
dir.x = 0;
dir.y = 0;
dir.z = -1;
F.p = p;
F.dir = dir;
```

Die Objekten und die Zugehörigkeiten ist in der Abbildung 13 dargestellt. Der Aufbau ist modular, d.h. wir haben ein Objekt **`p`** für das Abbild eines Punktes und ein Objekt **`dir`** für das Abbild eines Richtungsvektor. Beide zusammen sind im Objekt **`F`** zusammengetragen, sodass die Zusammengehörigkeit gegeben ist.

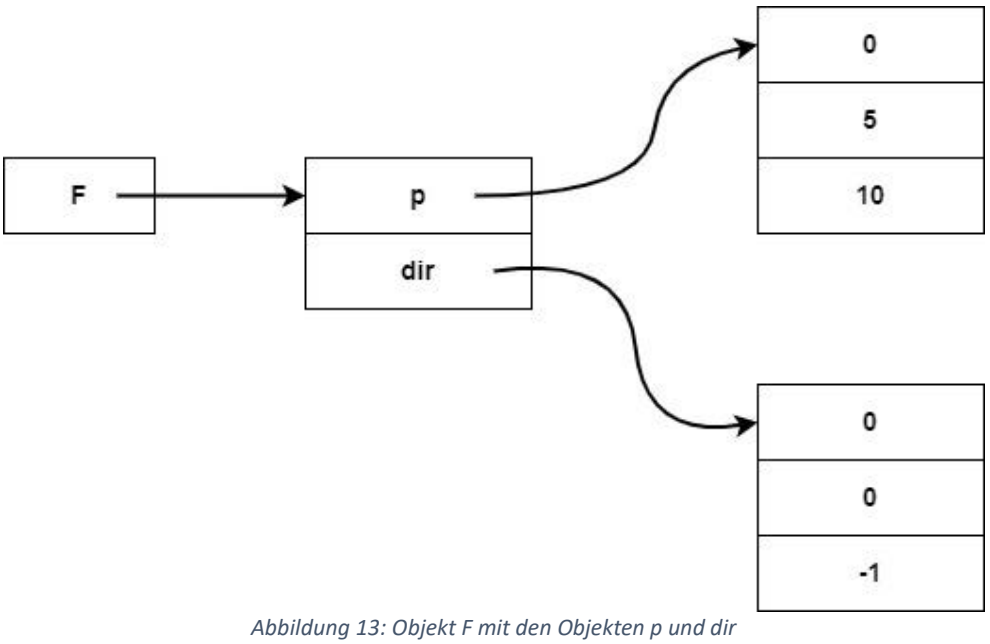

Abbildung 13: Objekt F mit den Objekten p und dir

Aufgabe 20

Erstellen Sie nun eine Klasse **`Line`**, welche einen Start- und Endpunkt der Klasse **`Point`** hat.
Danach erstellen Sie ein Objekt der Klasse **`Line`** mit dem Startpunkt bei (0, 0) und den Endpunkt bei (10, 10).
Visualisieren Sie die Objekten mit einer Skizze.

Lösung:

```
public class Line {
    Point start;
    Point end;
}
```

```
Line line = new Line();
Point startPoint = new Point();
Point endPoint = new Point();

line.start = startPoint;
line.end = endPoint;
```

Skizze:

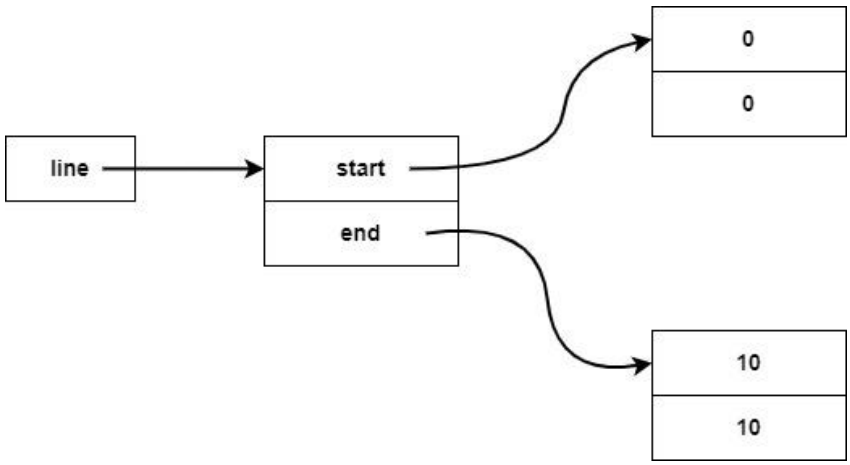

Abbildung 14: Linie mit Start- und Endpunkt

3.5 Methoden

In diesem Kapitel betrachten wir Methoden, um eine Klasse mit Operationen zu erweitern. Solche Erweiterungen können den Zustand des Objektes im Speicher verändern oder Informationen über den Zustand wiedergeben. Alle Methoden einer Klasse entsprechen der Menge der Operationen auf diesen neuen Datentyp, resp. auf diese Klasse. Wir beginnen mit den Methoden ohne Rückgabewert und ohne Parameter.

3.5.1 Ohne Rückgabewert und ohne Parameter

Die Definition einer Methode:

```
modifier void methodeName(){
}
```

modifier ist das Zugriffsrecht, welcher später in diesem Skript im Detail behandelt wird. Wir werden bis zur Behandlung der Zugriffsrechte den Zugriffsmodifikator weglassen.
void: Dies ist der Typ des Rückgabewertes, da wir nichts zurückgeben, schreiben wir einfach **void**.

Somit gilt für uns vorläufig:

```
void methodeName(){
}
```

Beispiel 16

Wir möchten nun, dass die Klasse **Point** die Informationen in der Konsole ausgeben kann, dazu schreiben wir eine Methode **void print()** in der Klasse **Point**.
Da die Klasse **Point** bereits bekannt ist, wird hier nur die **print()**-Methode gezeigt.

```
void print(){
    System.out.println("Koordinaten: x = " + x + " und y = " + y);
}
```

Aufgabe 21

Schreiben Sie analog zur **print**-Methode aus dem Beispiel 16 der Klasse **Point** eine Methode **void print()** für die Ausgabe der Daten eines Objektes der Klasse **Date**.

Lösung:

```
public class Date {
    int day;
    String month;
    int year;

    void print(){
        System.out.println("Datum: " + day + ". "+ month +
                           ". " + year);
    }
}
```

3.5.2 main-Methode

Bis jetzt haben wir nur Codefragmente betrachtet. Das Problem in Java ist, dass wir für jedes Programm eine Klasse benötigen, die eine bestimmte Methode für den Einstiegspunkt besitzt.
Die Java Entwicklungsumgebung startet das Programm, indem es die Methode **`public static void main(String[] args)`** sucht und diese ausführt.
Diese Methode darf nur einmal in einer Datei vorkommen, es dürfen aber mehrere Dateien im Projekt resp. im Verzeichnis mit einer **`main`**-Methode vorhanden sein.
Wir werden erst später auf die Bedeutung der einzelnen Schlüsselwörter eingehen.

Möchten wir die Methode **`print()`** aus Beispiel 16 testen, benötigen wir somit eine **`main`**-Methode, welche wir in der Klasse **`Point`** definieren.

```
    public static void main(String[] args) {
        Point pt = new Point(3.2, 4.5);
        pt.print();
    }
```

Falls wir das Beispiel mit der Klasse **`Force`** testen möchten, müssen wir der Klasse **`Force`** auch eine **`main`**-Methode hinzufügen. Dies ist im Beispiel 17 implementiert.

Beispiel 17

```
public class Force {
  Point3d p;
  Vector dir;

  public static void main(String[] args){
    Force F = new Force();
    Point3d p = new Point3d();
    p.x = 0;
    p.y = 5;
    p.z = 10;
    Vector dir = new Vector();
    dir.x = 0;
    dir.y = 0;
    dir.z = 1;
    F.p = p;
    F.dir = dir;
  }
}
```

Bemerkungen:

- In Java darf pro File nur eine **public** Klasse enthalten sein. Will man mehrere Klassen, da sie z.B. zusammengehören, so ist die Hauptklasse **public** und die anderen default (kein Bezeichner).
- Im obigen Beispiel ist die Klasse **Point** in der Datei «**Point.java»** gespeichert. Bei den Beispielen dieser Arbeit wird pro Klasse eine Datei erstellt, so ist auch die Klasse **Line** im File «**Line.java»** enthalten.
- Zur Erinnerung: Die Elementen (oder auch Komponenten) einer Klasse nennt man **Attribute** der Klasse. In der Klasse **Force** sind **p** und **dir** zwei Attribute.
- Wir werden alle unsere Java Dateien im selben Verzeichnis speichern und werden nicht auf das Thema **package** eingehen.

3.5.3 Ohne Rückgabewert und mit Parameter

```
void methodeName(type param1, type param2, ...){
}
```

Beispiel 18

Betrachten wir wieder die Klasse **Line**. Wir möchten nun die Linie, welche in der Klasse **Line** mit Start- und Endpunkt definiert ist, um einen bestimmten Faktor in dieselbe Richtung der Linie verlängern. Da die Klasse **Line** bereits oben abgedruckt ist, schreiben wir hier nur die Methode **void scale(double factor)** für die Skalierung und die Methode **void print()** für die Ausgabe des Start- und Endpunktes auf. Die Methode **void scale(double factor)** hat einen Parameter **factor**, dies ist der Faktor, um den die Linie verlängert wird.

```
    void scale(double factor){
        //dx und dy bilden einen Vektor vom Nullpunkt
        //aus in Richtung der Line
        double dx = end.x-start.x;
        double dy = end.y-start.y;
        //Vektor skalieren
        dx = factor * dx;
        dy = factor * dy;
        //Vektor zum Startpunkt addieren gibt neuer Endpunkt
        end.x = start.x  + dx;
        end.y = start.y + dy;
    }
    void print(){
        System.out.println("Startpunkt: ");
        start.print();
        System.out.println("Endpunkt: ");
        end.print();
    }
```

Danach schreiben wir eine **main**-Methode, um unsere Klasse zu testen.

```
    public static void main(String[] args) {
        Line l2 = new Line(1.2, 3.2, 5.1, 6.3);
        l2.scale(2);
        l2.print();
    }
```

Aufgabe 22

Erweitern Sie die Klasse **Date** um ein Attribut **monthNumber** vom Typ Integer, welches den n-ten Monat darstellt und um eine Methode **monthToNumber()**, welche ihnen je nach Monatsnamen im Attribut **month** die entsprechend Zahl ins Attribut **monthNumber** schreibt, z.B. bei Februar wird die Zahl 2 in das Feld **monthNumber** geschrieben. Eine ähnliche Aufgabe haben wir in Aufgabe 9 bereits gelöst.

Lösung:

```
    void monthToNumber(){
        switch(month){
        case "Januar":
            monthNumber = 1;
            break;
        case "Februar":
            monthNumber = 2;
            break;
        case "März":
            monthNumber = 3;
            break;
        case "April":
            monthNumber = 4;
            break;
        case "Mai":
            monthNumber = 5;
            break;
        case "Juni":
            monthNumber = 6;
            break;
        case "Juli":
            monthNumber = 7;
            break;
        case "August":
            monthNumber = 8;
            break;
        case "September":
            monthNumber = 9;
            break;
        case "Oktober":
            monthNumber = 10;
            break;
```

```
        case "November":
            monthNumber = 11;
            break;
        case "Dezember":
            monthNumber = 12;
            break;
        default:
            monthNumber = -1;
        }
    }
```

Als Lösung ist nur die Methode **monthToNumber()** abgedruckt.

Aufgabe 23

Ändern Sie in der Klasse **Date** die Methode **void print()** so ab, dass Sie einen Parameter hinzufügen, mit dem Sie die Ausgabe steuern können. Die angepasste Methode hat folgende Definition: **void print(boolean format)**, ist der Parameter **format == true**, so wird der Monat als Text (30. Dezember 2023) angezeigt sonst als Zahl (30.12.2023).

Lösung:

```
    void print(boolean format){
        if (format){
            System.out.println("Datum: " + day + ". "+ month + " "
                               + year);
        }else{
            System.out.println("Datum: " + day + ". "+
                               monthNumber + ". " + year);
        }
    }
```

3.5.4 Kleiner Exkurs Kalender

In Java kann die Systemzeit relativ einfach ausgelesen werden, in dem man eine Klasse für einen Kalender aus der Java Bibliothek verwendet. Da unsere Zeit Rechnung auf den gregorianischen Kalender basiert, werden wir im Beispiel auch einen solchen verwenden.

Der gregorianische Kalender, auch bürgerlicher Kalender, ist der weltweit meistgebrauchte Kalender. Er entstand gegen Ende des 16. Jahrhunderts durch eine Reform des julianischen Kalenders. Benannt ist er nach Papst Gregor XIII., der ihn 1582 mit der päpstlichen Bulle Intergravissimas verordnete. Er löste im Laufe der Zeit sowohl den julianischen als auch zahlreiche andere Kalender ab und bildet die Basis der Datumsdarstellung nach ISO 8601.
Der gregorianische ist wie der julianische Kalender ein Sonnenkalender, jedoch mit diesem gegenüber durch Interkalation (Einschiebung des 29. Februar) verbesserter Schaltjahresregelung. Damit liegt ihm eine durchschnittliche Jahreslänge von 365,2425 Tagen zugrunde, die den etwa 365,2422 Tagen des Sonnenjahres näherkommt als noch die 365,25 Tage des julianischen Kalenders (siehe auch Tropisches Jahr und Kalenderjahr).
(Quelle: https://de.wikipedia.org/wiki/Gregorianischer_Kalender)

Beispiel 19

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TestCalendar {

    void getTime(){
        GregorianCalendar gc = new GregorianCalendar();
        int day = gc.get(Calendar.DAY_OF_MONTH);
        int month = gc.get(Calendar.MONTH) + 1;
        int year = gc.get(Calendar.YEAR);
        int hour = gc.get(Calendar.HOUR_OF_DAY);
        int min = gc.get(Calendar.MINUTE);
        int sec = gc.get(Calendar.SECOND);

        System.out.println(day + ". " + month + ". " + year);
        System.out.println(std + ":" + min + ":" + sec);
    }

    public static void main(String[] args) {
        TestCalendar t = new TestCalendar();
        t.getTime();
    }
}
```

Für den Kalender müssen die entsprechenden Klassen aus der Java Bibliothek geladen werden, dies geschieht mit den **import**'s zu Beginn des Beispiels.

Aufgabe 24

Erstellen Sie nun eine Klasse **Time** mit den Attributen für die Stunden (**hour**), Minuten (**min**) und Sekunden (**sec**). Erstellen Sie zudem zwei Methoden, eine parameterlose **void setTime()**, welche die Systemzeit setzt und eine mit Parameter **void setTime(int h, int m, int s)** für das Setzen einer beliebigen Zeit.
Als nächstes schreiben Sie eine **void print()** Methode, um die Zeit auszugeben.
Testen Sie die obigen Methoden in einer **main**-Methode.

Lösung:

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Time {
    int hour;
    int min;
    int sec;
```

```
    public void setTime(){
        GregorianCalendar gc = new GregorianCalendar();
        hour = gc.get(Calendar.HOUR_OF_DAY);
        min = gc.get(Calendar.MINUTE);
        sec = gc.get(Calendar.SECOND);
    }

    public void setTime(int h, int m, int s){
        hour = h;
        min = m;
        sec = s;
    }

    void print(){
        System.out.println(hour + ":" + min + ":" + sec);
    }

    public static void main(String[] args) {
        Time t = new Time();
        t.setTime();
        t.print();

        t.setTime(8, 33, 45);
        t.print();
    }
}
```

Aufgabe 25

Benutzen Sie den Java Kalender, um den Tag zu berechnen, an dem Sie 10'000 Tage alt werden oder wurden. Dafür können Sie im Hauptprogramm den gregorianischen Kalender verwenden. Mit den Zeilen:

```
        //Achtung month ist 0-basiert, 0: Januar
        GregorianCalendar gc = new GregorianCalendar(1969, 8, 11);
```

Erstellen Sie ein Datum für den **11.9.1969**.

Mit der **`add()`** Methode können Sie nun das neue Datum berechnen. Lesen Sie in der Java Hilfe nach, wie Sie die **`add()`** Methode benutzen können.

Lösung:

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class Aufgabe25 {
    public static void main(String[] args){
        //Achtung month ist 0-basiert, 0: Januar
        GregorianCalendar gc = new GregorianCalendar(1969, 8, 11);
        //10000 dazu rechnen
        gc.add(Calendar.DAY_OF_MONTH, 10000);
```

```
        //Ausgabe
        int day = gc.get(Calendar.DAY_OF_MONTH);
        int month = gc.get(Calendar.MONTH) + 1;
        int year = gc.get(Calendar.YEAR);
        System.out.println(day + ". " + month + ". " + year);
    }
}
```

3.5.5 Mit Rückgabewert

```
type methodeName(param1, param2, ...){
     return val
}
```

Für dieses Kapitel benutzen wir die im Beispiel 15 erstellte Klasse **`Vector`**.
Ein Vektor besteht aus den x, y und z Komponenten und wir wissen von der Mathematik her, dass man mit Vektoren rechnen kann, wie z.B. Addition, Subtraktion, Skalarprodukt und Kreuzprodukt. Damit können wir eine praktische Anwendung der Vektorrechnung mit Methoden aufzeigen, welche den verschiedenen Operationen am neuen Datentyp **`Vector`** entsprechen.

Beispiel 20

Die Klasse **`Vector`** definiert einen Vektor im Raum und enthält als Attribut die **`x`**, **`y`**, und **`z`** Komponenten.

```
public class Vector {
    double x;
    double y;
    double z;

    double length(){
        double len;
        len = Math.sqrt(Math.pow(x, 2) + Math.pow(y, 2)
                        + Math.pow(z, 2));
        return len;
    }
}
```

Mit der Methode **`length()`** wird die Länge des Vektors berechnet.
Der Rückgabewert ist vom Typ **`double`** und mit **`return len`** wird das Ergebnis, nämlich die Länge des Vektors, zurückgegeben.

Die Methode kann nun folgendermassen angewendet werden:

```
public static void main(String[] args) {
        Vector v = new Vector();
        v.x = 1;
        v.y = 1;
        v.z = 0;
        double len = v.length();
        System.out.println("Die Länge ist: " + len);
    }
```

Die Zeile **`double len = v.length()`** bedeutet, dass das Objekt **`v`** die Methode **`length()`** aufruft und der Rückgabewert wird in der Variable **`len`** gespeichert. Nach dieser Zeile ist die Länge des Vektors in der Variable **`len`** gespeichert und der Wert kann weiterverwendet werden, wie hier für die Ausgabe. Das Prinzip des Rückgabewertes ist analog zu den Funktionen in Python.

Aufgabe 26

Erstellen Sie eine Methode **`int TimeAsNumber(int format)`** in der Klasse **`Time`** aus der Aufgabe 24, welche die Zeit abhängig vom Parameter **`format`** zurückgibt. Ist **`format == 1`**, so wird die gesamte Zeit in Sekunden zurückgegeben, ist **`format == 2`** folgt die gesamte Zeit in Minuten (abgerundet) und falls **`format == 3`** ist, so wird die gesamte Zeit in Stunden (abgerundet) zurückgegeben.

Lösung:

```
    int TimeAsNumber(int format){
        int z = 0;
        if (format == 1){
            z = hour*3600 + min*60 + sec;
        }else if (format == 2){
            z = hour*60 + min;
        }else if (format == 3){
            z = hour;
        }else{
            z = -1; //Fehler!!
        }
        return z;
    }

    public static void main(String[] args) {
        Time t = new Time();
        t.print();
        int sek = t.TimeAsNumber(1);
        System.out.println("Sekunden: " + sek);
        int min = t.TimeAsNumber(2);
        System.out.println("Minuten: " + min);
        int h = t.TimeAsNumber(3);
        System.out.println("Stunden: " + h);
    }
```

Es sind nur die Änderungen gegenüber der Aufgabe 24 angegeben.

3.5.6 this Referenz

Betrachten wir die Methode `void setTime(int h, int m, int s)` der Klasse **Time** nochmals. Die Parameternamen wurden absichtlich anders gewählt als die Namen der Attribute, damit es keine ungewollten Nebeneffekte gab. Falls die Namen der Parameter im Konstruktor gleich den Attribut-Namen gewählt wären, so würden in Java die Attribute durch die Parameter verdeckt werden und die Anweisung **hour = hour** in der Methode hätte keine Auswirkung. Um dies zu vermeiden, kann mit der **this** Referenz spezifiziert werden, ob das Attribut gemeint ist oder der Parameter. Die Referenz **this** ist das Objekt selbst, in dem sich die Ausführungen des Programmes befindet und zeigt somit auf denselben Speicherbereich, auf den das aktuelle Objekt zeigt. Damit können wir nun die Methode mit geeigneten Parameternamen neu implementieren.

```
    Time(int hour, int min, int sec){
        this.hour = hour;
        this.min = min;
        this.sec = sec;
    }
```

Bei der Ausführung wird mit **this.hour = hour** dem Attribut **hour** der Klasse **Time** den Wert des Parameters **hour** zugewiesen.

Das analoge Problem besteht auch, falls in einer Methode eine lokale Variable gleich heisst, wie ein Attribut auch dann wird das Attribut verdeckt und kann mit **this** darauf zugegriffen werden.

3.5.7 Exkurs I/O

Bis jetzt haben wir die Eingaben direkt im Programm als fixe Werten implementiert. In diesem Kapitel werden wir die Eingabe über die Tastatur betrachten, damit wir beim Testen eines Programmes flexibler sind. Wie man einen Text ausgibt, haben wir bereits in den vorherigen Beispielen und Aufgaben gesehen, nämlich mit **`System.out.println("falscher Monat!");`**

Die Eingabe über die Tastatur ist in Java recht aufwendig, deshalb wurden im File «**Stdin.java**» Methoden erstellt, damit wir die Eingabe als BlackBox betrachten können und damit wir die Methoden direkt nutzen können. Das File «Stdin.java» wird Ihnen zur Verfügung gestellt. Die Klasse **InputStream**, welche für die Eingabe über die Tastatur notwendig ist, werden wir nicht behandeln, da es nicht direkt zur Einführung OOP gehört.
Am einfachsten ist es, falls Sie das File «**Stdin.java**» direkt in ihr Verzeichnis mit allen Java Files kopieren, das hat aber den Nachteil, dass Sie evtl. das File in mehreren Verzeichnissen gespeichert haben.
Eine schönere Variante ist das File «Stdin.java» als Bibliothek im Projekt anzugeben, dazu müssen

Sie mit folgenden Schritten ein **jar**-File erzeugen.

1. Öffnen Sie ein Terminal oder Command Fenster.
2. Navigieren Sie zum Pfad, in dem sich das File «Stdin.java» befindet.
3. Erstellen Sie ein **class** File mit: **`javac Stdin.java`**, damit wird der Compiler aufgerufen.
4. Rufen Sie nun folgenden Befehl auf, um ein **jar**-File zu erzeugen:
   `jar cf Stdin.jar Stdin.class`
5. Fügen Sie das **jar**-File in ihrer Entwicklungsumgebung (IDE) unter «Referenced Libraries» ein.
6. Nun können Sie die Funktionalität von Stdin.java nutzen.

Beispiel 21

Als Test für die Klasse **`Stdin`** können Sie folgenden Code in Java ausführen.

```
import java.io.IOException;

public class TestStdIn {
    public static void main(String[] args) throws IOException{
        System.out.println("Bitte Zahl eingeben:");
        int a = Stdin.readInt();
        System.out.println("Die Zahl lautet: " + a);

        System.out.println("Bitte Gleitkommazahl eingeben:");
        double d = Stdin.readDouble();
        System.out.println("Die Zahl lautet: " + d);
    }
}
```

Neu ist, dass im obigen Programm zu Beginn ein **`import`** der **`IOException`** eingefügt ist und hinter der **`main`**-Methode steht **`throws IOException`**, da das Einlesen über die Tastatur eine solche **Exception** auslösen könnte. Wir gehen hier nicht weiter auf die **Exceptions** ein, sondern benutzten den obigen Code als Vorlage für das Einlesen einer Zahl über die Tastatur.

Aufgabe 27

Erweitern Sie die Aufgabe 4, sodass die Koeffizienten einer quadratischen Gleichung als **`double`** Werte über die Tastatur eingelesen werden. Lösen Sie danach die Gleichung wie in Aufgabe 4.

Lösung:

```
import java.io.IOException;

public class Aufgabe27 {
    public static void main(String[] args) throws IOException{
        System.out.println("Geben Sie den Koeffizienten von
                            x^2 an:");
        double a = Stdin.readDouble(); //1.4;
        System.out.println("Geben Sie den Koeffizienten von
                            x an:");
        double b = Stdin.readDouble(); //2.4;
```

```
        System.out.println("Geben Sie den Koeffizienten c an:");
        double c = Stdin.readDouble(); //-0.4;

        double res;
        //b^2 ist xor!
        res = -b + Math.sqrt(b*b - 4*a*c) / (2*a);
        System.out.println(res);
        res = -b - Math.sqrt(b*b - 4*a*c) / (2*a);
        System.out.println(res);
    }
}
```

3.6 Initialisierung von Objekten

Beim Erzeugen eines Objektes mit dem **new**-Operator wurde vorhin erwähnt, dass Java die Zahlenwerte mit Null initialisiert. Beim Instanziieren eines Objektes wird ein sog. Konstruktor aufgerufen, da wir aber in der Klasse keinen solchen definiert haben, hat Java für uns einen erstellt, einen sog. **Default-Konstruktor**. Es gibt aber zahlreiche Anwendungen, in denen man die Werte selbst setzen möchte, wie z.B. in der Klasse **Line** wäre das Definieren der Koordinaten der zwei Punkte beim Erstellen des Objektes sehr hilfreich. Mit einem Konstruktor müssen nicht im Nachhinein die Koordinaten von Hand gesetzt werden, sondern werden automatisch beim Instanziieren eines Objekten der Klasse **Line** gesetzt.

3.6.1 Konstruktor

Ein Konstruktor ist eine Methode, welche gleich heisst wie die Klasse und wird jedes Mal bei einer **new**-Operation aufgerufen. Der Konstruktor hat keinen Rückgabewert.

Beispiel 22

```
public class Point {
    double x;
    double y;
    Point(){
        x = 1;
        y = 0;
    }
}
```

Hier wurde der Klasse **Point** aus Beispiel 14 einen parameterlosen Konstruktor hinzugefügt. Er hat keinen Returnwert auch **keine** **void** Bezeichnung!

3.6.2 Default-Konstruktor

Der Default-Konstruktor wird von Java erzeugt, falls der Programmierer keinen Konstruktor definiert hat. Sobald irgendein Konstruktor in der Klasse programmiert wird, so wird keinen Default-Konstruktor automatisch erzeugt.

3.6.3 Parameterlose Konstruktor

Ein parameterloser Konstruktor hat, wie es der Name sagt, keine Parameter. Im Beispiel 22 haben wir der Klasse **`Point`** einen solchen hinzugefügt, um die Felder zu initialisieren.
Im Konstruktor können nicht nur die Felder initialisiert werden, sondern z.B. bei einer Steuerung, könnte man im Konstruktor alle Motoren einschalten und sie auf «Halten» stellen.
Nun ändern wir das Beispiel 22 ein wenig ab, um den Aufrufzeitpunkt des Konstruktors zu verfolgen, indem eine Meldung auf der Konsole erscheint.

Beispiel 23

```
public class Point {
    double x;
    double y;
    Point(){
        x = 1;
        y = 0;
        System.out.println("Ein Punkte wurde erstellt.");
    }
    public static void main(String[] args) {
        Point b = new Point();
    }
}
```

Dadurch ist der Zustand des Objektes **`b`** im Speicher mit unseren Werten definiert.

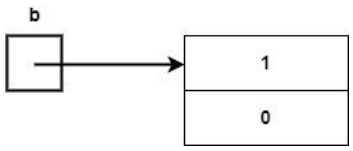

Abbildung 15: Objekt b mit Zustand nach Konstruktor Aufruf

Aufgabe 28

Testen Sie das Beispiel 23.

Aufgabe 29

Erstellen Sie für die Klasse **`Date`** aus Aufgabe 18 einen Konstruktor, welcher ein Objekt mit dem heutigen Datum erzeugt und das Datum auf der Konsole ausgibt. Erstellen Sie eine **`main-`** Methode und testen Sie den Konstruktor.
Erstellen Sie zudem eine Skizze des Speicherinhaltes.

Lösung:

```
public class Date {
    int day;
    String month;
    int year;
    Date(){
        day = 29;
        month = "Dezember";
        year = 2023;
```

```
        System.out.println("Datum: " + day + ". "+ month +
                           ". " + year);
    }

    public static void main(String[] args){
        Date today = new Date();
    }
}
```

Skizze:

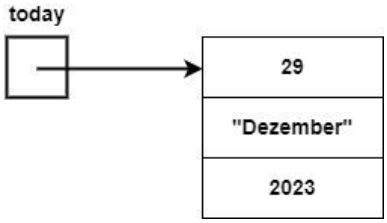

Abbildung 16: Objekt today mit Zustand nach Konstruktor Aufruf

3.6.4 Konstruktor mit Parameter

Mit dem parameterlosen Konstruktor konnten wir nur alle Attribute des Objektes mit einem konstanten Wert initialisieren. In den meisten Fällen möchte man aber jedes Objekt mit unterschiedlichen Daten erzeugen. Deshalb existieren Konstruktoren mit Parametern. Diese Art von Konstruktoren können die Attribute mit Werten initialisiert werden, welche der Anwender selbst definieren kann. Dazu zeigen wir ein Beispiel und erweitern dazu die Klasse **`Point`**.

Beispiel 24

Die Klasse **`Point`** haben wir bereits weiter vorne gesehen, nun möchten wir dem Anwender der Klasse erlauben die Felder **`x`** und **`y`** mit einem beliebigen Werten zu initialisieren. Dies wird mit einem Parameter im Konstruktor ermöglicht.

```
public class Point {
    double x;
    double y;
   Point(){
        x = 0;
        y = 0;
        System.out.println("Ein Punkt wurde erstellt.");
    }
   Point(double a){
        x = a;
        y = a;
        System.out.println("Ein Punkt mit " + x + " und "
                           + y + " wurde erstellt.");
    }
    public static void main(String[] args) {
       Point b = new Point();
       Point c = new Point(3.2);
    }
}
```

Beispiel 25

Der häufigerer Fall beim Erstellen eines Punktes ist die Attribute **x** und **y** mit verschiedenen Werten zu initialisieren, deshalb benötigen wir zwei Parameter im Konstruktor. Damit kann beim Erzeugen des Objektes der Klasse **Point** einen Punkt abgebildet werden, welche bereits benötigte x- und y-Werte enthält.

```
    Point(double x, double y){
        this.x = x;
        this.y = y;
        System.out.println("Ein Punkt mit " + x + " und "
                           + y + " wurde erstellt.");
    }
```

Aufgabe 30

Ergänzen Sie nun die Klasse **Date** (aus Aufgabe 18) mit einem Konstruktor, um ein beliebiges Datum zu setzen.

Lösung:
Es wird nur den zusätzlichen Konstruktor angegeben:

```
Date(int day, String month, int year){
    this.day = day;
    this.month = month;
    this.year = year;
    System.out.println("Datum: " + day + ". " + month + ". "
                       + year);
}
```

Nun können wir der Klasse **Line** mit den Attributen für den Start- und Endpunkt verschiedene Konstruktoren hinzufügen, um beim Erstellen des Objektes der Klasse **Line** ein Definieren dieser Punkte zu vereinfachen.

Beispiel 26

```
public class Line {
    Point start;
    Point end;

    Line(){
        start = new Point();
        end = new Point();
        System.out.println("Linie mit Standardwerten.");
    }

    Line(double startx, double starty, double endx, double endy){
        start = new Point(startx, starty);
        end = new Point(endx, endy);
        System.out.println("Linie mit x und y für Start
                           und End.");
    }
```

```
Line(Point start, Point end){
        this.start = start;
        this.end = end;
        System.out.println("Linie aus zwei Punkten.");
    }

    public static void main(String[] args) {
        Line l1 = new Line();
        Line l2 = new Line(1.2, 3.2, 5.1, 6.3);
        Point a = new Point(2, 1);
        Point b = new Point(2.3, 5);
        Line l3 = new Line(a, b);
    }
}
```

Beim ersten Konstruktor werden die Felder für den Start- und Endpunkt mit den Default-Werten erzeugt, beim zweiten Konstruktor werden je die x und y Koordinaten des Start- und Endpunktes angegeben.
Da die Klassen einen neuen Datentyp definieren, könnte man im Konstruktor als Parameter, ähnlich wie bei den Attributen, auch ein Objekt dieses Datentyps übergeben. Dies ermöglicht uns, wie im Konstruktor **`Line(Point start, Point end)`** implementiert, die Start- und Endpunkte im Objekt der Klasse **`Line`** mit zwei Objekten der Klasse **`Point`** zu setzen. Das Thema Objekte als Parameter wird im Kapitel 3.8 «Objekte als Parameter» genauer analysiert.

Aufgabe 31

Erweitern Sie die Klasse **`Time`** aus der Aufgabe 24 mit zwei Konstruktoren. Der erste erstellt ein Objekt mit dem heutigen Datum und der zweite mit einem beliebigen Datum. Verwenden Sie die bereits implementierten Methoden **`void setTime()`** und **`void setTime(int h, int m, int s)`**.

Lösung:

```
Time(){
    setTime();
}
```

```
Time(int hour, int min, int sec){
    setTime(hour, min, sec);
}
```

Aufgabe 32

Erweitern Sie im Konstruktor der Klasse **`Line`** aus Beispiel 26 die Ausgabe der Punkte, indem Sie diese Aufgabe den Objekten der Klasse **`Point`** delegieren.

Lösung:

```
public class Line {
    Point start;
    Point end;

    Line(){
        start = new Point();
        end = new Point();
        System.out.println("Linie mit Standardwerten.");
        start.print();
        end.print();
    }

    Line(double startx, double starty, double endx, double endy){
        start = new Point(startx, starty);
        end = new Point(endx, endy);
        System.out.println("Linie mit x und y für Start
                            und End.");
        start.print();
        end.print();
    }

    Line(Point p1, Point p2){
        start = p1;
        end = p2;
        System.out.println("Linie aus zwei Punkten.");
        start.print();
        end.print();
    }

    public static void main(String[] args) {
        Line l1 = new Line();
        Line l2 = new Line(1.2, 3.2, 5.1, 6.3);
        Point a = new Point(2, 1);
        Point b = new Point(2.3, 5);
        Line l3 = new Line(a, b);
    }
}
```

3.7 Zugriffsmodifikatoren

Wir haben vorhin gesehen, dass eine Klasse einen neuen Datentyp mit den zugehörenden Operationen definiert. Diese Operationen sind durch Methoden beschrieben, bilden die Schnittstelle der Klasse und werden auf Objekte angewendet. Zudem gibt es «interne» Operationen, welche nur innerhalb einer anderen Operation benutzt werden sollten und somit von aussen nicht direkt aufgerufen werden können.
Eine andere Frage ist die Reglung der Benutzung einer Klasse A. Welche anderen Klassen dürfen die Operationen der Klasse A benutzen und welche nicht?
Als nächster Punkt machen wir uns Gedanken über die Gültigkeit von Attributen. Sie beschreiben den Zustand eines Objektes und müssen deshalb beim Verändern eines Wertes überprüft werden, damit kein undefinierter Zustand vorliegen kann, z.B. sollten die Minuten einer Uhr zwischen 0 und 59 sein. Anders gesagt, gewisse Daten müssen geschützt werden.
Dies führt zu einem wichtigen Konzept der OOP, die sog. **Information Hiding**. Damit werden z.B. die Attribute eines Objektes versteckt und geschützt, sodass der Zugriff kontrolliert werden kann. Dies ermöglicht, dass nur «gültige» Zustände im Speicher vorliegen können.
Angenommen wir haben die folgende zwei Klassen **`A`** und **`B`**. Ein Objekt **`Obj_B`** der Klasse **`B`** kann Operationen (Dienste, Methoden) eines Objektes **`Obj_A`** der Klasse **`A`** nutzen (Abbildung 17).

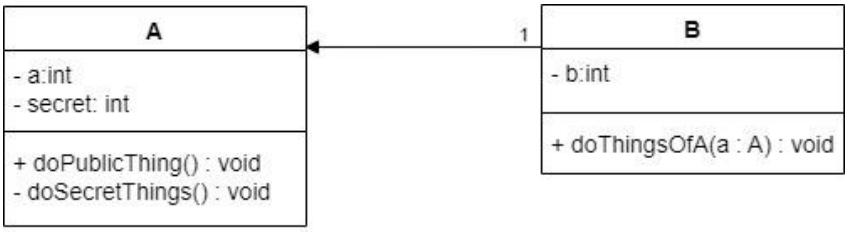

Abbildung 17: Klasse B nutzt die Klasse A

Das Objekt **`Obj_B`** darf aber nicht alle Operationen des Objektes **`Obj_A`** ausführen, da **`Obj_A`** Operationen enthält, um Geheimnisse der Klasse **`A`** zu bearbeiten. Deshalb müssen wir diese Methoden vor dem Objekt **`Obj_B`** verstecken oder schützen. Dafür existieren die Zugriffsmodifikatoren, um anzugeben, welche Methoden oder Attribute ausserhalb der Klasse **`A`** benützt werden können.

3.7.1 Zugriffsmodifikator default

Der Modifikator «**`default`**», welcher gilt, falls nichts angegeben wird, hat mit «**`package`**» zu tun, da wir aber in dieser Arbeit nicht auf «**`package`**» eingehen, ist er für diese Arbeit gleichwertig wie «**`public`**». Der Unterschied ist, dass mit dem «**`default`**» Zugriffsmodifikator alle Klassen und Objekten im selben Paket auf die Attribute oder Methoden zugreifen können. Falls man keinen Modifikator bei einer Klasse, einem Attribut oder einer Methode angibt, so ist automatisch «**`default`**» aktiv, dies entspricht dem, was wir bis jetzt verwendet haben.

3.7.2 Zugriffsmodifikator public

Besitzt eine Klasse, ein Attribut oder eine Methode den Modifikator «**public**», so können Objekte einer anderen Klasse darauf zugreifen.
Betrachten wir dazu die folgende Situation.

Eine Klasse **A** enthält ein Attribut **Obj_B** der Klasse **B**. Im Konstruktor der Klasse **A** kann ein Objekt **Obj_B** nur dann erzeugt werden, falls die Klasse **B** und der entsprechende Konstruktor der Klasse **B** den Zugriffmodifikator «**public**» hat oder «**default**» in unserem Fall.

Beispiel 27

Betrachten wir dazu die zwei Konstruktor der Klasse **Line** aus der Aufgabe 32:

```
    Line(){
        start = new Point();
        end = new Point();
        System.out.println("Linie mit Standardwerten.");
    }

    Line(double startx, double starty, double endx, double endy){
        start = new Point(startx, starty);
        end = new Point(endx, endy);
        System.out.println("Linie mit x und y für Start
                            und End.");
    }
```

In der Klasse **Line** werden zwei Objekte (**start**, **end**) der Klasse **Point** erzeugt. Wäre die Klasse **Point** und deren Konstruktor nicht **public** (oder **default**), so würde der **new**-Operator folgenden Fehler anzeigen:
Illegal modifier for the class Point; only public, abstract & final are permitted
Und die Objekte **start** und **end** könnten nicht instanziiert werden.

3.7.3 Zugriffsmodifikator private

Mit «**private**» kann der Zugriff auf die Attribute und Methoden geschützt werden, indem nur die Methoden der Klassen selbst darauf zugreifen können. «**private**» bei einer Klasse macht nur Sinn, falls es sich um eine innere Klasse handelt, damit ist gemeint, dass eine Klasse in einer bestehenden Klasse definiert wird. Innere Klassen werden wir nicht behandeln.

Beispiel 28

Anhand der Klasse **`TimeHiding`** möchten wir zeigen, welche Möglichkeit uns «Information Hiding» bietet.
Gehen wir zunächst von folgender Implementierung aus, was fast gleich aussieht wie die Klasse **`Time`**. Wir haben nun im Hauptprogramm eine Zeile hinzugefügt: **`t.min = 1200`**.

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TimeHiding {
    int std;
    int min;
    int sec;

    public TimeHiding(){
        GregorianCalendar gc = new GregorianCalendar();
        std = gc.get(Calendar.HOUR_OF_DAY);
        min = gc.get(Calendar.MINUTE);
        sec = gc.get(Calendar.SECOND);
    }

    public TimeHiding(int std, int min, int sec){
        this.std = std;
        this.min = min;
        this.sec = sec;
    }

    void print(){
        System.out.println(std + ":" + min + ":" + sec);
    }

    public static void main(String[] args) {
        TimeHiding t = new TimeHiding();
        t.min = 1200;
        t.print();
    }
}
```

Lassen wir nun das Programm laufen, so erhalten wir eine eigenartige Zeit: **10:1200:37**. Da wir die Minuten auf 1200 gesetzt haben. Wir wissen aber, dass die Minuten bei einer Zeitangabe zwischen 0 und 59 liegen müssen. Das Programm läuft trotzdem durch und ergibt auch keinen Fehler, obwohl der Wert nicht stimmt. Nun möchten wir diesen ungeschützten Zugriff auf die Datenfelder des Objektes **`t`** der Klasse **`TimeHiding`** schützen. Dazu schreiben wir **`private`** vor der Deklaration der Attribute.

```
    private int std;
    private int min;
    private int sec;
```

Aufgabe 33

Testen Sie mit dieser Änderung das Programm aus Beispiel 28.

Wie Sie bemerkt haben, funktioniert das Programm immer noch. Der Grund ist, die **main-**Methode ist auch in der Klasse **TimeHiding** definiert und somit ist **main(…)** ein Element der Klasse und kann auf die privaten Felder zugreifen.

Beispiel 29

Wir erstellen in diesem Beispiel eine Klasse **TestTimeHiding**, welche nur die **main(…)**-Methode enthält und löschen die **main**-Methode aus der Klasse **TimeHiding**.

```
public class TestTimeHiding {
    public static void main(String[] args) {
        TimeHiding t = new TimeHiding();
        t.min = 1200;
        t.print();
    }
}
```

Wir zeigen hier nur die Klasse **TestTimeHiding**.

Aufgabe 34

Versuchen Sie nun nochmals das Testprogramm aus dem Beispiel 29 zu starten.

Lösung:
Sie haben einen Compilerfehler erhalten, das Programm wurde nicht gestartet.
The field TimeHiding.min is not visible

Sie sehen nun, dass auf die Felder der Klasse **TimeHiding** nicht direkt zugegriffen werden kann und sie somit geschützt sind.

Aufgabe 35

Sie möchten aber trotzdem die Zeit einstellen. Überlegen Sie sich wie Sie die Klasse **TimeHiding** erweitern können, damit das geschützte Feld **min** nach dem Erzeugen des Objektes trotzdem verändert werden kann. Zusätzlich soll auch sichergestellt werden, dass die Minuten zwischen 0 und 59 sein sollen. Testen Sie ihre Lösung in der Klasse **TestTimeHiding** aus, die Uhrzeit sollte immer eine gültige Zeit sein.

Lösung:
Man schreibt eine Methode **public void setMin(int min)** in der Klasse **TimeHiding**, welche den Wert nur verändert, falls der Bereich stimmt.

```
    public void setMin(int min){
        if(min >= 0 && min < 60){
            this.min = min;
        }else{
            this.min = 59;
        }
    }
```

```
public class TestTimeHiding {
    public static void main(String[] args) {
        TimeHiding t = new TimeHiding();
        t.setMin(1200);
        t.print();
    }
}
```

Die Minuten haben nun trotz der Anweisung **`t.setMin(1200)`** einen sinnvollen Wert.

Aufgabe 36

Starten Sie folgendes Programm.

```
public class TestTimeHiding {
    private static void main(String[] args) {
        TimeHiding t = new TimeHiding();
        t.setMin(1200);
        t.print();
    }
}
```

Sie haben sicherlich bemerkt, dass Sie das Programm gar nicht starten können, da die Methode **`main(…)`** **`private`** ist und nur die Klasse diese Methode ausführen kann. Nicht einmal das System darf darauf zugreifen. Deshalb macht es keinen Sinn eine **`main`**-Methode **`private`** zu deklarieren.

Beispiel 30

Nun ändern wir in diesem Beispiel den Zugriffmodifikator bei den Konstruktoren der Klasse **`TimeHiding`** auf **`private`**, setzen die **`main(…)`** Methode wieder auf **`public`** und versuchen das Programm zu starten.

Die Konstruktoren sehen nun folgend aus:

```
    private TimeHiding(){
        GregorianCalendar gc = new GregorianCalendar();
        std = gc.get(Calendar.HOUR_OF_DAY);
        min = gc.get(Calendar.MINUTE);
        sec = gc.get(Calendar.SECOND);
    }
```

```
    private TimeHiding(int std, int min, int sec){
        this.std = std;
        this.min = min;
        this.sec = sec;
    }
```

Wir erhalten einen Fehler:
The constructor TimeHiding() is not visible

Ist ein Konstruktor **private**, so kann kein Objekt der Klasse **TimeHiding** ausserhalb dieser Klasse erzeugt werden.

3.7.4 Zugriffsmodifikator protected

Dieser Modifikator hat erst mit der Vererbung eine Bedeutung, trotzdem wird er hier erklärt, damit alle Zugriffmodifikatoren im selben Unterkapitel sind. Es wird auf das entsprechende Kapitel 5 «Vererbung» für das Verständnis der nächsten Zeilen verweisen.
Der Modifikator «**protected**» bewirkt dasselbe wie **private**, ausser bei einer abgeleiteten Klasse, welche auch auf die **protected** Datenfelder zugreifen darf. Betrachten Sie später in diesem Dokument das Beispiel 40 im Kapitel 5 «Vererbung» für eine detaillierte Erklärung.

3.7.5 Information Hiding

Mit den Zugriffsmodifikatoren, v.a. mit **private** können wir die Daten schützen, wir können auch Methoden schützen, sodass sie nur innerhalb eines Objektes aufgerufen werden können und von aussen nicht sichtbar sind. Dies ist ein wichtiges Konzept, denn so kann das Zuweisen eines Wertes an ein Attribut kontrolliert werden, sodass die Felder in einem Objekt immer sinnvolle Werte besitzen, wie wir es mit der Uhrzeit gemacht haben.
Wir greifen auf die Daten mit **get**... **(**...**)** zu und verändern sie mit **set**... **(**...**)**, dabei können wir als Programmierer entscheiden, was mit nicht sinnvollen Eingaben, resp. Parameterwerte gemacht wird. Die Methoden **get**... **(**...**)** und **set**... **(**...**)** sind üblich und man benennt sie in der Fachsprache auch **setter** und **getter**.

Beispiel 31

Es folgt das vollständige Beispiel der Klasse **TimeHiding** mit allen **setter** und **getter**.

```
import java.util.Calendar;
import java.util.GregorianCalendar;

public class TimeHiding {
    private int std;
    private int min;
    private int sec;

    public TimeHiding(){
        GregorianCalendar gc = new GregorianCalendar();
        std = gc.get(Calendar.HOUR_OF_DAY);
        min = gc.get(Calendar.MINUTE);
        sec = gc.get(Calendar.SECOND);
    }
```

```
    public TimeHiding(int std, int min, int sec){
        this.std = std;
        this.min = min;
        this.sec = sec;
    }

    public void setStd(int std){
        if(std >= 0 && std < 24){
            this.std = std;
        }else{
            this.std = 0;
        }
    }

    public void setMin(int min){
        if(min >= 0 && min < 60){
            this.min = min;
        }else{
            this.min = 0;
        }
    }

    public void setSec(int sec){
        if(sec >= 0 && sec < 60){
            this.sec = sec;
        }else{
            this.sec = 0;
        }
    }

    public int getStd(){
        return std;
    }

    public int getMin(){
        return min;
    }

    public int getSec(){
        return sec;
    }

    void print(){
        System.out.println(std + ":" + min + ":" + sec);
    }
}
```

```
public class TestTimeHiding {
    public static void main(String[] args) {
        TimeHiding t = new TimeHiding();
        t.setMin(1200);
        t.print();

        //oder
        if (t.getStd() < 10){
            System.out.print("0" + t.getStd() + ":");
        }else{
            System.out.print(t.getStd() + ":");
        }
        if (t.getMin() < 10){
            System.out.print("0" + t.getMin() + ":");
        }else{
            System.out.print(t.getMin() + ":");
        }

        if (t.getSec() < 10){
            System.out.print("0" + t.getSec());
        }else{
            System.out.print(t.getSec());
        }
    }
}
```

3.8 Objekte als Parameter

Übergeben wir einer Methode einen Basisdatentyp (**int**, **double**, …), so wird die übergebene Variable kopiert und in der Methode wird mit der Kopie gearbeitet, dies nennt man «**call by value**» (Abbildung 18).

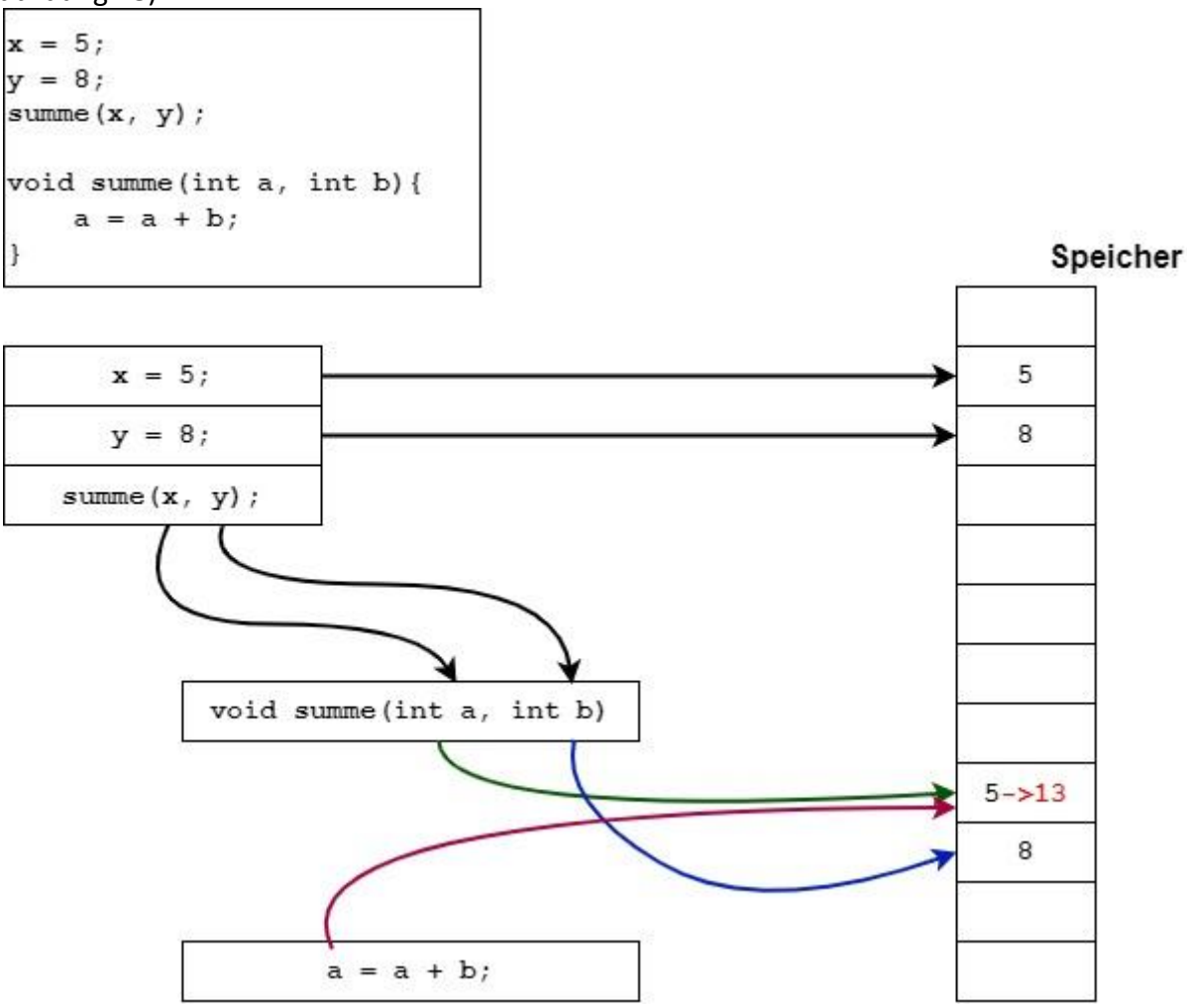

Abbildung 18: Speicherbelegung beim Aufruf mit call by value

In der Abbildung 18 ist die Speicherbelegung beim Aufruf der Methode **void summe(int a, int b)** dargestellt. Daraus ist ersichtlich, dass der Variablenwert von **x** dem Parameter **a** und der Variablenwert von **y** dem Parameter **b** zugewiesen wird und dass einen eigenen Speicher für die Parameterwerte belegt wird. Demzufolge arbeitet die Methode mit den Kopien der Variablen **x** und **y** und die Summe, welche im Parameter **a** gespeichert wird, ändert nicht den Variablenwert von **x**.
Bei einem Objekt als Parameter, wird das Objekt kopiert, d.h. die Kopie zeigt auf denselben Speicherbereich, so wie es im Kapitel 3.3 «Objekte» beschrieben ist, jede Änderung am Objekt, am formalen oder am aktuellen Parameter, ist somit direkt eine Änderung des Zustandes im Speicher, deshalb wird dies «**call by reference**» (Abbildung 19) genannt.

```
void mult(Fractional a, Fractional b){
 a.num = a.num * b.num;
 a.denom = a.denom * b.denom;
}
```

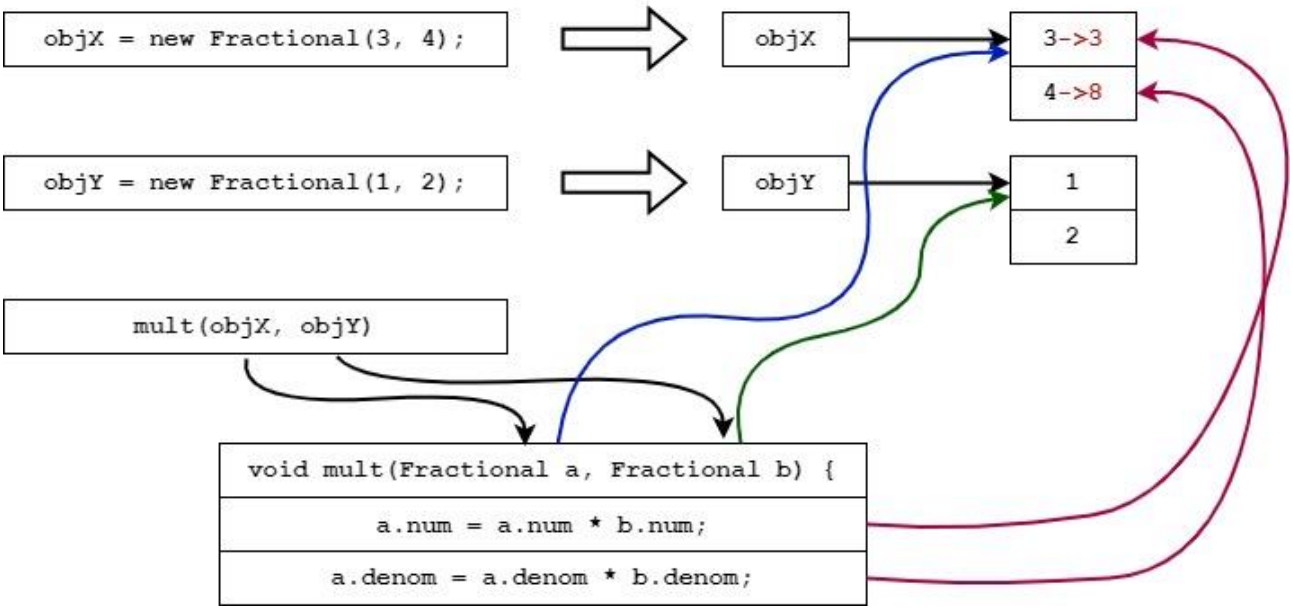

Abbildung 19: Speicherbelegung beim Aufruf mit call by reference

Wir verwenden hier eine Klasse **`Fractional`** mit den Attributen **`num`** und **`denom`**, der vollständige Code diese Klasse wird im Beispiel 34 gezeigt.

```
public class Fractional {
    int num;
    int denom;
}
```

In der Abbildung 19 ist ersichtlich, dass ein Objekt einen Speicherort referenziert, d.h. bei der Übergabe eines Objektes der Methode **`void mult(Fractional a, Fractional b)`** , erhalten die Parameter **`a`** und **`b`** eine Kopie der Referenz also eine Kopie der Objekte **`objX`** und **`objY`**, d.h. die Parameter **`a`** und **`b`** referenziert denselben Speicherort wie die Objekte **`objX`** und **`objY`**. Wird nun das Feld **`num`** oder **`denom`** des Parameters **`a`** verändert, so ist die Änderung im selben Speicherbereich, welcher das Objekt **`Obj_X`** referenziert. Also ist die Änderung auch im Objekt **`Obj_X`** sichtbar!

Dazu wird ein kleines Beispiel gezeigt, um diesen Effekt zu zeigen.

Beispiel 32

```
class Parameter{
    private int a;
    private double b;

    public Parameter(int a, double b){
        this.a = a;
        this.b = b;
    }

    public void setA(int a) {
        this.a = a;
    }

    public void setB(double b) {
        this.b = b;
    }
    public void print(){
        System.out.println("class Parameter: a: " + a + ", b: "
                           + b);
    }
}

public class ParameterTest {
    public ParameterTest(Parameter param){
        param.setA(0);
        param.setB(0);
    }

    public void changeParam(Parameter param, int a, double b){
        param.setA(a);
        param.setB(b);
        a = 0;
        b = 0;
    }
    public static void main(String[] args){
        int a = 5;
        double b = 3.14;
        Parameter p = new Parameter(a, b);
        p.print();
        ParameterTest pT = new ParameterTest(p);
        p.print();
        p.setA(8);
        p.setB(2.71);
        p.print();
        pT.changeParam(p, a, b);
        System.out.println("a global: " + a + ", b global: " + b);
        p.print();
    }
}
```

Um das Beispiel besser zu verstehen, gehen wir einige wichtige Punkte durch. Beginnen wir mit den Zeilen:

```
int a = 5;
double b = 3.14;
Parameter p = new Parameter(a, b);
```

Hier wird ein Objekt **p** der Klasse **Parameter** instanziiert (Abbildung 20).

Abbildung 20: Objekt p mit den Initialwerten

Die folgende Zeile erzeugt ein Objekt **pT** der Klasse **ParameterTest**, wobei das Objekt **p** dem Konstruktor übergeben wird, womit dann der formale Parameter **param** des Konstruktors denselben Speicherbereich wie **p** referenziert (Abbildung 21).

```
ParameterTest pT = new ParameterTest(p);
```

Im Konstruktor wird mit

```
param.setA(0);
param.setB(0);
```

den Zustand des Objektes **p** wird über den Parameter **param** auf null gesetzt (Abbildung 21).

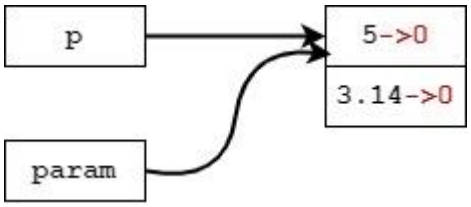

Abbildung 21: Zustand des Objekte p mit dem Objekt param verändern

Die nächsten Zeilen

```
p.setA(8);
p.setB(2.71);
```

ändern wieder den Zustand des Objektes **p**, nun stehen im Speicher die Werte **8** und **2.71** (Abbildung 22).

Danach folgen die Zeilen in der Methoden **pT.changeParam(p, a, b)**,

```
param.setA(a);
param.setB(b);
```

wobei **a** und **b** bereits definiert sind (**a = 5** und **b = 3.14**), welche den Zustand des Objektes **p** über den Parameter **param** wieder verändern (Abbildung 22).

Die Anweisungen

```
a = 0;
b = 0;
```

im Aufruf **pT.changeParam(p, a, b)** haben keinen Einfluss auf die Variablen **a** und **b**, da in der Methode mit einer Kopie gearbeitet wird (Abbildung 22).

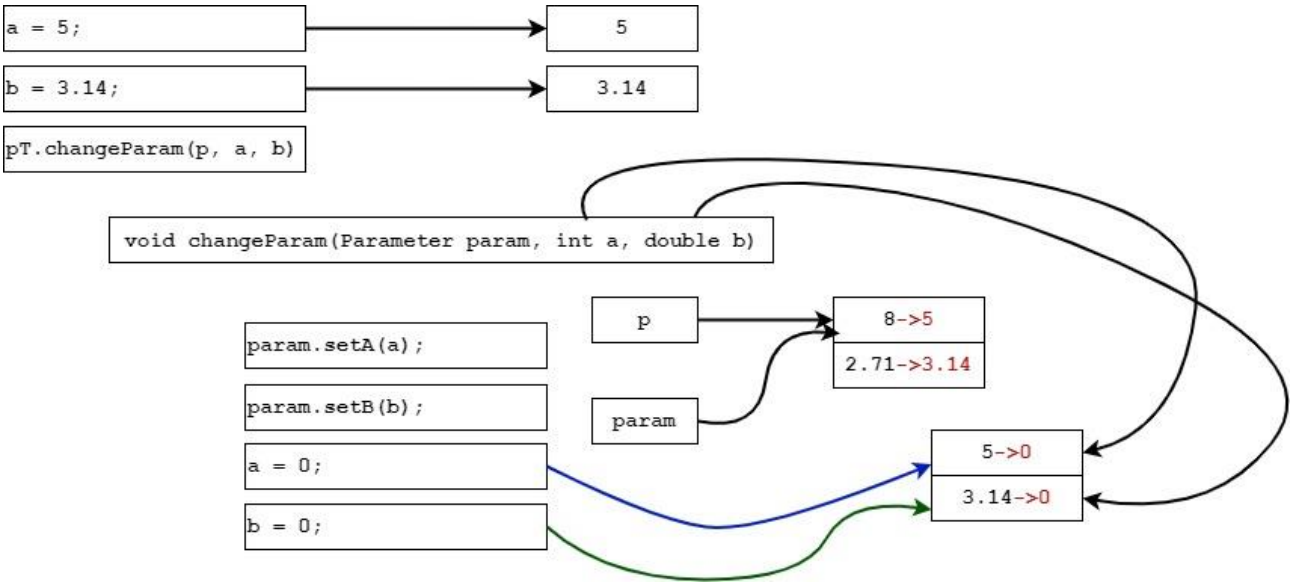

Abbildung 22: Unterschied zwischen call by reference und call by value

Aufgabe 37

Lassen Sie das obige Programm laufen und erklären Sie ihrem Nachbarn die Ausgaben in der Konsole.

Aufgabe 38

Erweitern Sie die Klasse **Vector** aus dem Beispiel 15 um eine Methode **add(Vector v)** welche zwei Vektoren addiert. Das Resultat steht im aufrufenden Objekt.

Lösung:

```
public void add(Vector v){
    x = x + v.x;
    y = y + v.y;
    z = z + v.z;
}
```

3.8.1 Objekt im Konstruktor kopieren

Wir haben bereits gesehen, dass eine Zuweisung von einem Objekt zu einem anderen also **`p1 = p2`** keinen neuen Speicherbereich für den Zustand des zweiten Objektes erzeugt. Möchte man eine Kopie des Zustandes erhalten, so muss ein neues Objekt erzeugt werden und die Felder einzeln kopiert werden.
Diese Erkenntnis benötigen wir, falls wir einen sog. **Copy-Konstruktor** schreiben möchten, dies bedeutet, dass ein Objekt, dessen Kopie erstellt werden soll, dem Konstruktor übergeben wird. Dadurch entsteht ein zweites Objekt mit demselben Zustand des ersten, aber mit einem eigenen Speicherbereich.

Beispiel 33

Die Klasse **`Time`** wurde um einen Copy-Konstruktor erweitert und im Hauptprogramm wird eine Kopie **`tCopy`** des Objekte **`t`** erzeugt.

```
//Copy-Konstruktor
public Time(Time z){
    this.std = z.std;
    this.min = z.min;
    this.sec = z.sec;
}
```

```
public static void main(String[] args) {
    Time t = new Time(); //enthält die aktuelle Systemzeit
    //Kopie erstellen:
    Time tCopy = new Time(t);
    t.print();
    tCopy.print();
}
```

Aufgabe 39

Ergänzen Sie die Klasse **`Vector`** mit einem Copy-Konstruktor.

Lösung:

```
public Vector(Vector other){
    this.x = other.x;
    this.y = other.y;
    this.z = other.z;
}
```

3.8.2 Objekt als Returnwert

Beispiel 34

Nun erstellen wir eine Klasse **`Fractional`** mit Zähler (numerator) und Nenner (denominator) und möchten eine Methode **`public void mult(Fractional other)`** für die Multiplikation zweier Brüche implementieren. Das Resultat steht wieder im aufrufenden Objekt.

```
public class Fractional {
    private int num;
    private int denom;

    public Fractional(){
        num = 1;
        denom = 1;
    }

    public Fractional(int z, int n){
        this.num = z;
        this.denom = n;
    }

    public void mult(Fractional other){
        num = num * other.num;
        denom = denom * other.denom;
    }

    public void print(){
        System.out.println(num + " / " + denom);
    }
    public static void main(String[] args){
        Fractional b1 = new Fractional(1, 2);
        Fractional b2 = new Fractional(1, 3);

        b1.mult(b2);
        b1.print();
    }
}
```

Im Hauptprogramm wird gezeigt, wie die Multiplikation angewendet wird.
Wir sind uns aus der Mathematik gewohnt, dass wir auch mehrere Brüche auf einer Zeile multiplizieren können, z.B. $\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{2}{5}$ dies geht mit dem obigen Beispiel nicht direkt, sondern wir müssen die Multiplikation der ersten zwei Brüche rechnen, dann das Resultat mit dem dritten Bruch multiplizieren.

```
b1.mult(b2); //das Resultat steht in b1
b1.mult(b3);
```

Es wäre doch schöner, falls wir **`b1.mult(b2).mult(b3)`** schreiben könnten.
Dies kann erreicht werden, falls die Methode **`mult(…)`** ein Objekt mit dem Resultat zurückgibt, entweder das Objekt **`b1`** selbst oder es wird ein neues Objekt erzeugt. Wir betrachten nun beide Varianten.

Zuerst geben wir das Objekt selbst zurück, dies kann einfach mit **this** erledigt werden.

```
public Fractional mult(Fractional other){
    num = num * other.num;
    denom = denom * other.denom;
    return this;
}
```

In der zweiten Variante erzeugen wir ein neues Objekt, dies hat den Vorteil, dass das Objekt, welchen die Methode **mult(…)** aufruft, den ursprünglichen Zustand behalten kann.

```
public Fractional mult(Fractional other){
    Fractional res = new Fractional();
    res.num = num * other.num;
    res.denom = denom * other.denom;
    return res;
}
```

Für beide Varianten funktioniert der folgende Code.

```
public static void main(String[] args){
    Fractional b1 = new Fractional(1, 2);
    Fractional b2 = new Fractional(1, 3);
    Fractional b3 = new Fractional(2, 5);
    Fractional b4 = new Fractional(7, 8);

    Fractional result = b1.mult(b2).mult(b3).mult(b4);
    result.print();
}
```

Damit können wir wie in der Mathematik in einer Zeile die Multiplikation. $\frac{1}{2} \cdot \frac{1}{3} \cdot \frac{2}{5} \cdot \frac{7}{8}$ rechnen.

Aufgabe 40

Erweitern Sie die Klasse **Vector** um eine Methode
public Vector crossProduct(Vector v), welche das Kreuzprodukt berechnet und das Resultat als neues Objekt retourniert, also wie in der zweiten Variante aus dem obigen Beispiel.

Lösung:
Es wird nur den zusätzlichen Code aufgelistet.

```
public Vector crossProduct(Vector v){
    Vector res = new Vector();
    res.x = y * v.z - z * v.y;
    res.y = z * v.x - x * v.z;
    res.z = x * v.y - y * v.x;
    return res;
}
```

```
public void print(){
    System.out.println("[ " + x + ", " + y + ", " + z + " ]");
}

public static void main(String[] args) {
    Vector v = new Vector(1, 1, 0);
    double len = v.length();
    System.out.println("Die Länge ist: " + len);

    Vector vx = new Vector(1, 0, 0);
    Vector vy = new Vector(0, 1, 0);
    Vector vz = vx.crossProduct(vy);
    vz.print();
}
```

Aufgabe 41

Erweitern Sie die Klasse **Fractional** um folgende Methoden:
private int ggT(): Berechnet den grössten gemeinsamen Teiler des Nenners und des Zählers.
private void kuerzen(): Kürzt den Bruch mit den ggt.
public Fractional add(Fractional other): Addiert zwei Brüche.

Lösung:
Es wird nur den zusätzlichen Code aufgelistet.

```
private int ggT(){
    int rest = -1;
    int a = num;
    int b = denom;
    while(rest != 0){
        rest = a % b;
        a = b;
        b = rest;
    }
    return a;
}

private void kuerzen(){
    int a = ggT();
    num = num / a;
    denom = denom / a;
}

public Fractional add(Fractional other){
    Fractional res = new Fractional();
    res.num = num * other.denom + other.num * denom;
    res.denom = denom * other.denom;
    res.kuerzen();
    return res;
}
```

```
public static void main(String[] args){
    Fractional b1 = new Fractional(1, 4);
    Fractional b2 = new Fractional(1, 4);
    Fractional res = b1.add(b2);
    res.print();
}
```

3.8.3 Klassenvariablen und Klassenmethode

Klassenvariablen und Klassenmethoden werden mit dem Schlüsselwort **static** definiert.
Der erste Kontakt mit **static** war die **main**-Methode, welche wir schon öfters benutzt haben.
Der Zusatz **static** bedeutet, dass die Methode oder das Attribut nicht zum Objekt gehört, sondern zur Klasse, deshalb auch der Name Klassenvariable resp. Klassenmethode. Dies bedeutet wiederum, dass eine **static** Methode ohne Objekt aufgerufen werden kann, sofern sie **public** deklariert ist und dass ein **static** Attribut für alle Objekte nur einmal im Speicher existiert.
Dieses Konzept ist bei der **main**-Methode besonders wichtig, da das Java System kein Objekt beim Start hat und trotzdem auf den Startpunkt des Programmes zugreifen muss. Dieser Startpunkt nennt man auch Einstiegspunkt.
Wir werden die Anwendung von **static** anhand von Beispielen einführen.

Beispiel 35

Für die Schülerverwaltung in einer Schule deklarieren wir die Klasse **Pupil**. Jeder Schüler hat eine eindeutige Nummer für die Identifikation, welche beim Einschreiben vergeben wird. Die maximale Anzahl Schüler aller sechs Klassen liegt bei 1000, deshalb wird die obere Grenze der Identifikation auf 10000 gesetzt, damit es sicher genügende Nummern hat. Alle Identifikationen sollten aus vier Stellen bestehen, deshalb beginnt das Zählen bei 1000.
Mit dieser Beschreibung erhalten wir folgende Klasse vorerst noch nicht vollständig.

```
public class Pupil {
    private int year;
    private static int counter = 1000;
    private int id;
    private String name;
    private String firstname;

    public Pupil(String name, String firstname, int year){
        this.firstname = firstname;
        this.name = name;
        this.year = year;
        id = counter;
        counter++;
    }

    public void print(){
        System.out.println(name + " " + firstname);
        System.out.println("Jahrgang: " + year);
        System.out.println("Id Nummer: " + id);
    }
}
```

Bei jeder neuen Einschreibung wird ein neuer Schüler ins System eingetragen, dies bedeutet, dass ein neues Objekt der Klasse **Pupil** erzeugt wird. Nun wird die Vergabe der Identifikation mit einem Zähler **counter** gelöst, welcher hochgezählt wird. Dies funktioniert nur, falls der Zähler für alle Objekte derselbe ist, also definieren wir das Attribut **counter** als **static**, sodass er zur Klasse gehört. Nun können wir beim Erzeugen, also im Konstruktor, den Wert einfach hochzählen. Die Id-Nummer **id** selbst ist aber pro Objekt verschieden, deshalb darf sie nicht **static** sein, dasselbe gilt für die restlichen Attribute.

Die Anwendung dieser ersten Erweiterung wird im Testprogramm **PupilTest** gezeigt.

```
public class PupilTest {
    public static void main(String[] args){
        Pupil pupil1 = new Pupil("Widmer", "Yannik", 2017);
        Pupil pupil2 = new Pupil("Huber", "Marcel", 2016);
        Pupil pupil3 = new Pupil("Hauser", "Levin", 2017);
        Pupil pupil4 = new Pupil("Staub", "Lena", 2016);

        pupil1.print();
        pupil2.print();
        pupil3.print();
        pupil4.print();
    }
}
```

Die Ausgabe zeigt, dass die Id-Nummer eindeutig ist und dass sie jeweils hochgezählt wird.

```
Widmer Yannik
Jahrgang: 2017
Id Nummer: 1000
Huber Marcel
Jahrgang: 2016
Id Nummer: 1001
Hauser Levin
Jahrgang: 2017
Id Nummer: 1002
Staub Lena
Jahrgang: 2016
Id Nummer: 1003
```

Weiter möchten wir die gesamte Anzahl Schüler ermitteln, dazu verwenden wir zusätzlich ein Attribut **public static int pupilCnt** und wir erweitern die Klasse um zwei Methoden **public void register()** und **public void deregister()**.
Das Attribut **pupilCnt** muss für alle Objekte dieselbe Variable sein, deshalb wird sie wiederum mit **static** als Klassenvariable deklariert. Wir benötigen für das Auslesen des Wertes noch eine weitere Methode, da **pupilCnt** **private** ist. Dafür definieren wir eine Methode **public static int getCounter()**, diese Methode wird **static** deklariert, da sie auch ohne Objekt der Klasse **Pupil** in der Verwaltung aufgerufen werden können soll.

Beim Einschreiben (register) wird nun die Id-Nummer vergeben und die Anzahl wird inkrementiert und beim Abmelden (deregister) dekrementieren wir die Anzahl.

Die geänderte Klasse **`Pupil`** sieht nun folgendermassen aus.

```
public class Pupil {
    private int year;
    private static int counter = 1000;
    private int id;
    private String name;
    private String firstname;
    private static int pupilCnt = 0;

    public Pupil(String name, String firstname, int year){
        this.firstname = firstname;
        this.name = name;
        this.year = year;
        //id = counter;
        //counter++;
    }

    public void print(){
        System.out.println(name + " " + firstname);
        System.out.println("Jahrgang: " + year);
        System.out.println("Id Nummer: " + id);
    }

    public void register(){
        id = counter;
        counter++;
        pupilCnt++;
    }

    public void deregister(){
        pupilCnt--;
    }

    public static int getCounter(){
        return pupilCnt;
    }
}
```

Und das Testprogramm:

```
public class pupilTest {
    public static void main(String[] args){
        Pupil pupil1 = new Pupil("Widmer", "Yannik", 2017, 1);
        Pupil pupil2 = new Pupil("Huber", "Marcel", 2016, 2);
        Pupil pupil3 = new Pupil("Hauser", "Levin", 2017, 1);
        Pupil pupil4 = new Pupil("Staub", "Lena", 2016, 2);

        pupil1.register();
        pupil2.register();
        pupil3.register();
        pupil4.register();

        int anz = Pupil.getCounter();
        System.out.println("Anzahl Schüler: " + anz);

        pupil3.deregister();
        pupil3.print();
        anz = Pupil.getCounter();
        System.out.println("Anzahl Schüler: " + anz);
    }
}
```

Beachten Sie, dass der Aufruf der Methode **`getCounter()`** über den **Klassennamen** erfolgt und nicht über das Objekt, was auch möglich wäre. Der Code ist aber einfacher lesbar, falls man alle **`static`** Methoden und Attribute über den Klassennamen aufruft, da sie zur Klasse gehören.

```
int anz = Pupil.getCounter();
```

Aufgabe 42

Erstellen Sie eine Klasse für einen Kinobesucher (Moviegoer). An der Kasse wird beim Verkauf des Tickets eine Nummer von 1 bis 200 dem Besucher zugeteilt und damit verbunden ist direkt eine Platzreservierung (dies zur Vereinfachung der Aufgabe). Im Kino hat es max. 200 Plätze, sobald es 200 Besucher hat, können keine Tickets mehr gekauft werden.
Das Hauptprogramm für den Test ist vorgegeben, nun sollen Sie die Klasse **`Moviegoer`** schreiben, sodass das Hauptprogramm fehlerfrei läuft.

```
public class MovigoersTest {
    public static void main(String[] args){
        Moviegoer person1 = new Moviegoer("Meier", "Peter");
        Moviegoer person2 = new Moviegoer("Rechsteiner",
                                          "Thomas");
        Moviegoer person3 = new Moviegoer("Egger", "Ivan");

        person1.buyTicket();
        person2.buyTicket();
        person3.buyTicket();
```

```
        person2.print();

        int anz = Moviegoer.leftSeats();
        System.out.println("Freie Plätze: " + anz);
    }
}
```

Ergibt die untenstehende Ausgabe.

```
Sitznummer von Rechsteiner, Thomas ist 2
Freie Plätze: 197
```

Lösung:

```
public class Moviegoer {
    private int seatNumber;
    private String name;
    private String firstName;
    private static int currNumber = 0;

    public Moviegoer(String name, String firstName){
        this.name = name;
        this.firstName = firstName;
        this.seatNumber = 0;
    }
    public void buyTicket(){
        if (currNumber >= 200){
            System.out.println("Kino ist ausverkauft.");
        }else{
            currNumber++;
            seatNumber = currNumber;
        }
    }

    public void print(){
        System.out.println("Sitznummer von " + name + ", "
                           + firstName + " ist " + seatNumber);
    }
    public static int leftSeats(){
        return 200 - currNumber;
    }
}
```

3.9 Graphische Darstellung einer Klasse mit UML

UML bedeutet Unified Modelling Language, damit können OOP-Projekt modelliert werden. Uns interessiert vorläufig nur die Darstellung einer Klasse mit ihren Beziehungen zu anderen Klassen. UML ist unabhängig von einer Programmiersprache, man kann sie für jede OO-Programmiersprache benutzen. Wir werden einen kleinen Einblick in UML aufzeigen, sodass wir unsere Klassen graphisch darstellen können, konkret bedeutet dies, dass wir in dieser Arbeit nur das Klassendiagramm benutzen werden.
In diesem Kapitel wird gezeigt, wie eine Klasse mit ihren Attributen und Methoden dargestellt wird. Dies ist nützlich, um alle Operationen einer Klasse zu sehen, ähnlich wie das Inhaltsverzeichnis eines Dokumentes.

Beispiel 36

Als erstes werden wir das Klassendiagramm der Klasse **`Pupil`** darstellen (Abbildung 23).

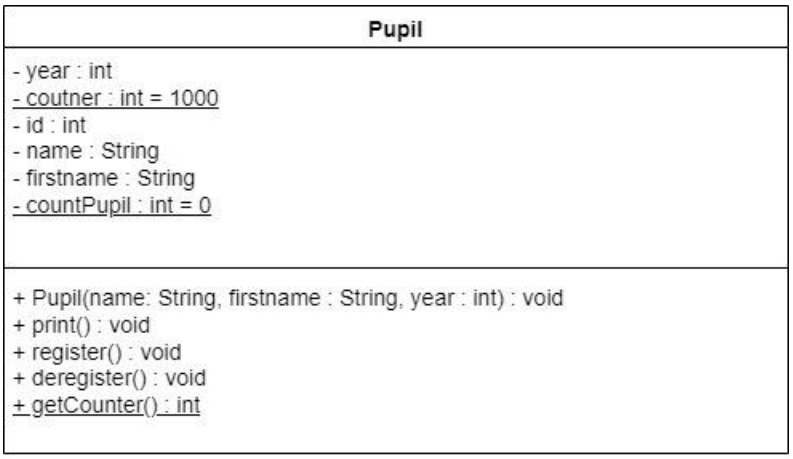

Abbildung 23: UML-Klassendiagramm der Klasse Pupil

Die Attribute werden im oberen Teil angegeben und werden mit der Struktur
`Zugriffsmodifikator Name : Typ [= Startwert]`
dargestellt.
Zugriffsmodifikator:

- **−**: **private**
- **+**: **public**
- **#**: **protected**

Name ist der Name des Attributes.
Typ ist der Datentyp.
[= Startwert] ist optional für den Initialwert.
Unterstrich: Klassenvariable, d.h. **static.**

Daraus ist ersichtlich, dass **`– year : int`** ein privates Attribut vom Typ Integer ist.

Bei den Methoden ist es analog, nur gibt es hier keinen Startwert und man muss die Parameter auch angeben.
`Zugriffsmodifikator Name(Name1 : Typ, Name2 : Typ, …) : Typ`

Somit ist **`+getCounter() : int`** eine **`public`** Klassenmethode (**`static`**) ohne Parameter mit einem Integer als Rückgabewert.

Aufgabe 43

Erstellen Sie analog zum vorherigen Beispiel ein UML-Klassendiagramm für die Klasse **`Time`** aus der Aufgabe 24.

Lösung:

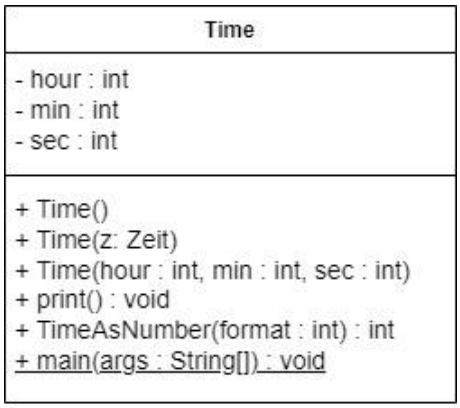

Abbildung 24: Klasse Time in UML

Beispiel 37

Die Klasse **`Line`** besteht aus zwei Attribute der Klasse **`Point`**, dies wird in der Fachsprache Aggregation genannt und in UML mit einer Raute und einer Linie dargestellt, am Ende der Linie steht die Anzahl der Objekte der Klasse **`Point`**. In der Abbildung 25 wird dies in UML dargestellt.

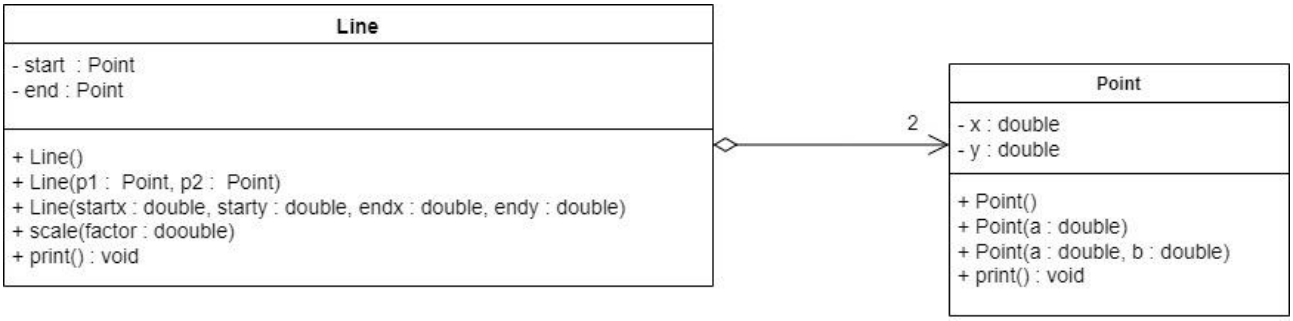

Abbildung 25: Klasse Line mit Klasse Point in UML

3.10 Überladen von Methoden

Es kommt häufig vor, dass man dieselbe Operation für verschiedene Parameter in einer Klasse implementieren möchte. Anstatt in diesem Fall jeweils einen neuen Namen für die Methode zu suchen, dürfen in einer Klasse mehrere Methoden mit demselben Namen existieren. Die einzige Bedingung ist, dass man verschiedene Parameter haben muss, das bedeutet entweder verschiedene Datentypen oder eine andere Anzahl Parameter.

Beispiel 38

Ein Beispiel, welches wir bis jetzt verwendet haben, ist die Ausgabe der Daten auf die Konsole. Wir können der Methode **`System.out.println(…)`** verschiedene Parameter mitgeben.

```
public class Beispiel38 {
    public static void main(String[] args){
        //Ausgabe eines Integer
        int a = 56;
        System.out.println(a);
        //Ausgabe eines Doubles
        double d = 3.1459;
        System.out.println(d);
        //Ausgabe eines Booleans
        boolean b = true;
        System.out.println(b);
        //Text Ausgabe
        System.out.println("Hello World!");
    }
}
```

Die Methode **`System.out.println(…)`** funktioniert für verschieden Datentypen. Dies wird überladen von Methoden genannt.
Dasselbe Konzept haben wir bereits bei den verschiedenen Konstruktoren benutzt, die Konstruktoren heissen alle gleich wie die Klasse selbst und haben verschiedene Parameter.

Aufgabe 44

Erweitern Sie die Klasse **`Fractional`** aus Beispiel 34 und Aufgabe 41 um zwei weitere **`add`**-Methoden, eine um eine Addition mit einer ganzen Zahl und eine um eine Addition mit Zähler und Nenner zu berechnen, also **`Fractional add(int zahl)`** und **`Fractional add(int z, int n)`**.

Lösung:
Sie sehen hier nur den Zusatz der drei überladenen Methoden **`add(…)`**.

```
public Fractional add(Fractional other){
    Fractional res = new Fractional();
    res.num = num * other.denom + other.num * denom;
    res.denom = denom * other.denom;
    res.kuerzen();
    return res;
}
```

```
public Fractional add(int zahl){
    Fractional res = new Fractional();
    res.num = num + zahl * denom;
    res.denom = denom;
    res.kuerzen();
    return res;
}
public Fractional add(int z, int n){
    Fractional res = new Fractional();
    res.num = num * n + z * denom;
    res.denom = denom * n;
    res.kuerzen();
    return res;
}
```

```
public static void main(String[] args){
    Fractional b1 = new Fractional(1, 4);
    Fractional b2 = new Fractional(1, 4);
    Fractional res = b1.add(b2);
    res.print();
    res = b1.add(3);
    res.print();
    res = b2.add(2, 3);
    res.print();
}
```

3.11 Destruktoren

Neben Konstruktoren, die bei der Initialisierung eines Objekts aufgerufen werden, gibt es in Java wie auch in anderen OO-Programmiersprachen **Destruktoren**. Sie werden unmittelbar vor dem Freigeben des Speicherbereiches eines Objektes, d.h. vor dem Löschen des Speichers, aufgerufen. Das Problem in Java ist, dass das Speichermanagement vom System erledigt wird und obwohl ein Objekt nicht mehr sichtbar ist, dieses sich immer noch im Speicher befinden kann. D.h. es ist nicht definiert, wann der Destruktor aufgerufen wird. In Java erledigt der sog. «**Garbage Collector»** das Aufräumen des Speichers. Damit wir in unserem Beispiel den Aufruf des Destruktors sehen, werden wir den «Garbage Collector» direkt mit **`System.gc();`** aufrufen, was aber wohl bemerkt **nicht** so programmiert werden soll.
Der Einsatz von Destruktoren in Java sollte also mit der nötigen Vorsicht erfolgen oder gar nicht.

Beispiel 39

Dazu verwenden wir die Klasse **`ParameterTest`** aus Beispiel 32 und erweitern die **`main-`** Methoden um die folgenden Zeilen.

```
pT.param = null;
p = null;
//Garbage Collector Aufruf. Sollte nicht gemacht werden.
System.gc();
```

In der Klasse **`Parameter`** aus Beispiel 32 fügen wir den Destruktor **`protected void finalize()`** hinzu.

```
protected void finalize(){
    System.out.println("class Parameter wird gelöscht.");
}
```

Falls wir nun das Programm starten, dann sehen wir in der letzten Zeile in der Konsole, dass der Destruktor aufgerufen worden ist.
class Parameter wird gelöscht.
Wir gehen an dieser Stelle nicht tiefer auf diesen Punkt ein, da es für diese Arbeit keine Relevanz hat.

4 Einfach verkettete Listen

In Java existiert bereits eine Klasse für die verkettete Liste, die Klasse **`LinkedList`**. Trotzdem möchten wir in diesem Kapitel erklären, wie eine verkettete Liste aufgebaut ist, um die in den vorherigen Kapiteln behandelten Themen zu vertiefen.

4.1 Aufbau

Es existieren zwei möglichen Varianten der verketteten Listen, die einfach verkettete Liste und die doppelt verkettete Liste, der Aufbau beider Typen ist sehr ähnlich.
In der Abbildung 26 ist der Aufbau einer einfach verketteten Liste graphisch dargestellt.

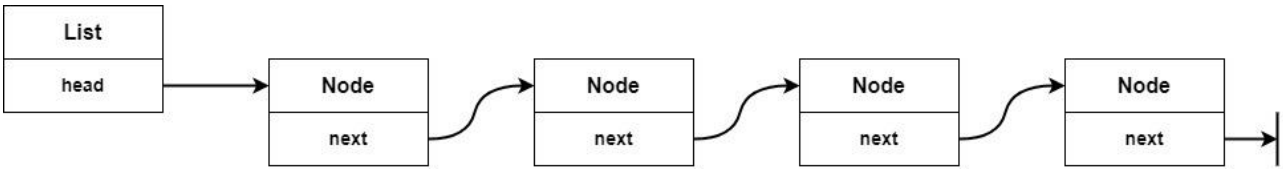

Abbildung 26: einfach verkettete Liste

Die Liste hat einen Anker (**`head`**), welcher auf den ersten Knoten (**`Node`**) zeigt und jeder Knoten hat einen Zeiger auf den nächsten Knoten (**`next`**). Der Zeiger des letzten Knotens zeigt auf keinen Speicherbereich, in Java bedeutet dies, dass er auf **`null`** zeigt.
Gemäss obiger Beschreibung hat die Klasse **`List`** einen Anker (**`head`**) , dies ist in der UML-Darstellung in Abbildung 27 der Klasse **`List`** zu sehen. Das Objekt **`head`** ist von der Klasse **`Node`** und zeigt beim Erstellen der Liste auf keinen Speicherbereich, d.h. der Wert von **`head`** ist **`null`**.

Abbildung 27: UML der Klasse List und Node (ohne Methoden)

Die Klasse **`Node`** ist die Vorlage für alle Knoten der Liste. Das erste erzeugte Objekt der Klasse **`Node`** wird dem Attribut **`head`** des Objektes **`myList`** der Klasse **`List`** zugewiesen. So wird erreicht, dass der erste Knoten über das Objekt **`myList.head`** angesprochen werden kann. Das zweite Objekt der Klasse **`Node`**, wird dem Objekt **`next`** des ersten Objektes zugewiesen usw. Schlussendlich können wir mit diesem Vorgehen eine Liste, wie sie in der Abbildung 26 dargestellt ist, mit der gewünschten Anzahl Knoten erstellen.

Eine Liste benutzt man, um Daten zu verwalten, in unserem Beispiel fehlen diese noch. Die Daten gehören zum Knoten und damit es hier nicht zu komplex wird, werden wir für den Anfang als Daten nur eine ganze Zahl (Integer) verwenden. Wir werden in einem späteren Kapitel sehen, wie dies zum Teil verallgemeinert werden kann, sodass nicht jedes Mal eine neue Liste mit neuen Knoten erstellt werden muss. Dafür benötigen wir das Konzept der Vererbung aus Kapitel 5 «Vererbung». Eine vollkommene Verallgemeinerung wäre die Implementierung wie sie in der **`LinkedList`** von Java zu finden ist. Dazu benötigt man das Konzept des sog. «generic type», was nicht in dieser Arbeit behandelt wird.

In den weiteren Unterkapitel werden in den Beschreibungen direkt auf die Attribute der Klasse **`List`** und **`Node`** zugegriffen, damit die Arbeit lesbarer wird. Das bedeutet, dass wir in den Beschreibungen keine «**setter»** und «**getter**» benutzen werden, sondern direkt **`myList.head`**

oder **`node.next`** schreiben. Wobei **`myList`** ein Objekt der Klasse **`List`** ist und **`node`** ein Objekt der Klasse **`Node`** ist. Später in der Implementierung werden wir die Daten mit **`private`** schützen, sodass wir die «**setter**» und «**getter**» benötigen werden.

4.2 Knoten einfügen

4.2.1 Am Anfang einfügen

Aufgabe 45

Überlegen Sie sich, was Sie alles tun müssen, um ein Element bei einer bestehenden Liste am Anfang einzusetzen. Die Schritte müssen nicht programmiert werden, sondern es genügt eine graphische Darstellung.

Lösung:
Schritt 1: Neues Objekt **`nodeNew`** der Klasse **`Node`** erzeugen (Abbildung 28).

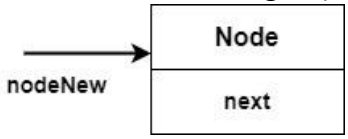

Abbildung 28: Schritt 1: Objekt nodeNew erzeugen.

Schritt 2: **`next`** vom neuen Objekt **`nodeNew`** auf das bisherige erste Element setzen (Abbildung 29).

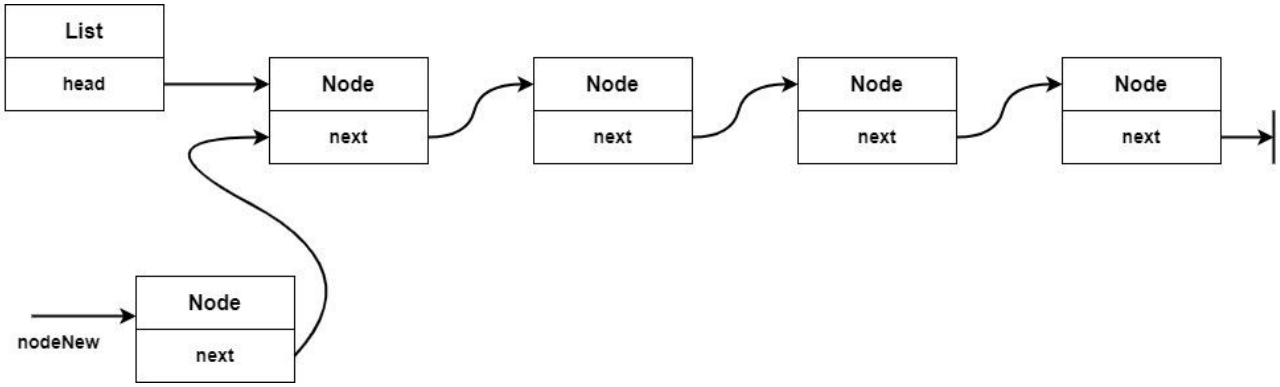

Abbildung 29: Schritt 2: next von nodeNew auf das bisherige erste Element setzen.

Schritt 3: Objekte **`nodeNew`** an Objekt **`head`** der Liste zuweisen (Abbildung 30).

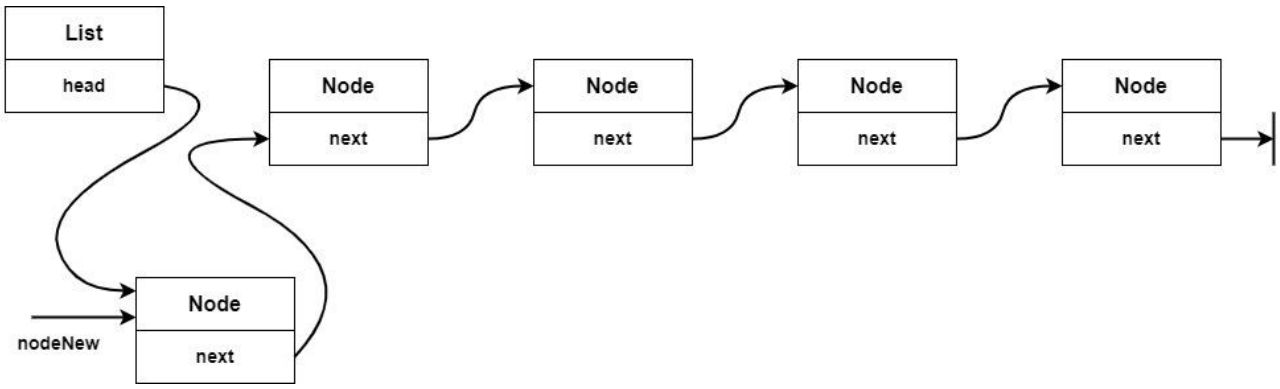

Abbildung 30: Schritt 3: nodeNew an head der Liste zuweisen.

4.2.2 Am Ende einfügen

Aufgabe 46

Überlegen Sie sich, was Sie alles tun müssen, um ein Element bei einer bestehenden Liste am Ende einzusetzen. Die Schritte müssen nicht programmiert werden, sondern es genügt eine graphische Darstellung.

Lösung:
Schritt 1: Zuerst muss man ans Ende navigieren. Das Ende ist erreicht, falls die Referenz **`next`** des Objektes der Klasse **`Node`** auf **`null`** zeigt (Abbildung 31).

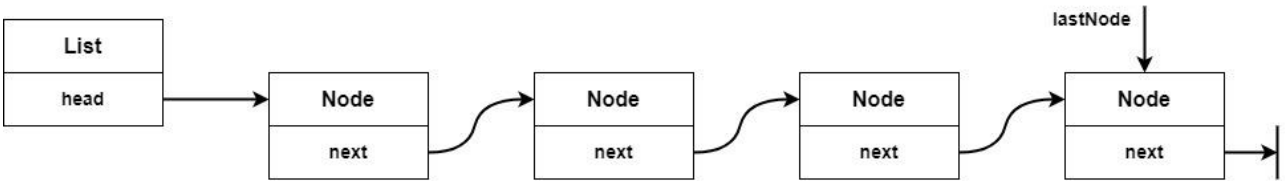

Abbildung 31: Schritt 1: Zum letzten Knoten navigieren.

Schritt 2: Neues Objekt **`nodeNew`** der Klasse **`Node`** erzeugen, dabei wird **`next = null`** im Konstruktor gesetzt (Abbildung 32).

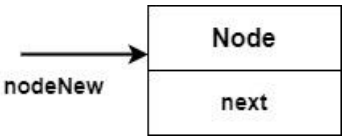

Abbildung 32: Schritt 2: Neues Objekt nodeNew erzeugen.

Schritt 3: Das Objekt **`nodeNew`** an die Referenz **`next`** vom letzten Knoten in der Liste zuweisen (Abbildung 33).

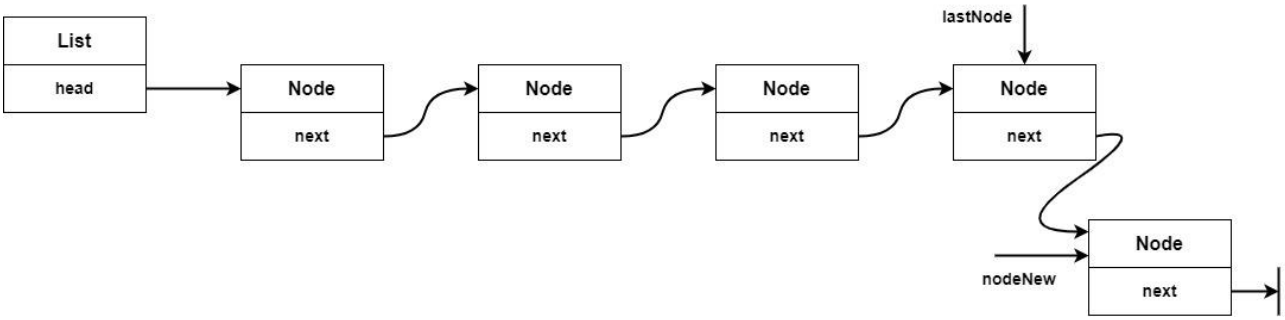

Abbildung 33: Schritt3: Objekt nodeNew an Referenz next von Objekt lastNode zuweisen.

4.2.3 Irgendwo einfügen

Einen neuen Knoten irgendwo in der Liste einfügen, wird hier als Beispiel gezeigt. Gemeint ist nicht am Anfang und nicht am Ende einfügen. Die Schritte haben Sie sich vorhin alle überlegt, man muss sie nur noch zusammenfügen.

Schritt 1: Neues Objekt **`nodeNew`** erzeugen (Abbildung 34).

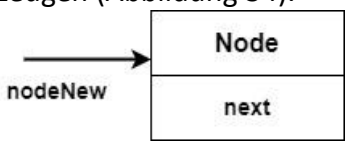

Abbildung 34: Schritt 1: neues Objekt nodeNew erzeugen

Schritt 2: Die Position **`pos`** nach dem dieser neuen Knoten eingefügt werden soll, sei bereits bekannt (Abbildung 35).

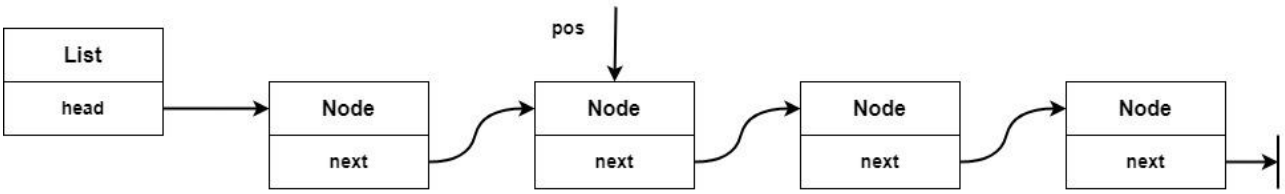

Abbildung 35: Schritt 1: Position für das Einfügen bestimmen.

Schritt 3: Referenz **`next`** vom Objekt **`pos`** der Referenz **`next`** des Objektes **`nodeNew`** zuweisen also **`nodeNew.next = pos.next`** (Abbildung 36).

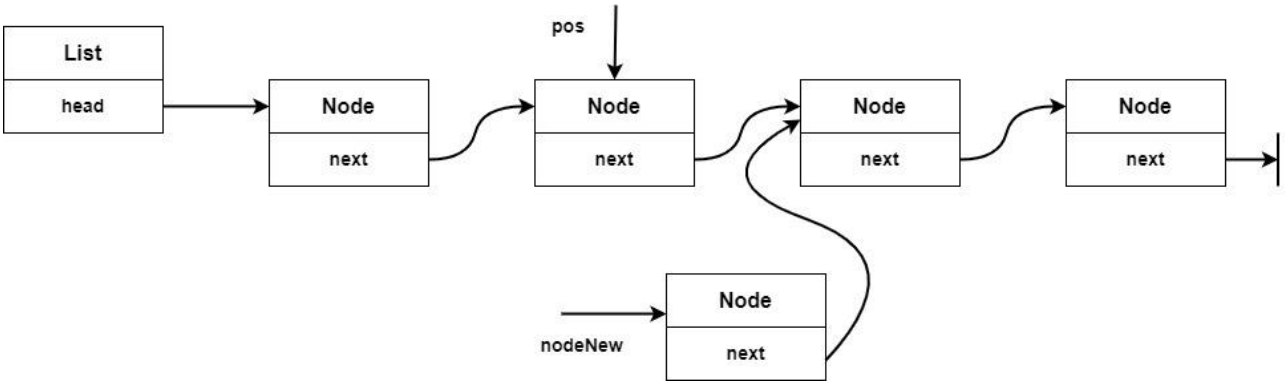

Abbildung 36: Schritt 3: nodeNew.next = pos.next

Schritt 4: Objekt **`nodeNew`** der Referenz **`next`** vom Objekt **`pos`** zuweisen, also **`pos.next = nodeNew`** (Abbildung 37).

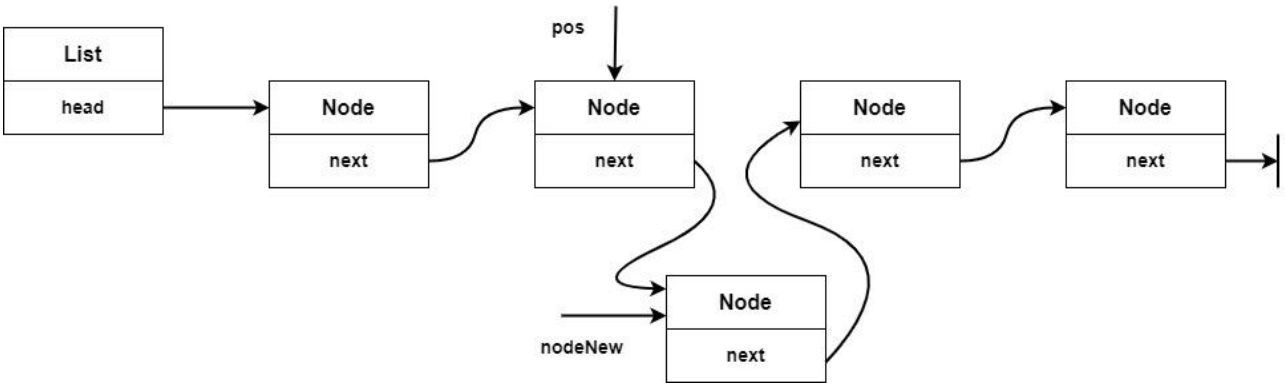

Abbildung 37: Schritt 4: pos.next = nodeNew

4.3 Knoten löschen

4.3.1 Am Anfang löschen

Wir werden nun den ersten Knoten in der Liste löschen.

Schritt 1: Es wird eine Referenz auf das erste Element gesetzt, d.h. **`head`** wird einer neuen Referenz **`pos`** zugewiesen, also **`pos = head`** (Abbildung 38).

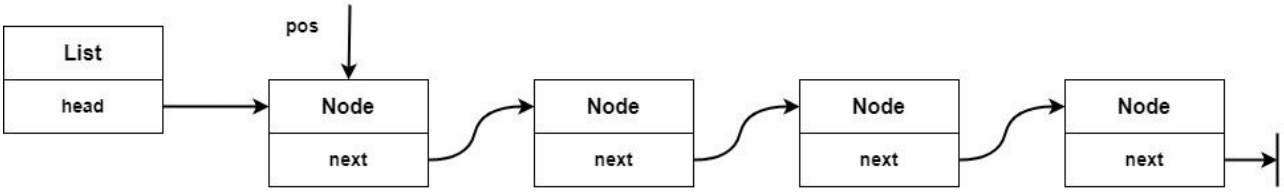

Abbildung 38: Schritt 1: pos = head

Schritt 2: Dem Objekt **`head`** wird die Referenz **`next`** des Objektes **`pos`** zugewiesen, also **`head = pos.next`** (Abbildung 39).

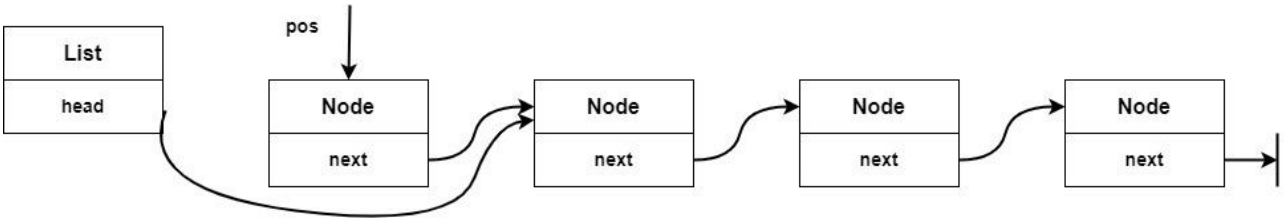

Abbildung 39: Schritt 2: head = pos.next

Schritt 3: Erstes Element löschen, in Java bedeutet dies auf **`null`** setzen. Damit wird der Speicherbereich durch den «Garbage Collector» gelöscht, also **`pos.next = null`** und **`pos = null`** (Abbildung 40).

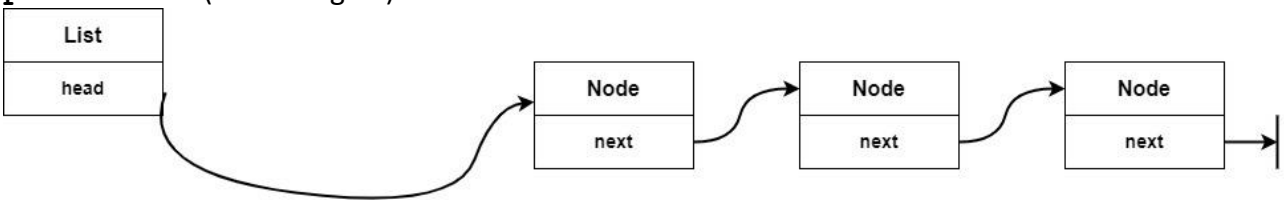

Abbildung 40: Schritt 3: pos.next = null; pos = null

4.3.2 Am Ende löschen

Man muss zum zweitletzten Knoten navigieren (Abbildung 41). Beim zweitletzten Knoten zeigt das Objekt **`next`** auf einen Knoten, dessen Referenz **`next`** auf null zeigt.

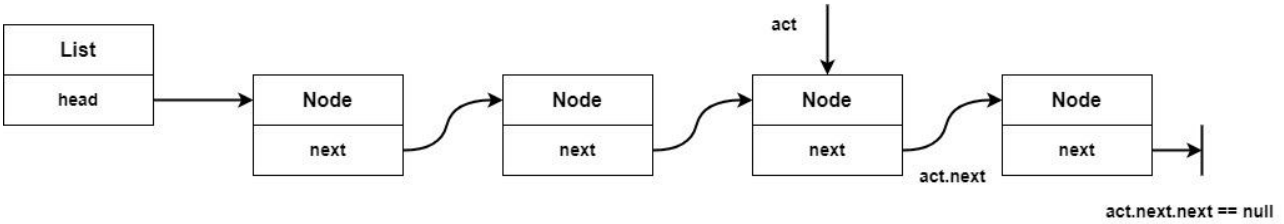

Abbildung 41: Navigieren bis zum vorletzten.

Dann kann das Objekt **`next`** an der aktuellen Position auf **`null`** gesetzt werden.

Mit Pseudocode ausgedrückt:

```
Falls act.next.next == null
Dann act.next = null
```

Dies bewirkt, dass der Speicherbereich gelöscht wird (Abbildung 42).

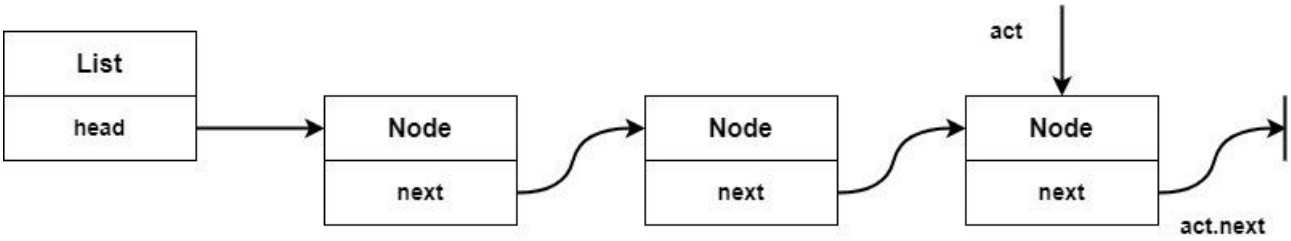

Abbildung 42: Letzter Knoten löschen.

4.3.3 Irgendwo löschen

Um einen Knoten irgendwo in der Liste zu löschen, benötigen wir auch den Vorgänger, damit wir die Verknüpfungen wieder herstellen können.

Schritt 1: Das zu löschende Objekt **`pos`** wird vorgegeben (Abbildung 43).

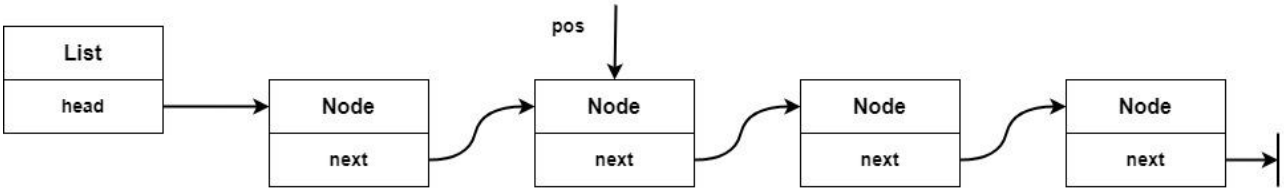

Abbildung 43: Ausgangslage

Schritt 2: Vorgänger vom Objekt **`pos`** suchen, liefert uns das Objekt **`prevPos`**. Die Referenz **`next`** des Vorgängers **`prevPos`** ist gleich der Referenz **`pos`**, d.h. beide zeigen auf denselben Speicherbereich (Abbildung 44).

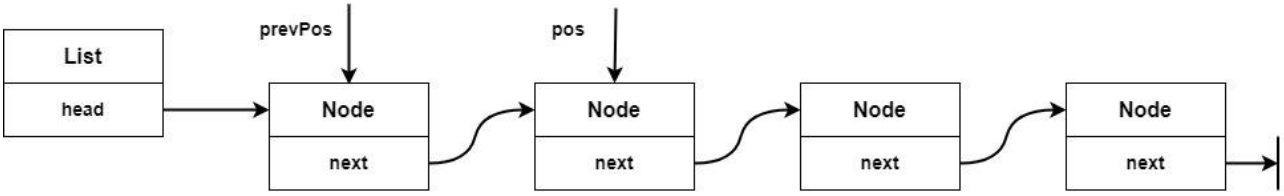

Abbildung 44: Schritt 2: Vorgänger von pos suchen.

Schritt 3: Der Referenz **`next`** des Objektes **`prevPos`** wird die Referenz **`next`** des Objektes **`pos`** zugewiesen (Abbildung 45). Mit der obigen Notation ist damit folgende Anweisung gemeint **`prevPos.next = pos.next`**.

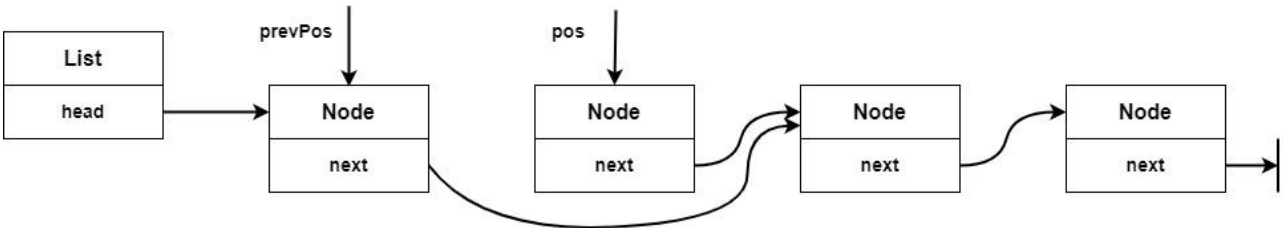

Abbildung 45: Schritt 3: prevPos.next = pos.next

Schritt 4: Nun kann die Referenz **`next`** des Objektes **`pos`** auf **`null`** gesetzt werden sowie auch die Referenz **`pos`** selbst, damit wird der Speicherbereich durch den «Garbage Collector» gelöscht resp. aufgeräumt (Abbildung 46).

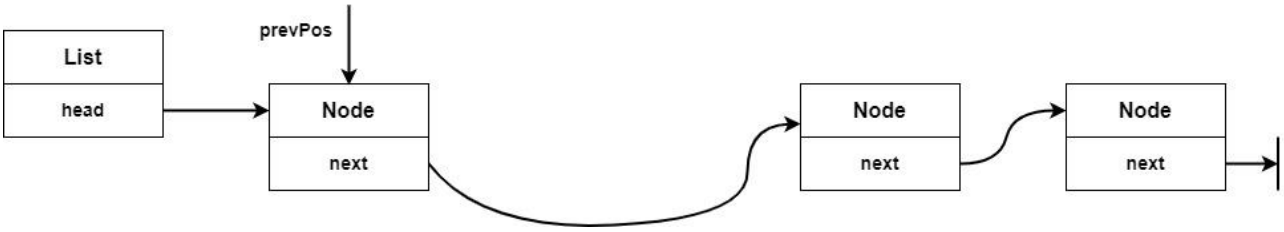

Abbildung 46: Schritt 4: nicht mehr benötigte Objekte löschen.

4.4 Suchen

Die obigen Funktionalitäten wurden ohne Daten beschrieben, falls wir nun einen Wert in der Liste suchen möchten, müssen wir dem Knoten, d.h. der Klasse **`Node`** Daten hinzufügen (Abbildung 47). Wie bereits erwähnt, werden wir eine ganze Zahl als Daten der Klasse **`Node`** hinzufügen.

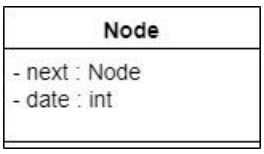

Abbildung 47: Klasse Node mit Daten

Aufgabe 47

Schreiben Sie eine Methode **`search(…)`** mit Pseudocode, mit der nach einem Wert in der Liste gesucht wird. Da es nur um die Logik geht, kann in dieser Aufgabe, falls Sie ein Objekt **`node`** der Klasse **`Node`** haben, mit **`node.next`** und **`node.data`** auf die Attribute zugegriffen werden. Die Methode **`Node search(int data)`** liefert das Objekt mit den gesuchten Daten zurück, falls kein Objekt gefunden wird, liefert die Methode **`null`**.

Lösung:

```
Search(data){
      tmpNode = head
      solange (tmpNode.data != data und tmpNode.next != null){
            tmpNode = tmpNode.next
      }
      falls tmpNode.data == data{
            return tmpNode
      }sonst{
            return null
      }
}
```

4.5 Methoden der Klasse List

Aus den obigen Funktionalitäten (einfügen, löschen, suchen) erhalten wir somit folgende Methoden für die Klasse **`List`** und für die Klasse **`Node`**.

Klasse List:
Konstruktoren der Klasse **`List`**:
```
List()
List(Node head)
```

Methoden in der Klasse **`List`**:

`void setHead(Node node)`:	Der Anker der Liste wird gesetzt.
`Node getHead()`:	Der Anker der Liste wird zurückgegeben.
`void insertBegin(Node node)`:	Der Knoten **`data`** wird als erstes Element eingefügt.
`void insertBegin(int data)`:	Es wird einen neuen Knoten mit **`data`** erzeugt und als erstes Element hinzugefügt.
`void insertEnd(Node node)`:	Der Knoten **`data`** wird ans Ende der Liste gesetzt.
`void insertEnd(int data)`:	Es wird einen Knoten mit **`data`** erzeugt und als letztes Element in der Liste gesetzt.
`void insertPos(Node node, Node pos)`:	Der Knoten **`data`** wird nach **`pos`** eingefügt.
`void insertPos(int data, Node pos)`:	Es wird einen neuen Knoten mit **`data`** erzeugt. Dieser wird nach **`pos`** eingefügt.
`boolean remove(Node node)`:	Der Knoten **`data`** wird gelöscht. War das Löschen erfolgreich, wird **`true`** zurückgegeben, sonst **`false`**, z.B. falls der Knoten nicht in der Liste ist.
`Node search(int data)`:	Sucht in der Liste nach einem Knoten mit der Zahl **`data`** und gibt den Knoten (**`Node`**) zurück, falls nichts gefunden wurde, wird **`null`** zurückgegeben.

Klasse Node:
Konstruktoren der Klasse **`Node`**:
```
Node()
Node(Node next, int data)
```

Methoden der Klasse **`Node`**:

`void setData(int data)`:	Die Zahl **`data`** wird gesetzt.
`int getData()`:	Die Zahl des Knotens wird zurückgegeben.
`void setNext(Node node)`:	Das Objekt **`next`** wird auf denselben Speicherbereich wie das Objekt **`node`** gesetzt.
`Node getNext()`:	Das Objekt **`next`** wird zurückgegeben.

4.5.1 Vollständiges UML

Nun können wir das vollständige UML (Abbildung 48) angeben.

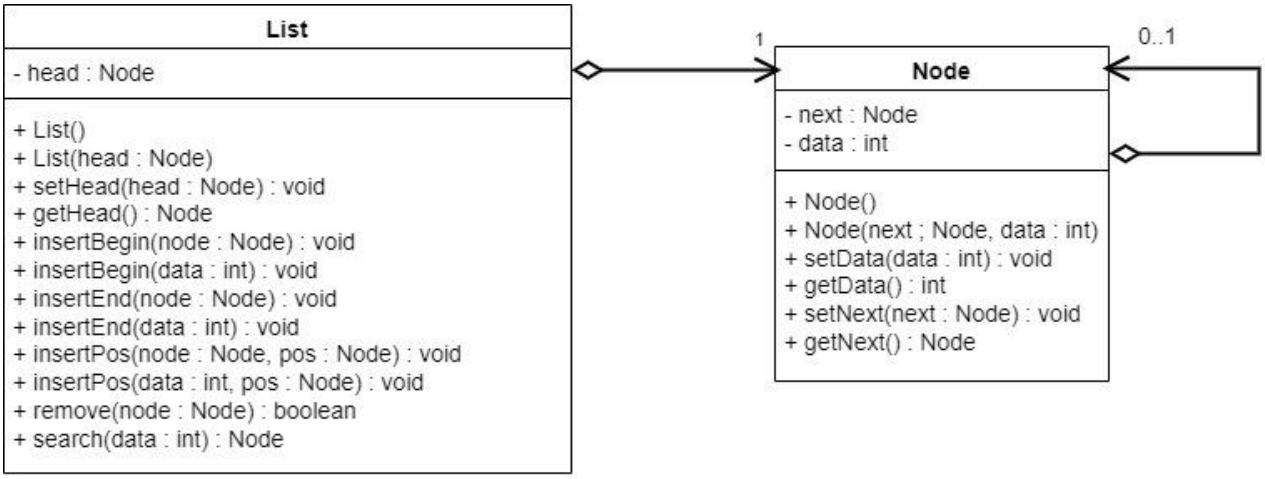

Abbildung 48: UML der Klasse List und der Klasse Node

4.5.2 Code

Aufgabe 48

Diese Aufgabe ist vom Umfang her wie ein kleines Projekt.
Erstellen Sie nun die Klassen **`List`** und die Klasse **`Node`** in Java mit allen notwendigen Attributen und Methoden, entsprechend dem UML-Klassendiagramm.
Implementieren Sie danach alle Methoden der Klasse **`Node`** und der Klasse **`List`.**

Lösung:

```
public class Node {
    private Node next;
    private int data;
    public Node(){
        data = 0;
        next = null;
    }

    public Node(Node next, int data){
        this.data = data;
        this.next = next;
    }

    public void setData(int data){
        this.data = data;
    }

    public int getData(){
        return data;
    }
```

```
    public void setNext(Node next){
        this.next = next;
    }

    public Node getNext(){
        return next;
    }
}
```

```
public class List {
    private Node head;

    public List(){
        head = null;
    }

    public List(Node head){
        this.head = head;
    }

    public void setHead(Node head) {
        this.head = head;
    }
    public Node getHead(){
        return head;
    }

    public void insertBegin(Node node){
        node.setNext(head);
        head = node;
    }

    public void insertBegin(int data){
        Node node = new Node(head, data);
        head = node;
    }

    public void insertEnd(Node node){
        //ans Ende navigieren
        Node lastNode = head;
        if (head != null){
            while (lastNode.getNext() != null){
                lastNode = lastNode.getNext();
            }
            lastNode.setNext(node);
        }else{
            //erstes Element!
            head = node;
        }
        node.setNext(null);
    }
```

```
    public void insertEnd(int data){
        Node node = new Node(null, data);
        insertEnd(node);
    }

    public void insertPos(Node node, Node pos){
        node.setNext(pos.getNext());
        pos.setNext(node);
    }

    public void insertPos(int data, Node pos){
        Node node = new Node(null, data);
        insertPos(node, pos);
    }
    public boolean remove(Node node){
        //Achtung node könnte nicht in der Liste sein!
        Node pos = head;
        if (head == node){
            //Am Anfang löschen
            head = head.getNext();
            //aufräumen
            pos.setNext(null);
            pos = null;
            return true;
        }else if (node.getNext() == null){
            //Am Ende löschen, falls node in der Liste ist
            //zum Vorgänger navigieren
            while (pos.getNext().getNext() != null){
                pos = pos.getNext();
            }
            if (pos.getNext() != node){
                //node ist nicht der letzte Knoten!
                return false;
            }
            pos.setNext(null);
            //aufräumen
            node = null;
            return true;
        }else{
            //irgendwo löschen
            //zum Vorgänger navigieren
            while (pos.getNext() != node){
                pos = pos.getNext();
                if (pos == null){
                    //node ist nicht der letzte Knoten!
                    return false;
                }
            }
            pos.setNext(node.getNext());
            //aufräumen
```

```
            node.setNext(null);
            node = null;
            return true;
        }
    }

    public Node search(int data){
        Node pos = head;
        if (head.getData() == data){
            return head;
        }
        while (pos.getNext() != null){
            pos = pos.getNext();
            if (pos.getData() == data){
                return pos;
            }
        }
        return null;
    }

    //oder mit do while
    public Node searchV2(int data){
        Node pos = head;

        do{
            if (pos.getData() == data){
                return pos;
            }
            pos = pos.getNext();
        }while (pos != null);

        return null;
    }

    //Addon
    public void print(){
         Node pos = head;
         do{
            System.out.println("Data: " + pos.getData());
            pos = pos.getNext();
         }while (pos != null);
    }
}
```

```
public class TestList {
    public static void main(String[] args){
        List myList = new List();
        Node node = new Node(null, 10);
        myList.insertBegin(node);
        node = new Node(null, 23);
        myList.insertBegin(node);
```

```
        node = new Node(null, 56);
        myList.insertBegin(node);
        node = new Node(null, 7);
        myList.insertBegin(node);
        node = new Node(null, 343);
        myList.insertBegin(node);
        node = new Node(null, 221);
        myList.insertBegin(node);
        node = new Node(null, 33);
        myList.insertBegin(node);
        node = new Node(null, 55);
        myList.insertBegin(node);
        node = new Node(null, 99);
        //myList.insertBegin(node);
        myList.insertEnd(node);
        node = new Node(null, 1);
        myList.insertBegin(node);

        myList.print();

        Node res = myList.search(343);
        //Node res = myList.search(342);
        node = new Node(null, 1000);
        if (res != null){
            myList.insertPos(node, res);
            myList.remove(res);
        }

        myList.print();
    }
}
```

5 Vererbung

5.1 Einführung

Bei der Vererbung erbt eine sog. abgeleitete Klasse alle **Eigenschaften** (Datenfelder und Methoden – also die Struktur und die Operationen) ihrer Basisklasse und fügt ihre eigenen individuellen Eigenschaften hinzu, damit wird eine Spezialisierung der Basisklasse erreicht.

Am einfachsten soll dies anhand des Beispiels eines Studenten erläutert werden. Ein Student ist eine Person, die studiert. Wenn man studieren möchte, muss man immatrikuliert werden und man erhält eine Matrikelnummer. Kurz, wer eine Matrikelnummer hat, ist eingeschrieben und ist somit ein Student. Also kann man einen Studenten als eine Person beschreiben, die eine Matrikelnummer hat, somit ist der Student eine spezielle Person. Im Folgenden erfolgt die Visualisierung dieser Vererbungsbeziehung in UML (Abbildung 49).

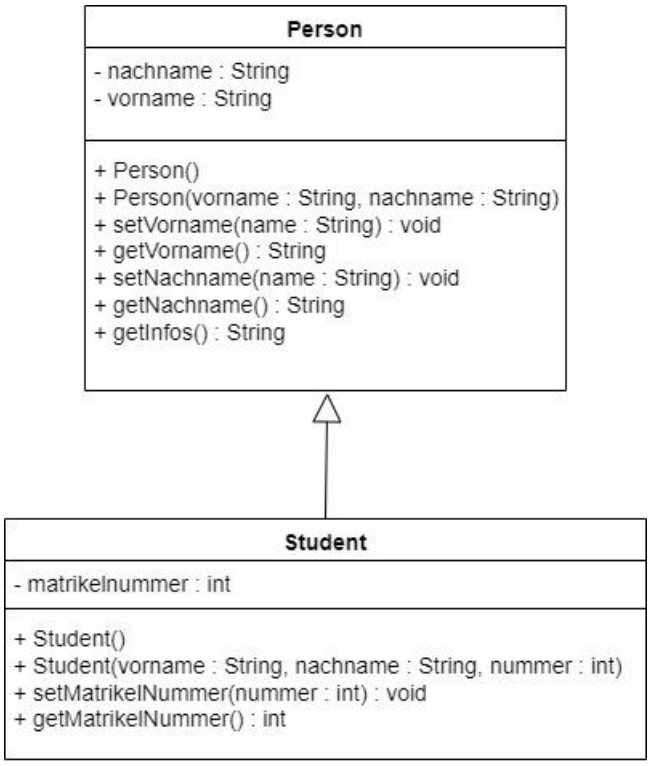

Abbildung 49: Vererbung Person - Student

5.1.1 Begriffe

In diesem Kapitel erscheinen neue Begriffe, welche folgend erklärt werden.
Basisklasse: Die Basisklasse ist in der Abbildung 49 die Klasse **`Person`**. Diese Klasse wird auch **Oberklasse** genannt. Die Basisklasse ist eine Generalisierung aller unteren Klassen. Sie wird auch **Superklasse** genannt.

Abgeleitete Klasse: Die Klasse **`Student`** ist eine **abgeleitete Klasse**. Sie erbt alle Datenfelder und Methoden der Basisklasse, das bedeutet, dass alle Attribute und Methoden der Klasse **`Person`** in der Klasse **`Student`** enthalten sind und ein Objekt der Klasse **`Student`** kann sie nutzen, als wären sie direkt in der Klasse **`Student`** definiert worden. Da die Klasse **`Student`** eine Unterart der Basisklasse ist, wird sie auch **Unterklasse** genannt. Ein weiterer Begriff für diese Klasse ist der Begriff **Subklasse**.

Vererbung: Die Klasse **`Student`**, wie oben bereits erwähnt, enthält alle Datenfelder und Methoden der Klasse **`Person`**. Man sagt, dass die Klasse **`Student`** von der Klasse **`Person`** erbt. Der Programmierer muss nur die neuen Attribute und Methoden hinzufügen, was zu einer Spezialisierung der Oberklasse **`Person`** führt.

Spezialisierung: Eine Subklasse spezialisiert die Superklasse, d.h. sie erweitert die Operationen der Superklasse.

Generalisierung: Im Gegensatz zur Spezialisierung ist die Oberklasse eine Generalisierung aller Unterklassen.

Wenn wir nun einen Assistenten betrachten, so ist er auch ein Student, welcher in den unteren Semester die Übungen betreut. In OOP könnte man dies so darstellen, dass die Klasse **`Assistant`** von der Klasse **`Student`** abgeleitet wird. Diese Beziehung, «ist ein» oder auf Englisch «is a», wird in UML durch den Vererbungspfeil angedeutet. Also ist der Student eine Person und ein Assistent ist ein Student.
Hingegen ist der Professor kein Student und kein Assistent, deshalb müsste man die Klasse **`Professor`** von der Klasse **`Person`** direkt ableiten (Abbildung 50). Diese Darstellung wird **UML-Klassendiagramm** genannt.

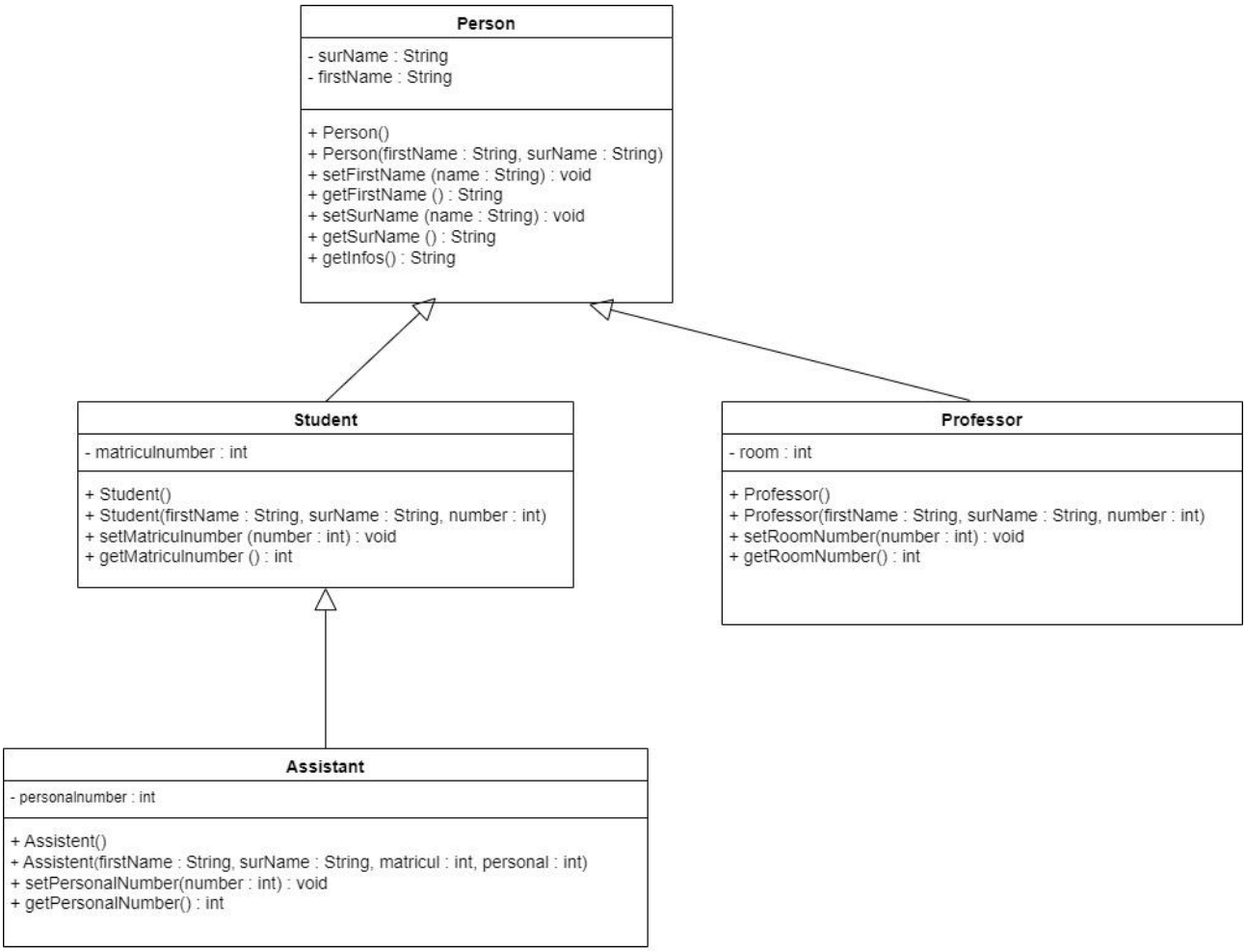

Abbildung 50: UML-Klassendiagramm der Klassen: Person, Student, Assistant und Professor

Aufgabe 49

Überlegene Sie sich ein UML-Klassendiagramm für folgende Arzneimittel (Pharmaceutical): Tabletten (Tablet), Salbe (Ointment), Augensalbe (EyeOintment) und Magentabletten (StomachTablet)
Zeichnen Sie im Klassendiagramm die Klassen nur mit dem Namen ein, d.h. ohne Attribute und ohne Methoden.

Lösung:

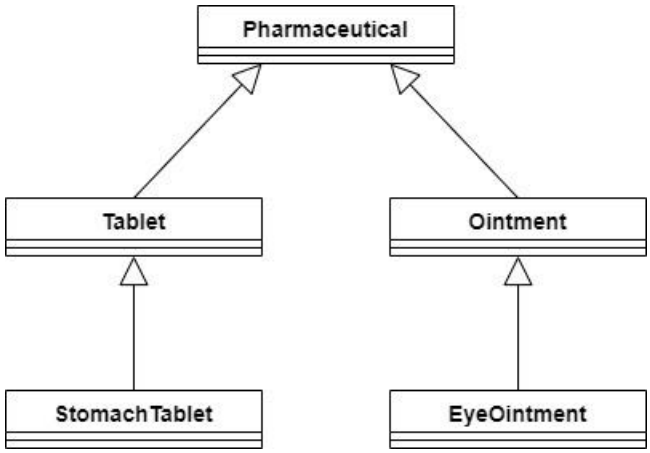

Abbildung 51: UML-Klassendiagramm der Arzneimittel (Pharmaceutical)

Beispiel 40

Der Java Code für die zwei Klassen **Person** und **Student** aus der Abbildung 49 kann mit dem oben erworbenen Wissen folgendermassen geschrieben werden.

```
public class Person {
    private String surName;
    private String firstName;

    public Person(){
        surName = "";
        firstName = "";
    }
    Person(String surName, String firstName){
        this.surName = surName;
        this.firstName = firstName;
    }
    public void setSurName(String surName){
        this.surName = surName;
    }
    public String getSurName(){
        return surName;
    }
    public void setFirstName(String firstName){
        this.firstName = firstName;
    }
    public String getFirstName(){
        return firstName;
    }
    public String getInfos(){
        String info = surName + " " + firstName;
        return info;
    }
}
```

In Java wird die Vererbung mit dem Schlüsselwort **extends** deklariert, im Grunde genommen wird bei der Vererbung die Klasse **Person** von der Klasse **Student** erweitert. Die Klasse **Student** hat nun alle Attribute und Methoden der Klasse **Person** geerbt und kann diese direkt nutzen oder neu definieren.

```
public class Student extends Person{
    private int matriculNumber;
    public Student(){
        firstName = "";
        surName = "";
        matriculNumber = 0;
    }
```

```
public Student(String surName, String firstName,
                    int matriculNumber){
        this.firstName = firstName;
        this.surName = surName;
        this.matriculNumber = matriculNumber;
    }
    public void setMatriculNumber(int number){
        this.matriculNumber = number;
    }
    public int getMatriculNumber (){
        return matriculNumber;
    }
}
```

```
public class TestStudent {
    public static void main(String[] args){
        Student stud = new Student("Meier", "Paul", 90446345);
        System.out.println(stud.getInfos());
        System.out.println(stud.getMatriculNumber ());
    }
}
```

Falls Sie nun versuchen das Testprogramm zu starten erhalten Sie einen Fehler.
«**The field Person.firstName is not visible**»
«**The field Person.surName is not visible**»

Bei der näheren Betrachtung des Problems, sehen wir, dass die Felder **`firstName`** und **`surName`** der Klasse **`Person`** **`private`** sind und bemerken, dass **`private`** zur Folge hat, dass nur die Klasse **`Person`** selbst darauf zugreifen kann. Also benötigen wir etwas anderes, nämlich ein Zugriffsrecht, sodass der Zugriff auf diese Felder auch der Subklasse erlaubt wird, dafür existiert das Schlüsselwort **`protected`**. Mit **`protected`** sind die Felder gleich wie bei **`private`** geschützt, ausser dass eine Subklasse auch auf diese Attribute und Methoden zugreifen darf. Falls wir nun die Felder **`firstName`** und **`surName`** mit **`protected`** deklarieren anstatt mit **`private`**, läuft das Testprogramm ohne Fehlermeldung durch.
Der Code wird nicht noch einmal kopiert, sondern es werden nur die modifizierten Zeilen der Klasse **`Person`** abgedruckt.

```
public class Person {
    protected String surName;
    protected String firstName;
…
}
```

Bei der genaueren Analyse des Beispiels fällt uns noch etwas auf. Im Konstruktor für die Klasse **`Student`** wird derselbe Code wie in der Klasse **`Person`** verwendet, doppelten Code sollte vermieden werden, d.h. wir müssen hier nach einer Lösung suchen.
Java bietet dafür eine Möglichkeit, nämlich mit **`super(…)`** kann auf die Konstruktoren der Superklasse zugegriffen werden. Wird dies nun angewendet, so sehen die Konstruktoren der Klasse **`Student`** eleganter aus. Auch hier werden nur die zwei Konstruktoren dargestellt.

```
public Student(){
        super();
        matriculNumber = 0;
    }
    public Student(String surName, String firstName,
                   int matriculNumber){
        super(surName, firstName);
        this.matriculNumber = matriculNumber;
    }
```

Aufgabe 50

Ändern Sie die zwei Nachträge im Beispiel 40 ab und lassen Sie das Testprogramm laufen.

Aufgabe 51

Erstellen Sie in Java die Klassen für die Arzneimittel aus der Aufgabe 49: Tabletten, Salbe, Augensalbe und Magentabletten. Die Klassen haben jeweils ein Attribut für die Menge der Packung. Bei den Tabletten ist es die Anzahl Tabletten und bei der Salbe sind es die ml (Milliliter). Die Klasse Magentablette enthält ein Feld für die Dosierung, welche die maximale Anzahl Tablette pro Tag angibt.
Die Klasse Augensalbe hat noch zusätzlich die Dosierung pro Anwendung in Anzahl Tropfen.
Erweitern Sie danach die Klassen um eine Methode **`print(..)`**, welche alle Infos der Klasse ausgibt.
z.B.:
Tablette:
«Es enthält 10 Tabletten.»
Magentablette:
«Es enthält 10 Tabletten.»
«Einnahme: Maximal 3 Tabletten pro Tag.»
Salbe:
«Es enthält 200ml Salbe.»
Augensalbe:
«Es enthält 200ml Salbe.»
«Anwendung: Maximal einen Tropfen pro Auge.»

Schreiben Sie keinen doppelten Code auf, Sie können z.B. mit **`super.print(...)`** die Methode **`print()`** der Basisklasse aufrufen.

Lösung:

```
public class Pharmaceutical {
    private String name;
    public Pharmaceutical (String name){
        this.name = name;
    }
    public void print(){
        System.out.println ("Das Arzneimittel heisst: " + name);
    }
}
```

```
public class Tablet extends Pharmaceutical{
    private int count; // Anzahl der Tabletten
    public Tablet (String name, int count){
        super(name);
        this.count = count;
    }
    public void print(){
        super.print();
        System.out.println ("Es enthält "+ count +" Tabletten.");
    }
}
```

```
public class StomachTablet extends Tablet{
    private int dosage;
    public StomachTablet(String name, int count, int dosage){
        super(name, count);
        this.dosage = dosage;
    }
    public void print(){
        super.print();
        System.out.println("Einnahme: Maximal " + dosage +
                           " Tabletten pro Tag." );
    }
}
```

```
public class Ointment extends Pharmaceutical{
    private int quantity; // Menge der Salbe in ml
    public Ointment (String name, int quantity){
        super(name);
        this.quantity = quantity;
    }
    public void print(){
        super.print();
        System.out.println ("Es enthält " + quantity +
                            " ml Salbe.");
    }
    public int getQuantity(){
        return this.quantity;
    }
}
```

```
public class EyeOintment extends Ointment{
    private int dosage;
    public EyeOintment (String name, int quantity, int dosage){
        super(name, quantity);
        this.dosage = dosage;
    }
    public void print(){
        super.print();
```

```
        //System.out.println ("Die Augensalbe enthält " +
                                super.getQuantity() + " ml.");
        System.out.println("Anwendung: Maximal " + dosage +
                           " Tropfen pro Auge.");
    }
}
```

```
public class TestPharmaceutical {
    public static void main(String[] args){
        Pharmaceutical pharma = new Pharmaceutical
("Schmerzmittel");
        Tablet tablet = new Tablet ("Schmerztablette", 12);
        Ointment ointment = new Ointment ("Wundsalbe", 200);
        EyeOintment eyeOintment = new EyeOintment ("Augensalbe",
                                                    50, 1);
        StomachTablet stomachTablet =
                    new StomachTablet("Magentablette", 10, 3);
        pharma.print();
        tablet.print();
        ointment.print();
        eyeOintment.print();
        stomachTablet.print();
    }
}
```

5.2 Überschreiben von Methoden

In der Aufgabe 51 haben wir bereits die Methode **`print(…)`** überschrieben. Das bedeutet, dass eine Methode der Superklasse in der Subklasse neu definiert wird. Danach kann über ein Objekt der Subklasse nicht direkt auf die Methode der Superklasse zugegriffen werden. In der Methode selbst kann mit **`super.methode(…)`** die Methode der Superklasse aufgerufen werden, dies haben wir in der Methode **`print(..)`** in der Aufgabe 51 mit **`super.print()`** bereits angewendet. Im Testprogramm sieht man im Ausschnitt,

```
pharma.print();
tablet.print();
ointment.print();
eyeOintment.print();
stomachTablet.print();
```

dass die Methode **`print(…)`** der entsprechenden Objekten aufgerufen wird.
`ointment.print(…)` ruft die Methode **`print()`** des Objektes **`ointment`** der Klasse **`Ointment`** auf.

Aufgabe 52

Überschreiben Sie in der Klasse **Student** aus Beispiel 40 die Methode **public String getInfos()** der Klasse **Person**, sodass auch die Matrikelnummer ausgegeben wird. Vermeiden Sie doppelten Code.

Lösung:

```
public String getInfos(){
    String res = super.getInfos();
    res = res + " Matrikelnummer: " + matriculNumber;
    return res;
}
```

5.2.1 Klasse Object

Oracle bietet für die Programmiersprache Java eine sehr gute Online-Hilfe an. Auf der Seite https://docs.oracle.com/javase/8/docs/api/
finden Sie alle Klassen, die in der «Java Runtime Environment» (jre) zur Verfügung stehen.
Wir haben die Klasse **String** mehrfach verwendet. Da es für uns nur einen Text war, sind wir nicht weiter darauf eingegangen und werden dies in dieser Arbeit auch nicht tun.
Sucht man in der Hilfe nach der Klasse **String**, so erhält man eine ausführliche Beschreibung aller Methoden. Zusätzlich sehen Sie oben links die Vererbungshierarchie (Abbildung 52) der Klasse **String**. Man sieht, dass die Klasse **String** von einer Klasse **Object** ableitet, somit ist **Object** die Basisklasse von **String**.

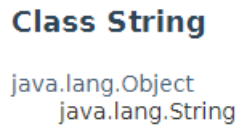

Abbildung 52: Vererbungshierarchie der Klasse String

Aufgabe 53

Wählen Sie vier beliebige Klassen aus der Hilfe aus und prüfen Sie die Vererbungshierarchie. Was fällt Ihnen auf?

Lösung:
Alle Klassen in Java haben im Vererbungsbaum als oberste Klasse die Klasse **Object**.

Sie sehen also, dass **Object** so zu sagen der Urvater aller Klassen ist.
Die Klasse **Object** hat unter anderem eine Methode **toString()**, diese Methode wird von **System.out.println()** verwendet um das Objekt als Text auszugeben. Bei den Standard-Datentypen haben wir bereits gesehen, dass z.B. der Wert einer Variable als Text ausgegeben wird. Analog haben wir in der Klasse **EyeOintment** die Dosierung **dosage** ausgegeben.

```
System.out.println("Anwendung: Maximal " + dosage +
                   " Tropfen pro Auge.");
```

Beispiel 41

Als Beispiel erweitern wir die Klasse **Line** aus Beispiel 26 um diese spezielle Methode **toString()**, mit der die Informationen der Klasse **Line** als Text ausgegeben werden. Da die Klasse **Line** eine Klasse in Java ist, wird sie auch von der Klasse **Object** abgeleitet und enthält durch die Vererbung die Methode **toString()**, welche von **System.out.println()** aufgerufen wird.
Die Signatur der Methode können wir aus der Hilfe ablesen: **public String toString()**, womit wir unsere Erweiterung implementieren können.

```
public String toString(){
    String lineText = "Die Linie beginnt bei (" + start.x +
                      "," + start.y + ") und endet bei ("
                      + end.x + ", " + end.y + ")";

    return lineText;
}
```

Der String kann genau gleich erstellt werden wie der Parameter von **System.out.println(…)** bei einer Ausgabe.
Mit dieser Erweiterung können wir im Testprogramm das Objekt **l2** der Klasse **Line** direkt der **System.out.println(…)** Anweisung übergeben.

```
public static void main(String[] args) {
    Line l2 = new Line(1.2, 3.2, 5.1, 6.3);
    System.out.println(l2);
}
```

Und erhalten die gewünschte Ausgabe.
Die Line beginnt bei (1.2, 3.2) und endet bei (5.1, 6.3)

6 Polymorphismus und dynamische Bindung

Das Thema, welches in diesem Kapitel angesprochen wird, ist sehr umfangreich, wir werden es nur so weit erläutern, damit wir später ein kleines Projekt in OOP aufsetzen können.
Als erstes führen wir das Array ein, bei dem der Polymorphismus eine entscheidende Rolle im Zusammenhang mit der OOP spielt.

6.1 Array

Array hat nichts direkt mit Polymorphie zu tun, aber wir werden einige Beispiele mit Array sehen, bei denen wir Polymorphismus anwenden, deshalb werden hier die Arrays in Java mit Beispielen eingeführt, ohne uns vertiefte Gedanken dazu zu machen. Schlussendlich ist es fast wie eine Liste in Python, nur dass die Grösse nicht verändert werden kann.

Ein Array kann auf zwei verschiedene Arten erstellt werden, entweder man gibt alle Zahlen des Arrays an

```
Int[] arr = { 10, 50, 60, 80, 90 };
```

oder man erstellt zuerst das Array mit

```
int[] arr = new int[10];
```

und füllt es danach über den Index . Die Indizierung beginnt wie bei Python bei 0.
Der **new**-Operator zeigt, dass ein Array in Java als Objekt behandelt wird, so ist die Variable **arr** ein Objekt und zeigt auf einen Speicherbereich mit 10 Integer-Werten.
Wichtig ist und anders als in den Listen von Python, dass alle Elementen des Arrays vom selben Datentyp sein müssen. **int[]** ist eigentlich ein eigener Datentyp und steht für ein Array von Integer, damit ist klar, dass dieser Datentyp beim Erstellen angegeben werden muss.

Um eine Zahl in einem Array zu suchen, müssen alle Elementen geprüft werden. Dies kann am einfachsten mit einer **for**-Schleife programmiert werden.

```
//suchen
int searchIndex = -1;
int searchNumber = 23;
for (int i = 0; i < arr.length; i++){
    if (arr[i] == searchNumber){
      searchIndex = i;
    }
}
```

Diese Variante der **for**-Schleife im Gegensatz zur **for-each** Variante, liefert uns den Index der gesuchten Position.

Nun können wir verstehen, was der Parameter **String[]** der **main**-Methode bedeutet

```
public static void main(String[] args)
```

`String[]` ist ein Array vom Typ `String` und beinhaltet alle Werte, welche beim Programmstart über die Konsole mitgegeben werden.
z.B. `java HelloWorld Byte-Welt Besucher` ruft in einer Konsole die `main`-Methode in der Klasse `HelloWorld` mit den Argumenten **Byte-Welt** und **Besucher** auf. Das Array `String[]` hat dann die zwei Argumente `Byte-Welt` in `args[0]` und `Besucher` in `args[1]`. Dies wird aber nicht weiter diskutiert.

Ein Array von `double` wird mit folgender Zeile erstellt.

```
double[] arr = new double[10];
```

Also können wir von jedem Datentyp ein Array erstellen, somit auch von unserer Klasse `Student` -was ein neuer Datentyp ist- und erhalten so einen neuen Datentyp für ein `Student` Array.

```
Student[] students = new Student[10];
```

Beachten Sie, dass mit der obigen Zeile nur ein Array vom Typ Student für 10 Objekten der Klasse `Student` erzeugt wird, nicht aber den notwendigen Speicherbereich. Die Objekte im Array verweisen alle auf `null`. Um jeweils einen Zustand der Objekte speichern zu können, muss für jedes Objekt im Array eine Instanz der Klasse `Student` mit `new Student()` erstellt werden.

```
students[0] = new Student();
```

6.2 Polymorphismus

Aufgabe 54

Im Beispiel 40 haben wir die Klassen für `Person` und für `Student` erstellt. Überlegen Sie sich wie man eine Datenstruktur wie z.B. ein Array erstellen kann, um alle Personen und Studenten darin zu speichern.
Diskutieren Sie es mit Ihrem Nachbarn.

Lösung:
Mit dem jetzigen Wissen kommt man nicht auf eine Lösung, man könnte für jede Klasse ein Array erstellen, dann hätte man zwei Arrays für die gesamte Schule.

Diese Lösung ist nicht befriedigend, deshalb führen wir ein neues Konzept der OOP ein, mit dem dieses Problem einfach gelöst werden kann. Es geht um Polymorphismus.

Beispiel 42

```
public class A {
    public void print(){
        System.out.println("Ich bin die Klasse A.");
    }
}
```

```
public class B extends A{
    public void print(){
        System.out.println("Ich bin Klasse B und ein Erbe
                            von A.");
    }
}
```

```
public class C extends B{
    public void print(){
        System.out.println("Ich bin Klasse C und ein Erbe
                            von B.");
    }
}
```

```
public class TestABC {
    public static void main(String[] args){
        A a = new A();
        B b = new B();
        C c = new C();

        a.print();
        b.print();
        c.print();
    }
}
```

Das Testprogramm erzeugt folgenden Ausgabe.
Ich bin die Klasse A.
Ich bin Klasse B und ein Erbe von A.
Ich bin Klasse C und ein Erbe von B.

Was bis jetzt nichts neues ist, die Methode **`print()`** wurde in den Klassen **`B`** und **`C`** überschrieben.
Nun ändern wir das Testprogramm ein wenig ab.

```
public class TestABC {
    public static void main(String[] args){
        A a = new A();
        a.print();
        a = new B();
        a.print();
    }
}
```

Aufgabe 55

Lassen Sie dieses Beispiel laufen und achten Sie auf die Ausgabe.

Lösung:
Die Ausgabe ist die folgende:

```
Ich bin die Klasse A.
Ich bin Klasse B und ein Erbe von A.
```

Obwohl das Objekt **a** von Typ der Klasse **A** ist, wird die **print**-Methode der Klasse **B** aufgerufen.

Das, was Sie im Beispiel 49 gesehen haben, ist der sogenannte Polymorphismus.
Dabei gilt folgendes:

1. Polymorphie von Objekten gibt es nur bei Vererbungshierarchien.
2. Ein Objekt einer Basisklasse kann auch auf einen Speicherbereich einer Unterklasse verweisen. Deshalb wird Polymorphie auch Vielgestaltigkeit von Objekten genannt.

Punkt 2 haben wir im obigen Code bereits gezeigt, aber

```
b = new A();
```

ergibt einen Compilerfehler und somit kein lauffähiges Programm, da die Klasse A keine Unterklasse von B ist, sondern die Oberklasse.

Hingegen könnte man dem Objekt **b** der Klasse **B** mit Typecast ein Objekt **a** der Klasse **A** zuweisen, falls **a** auf ein Objekt der Klasse **B** im Speicherbereich zeigt.
Mit Java Code dargestellt:

```
A a = new B();
B b = (B)a; //Typecast von a auf Klasse B
```

Zeigt **a** nicht auf ein Objekt der Klasse **B** im Speicherbereich, funktioniert dies nicht. Die folgenden Anweisungen ergeben einen Fehler zur Laufzeit.

```
A a = new A();
B b = (B)a;
```

Kurz gesagt, mit Polymorphismus wird erst zur Laufzeit des Programmes entschieden, welche Methode konkret aufgerufen wird, nämlich diejenige, auf die das Objekt zeigt, unabhängig, ob es als ein Objekt einer Oberklasse oder einer Unterklasse deklariert worden ist.

Aufgabe 56

Erstellen Sie das UML-Klassendiagramm des Beispiels 42.

Lösung:

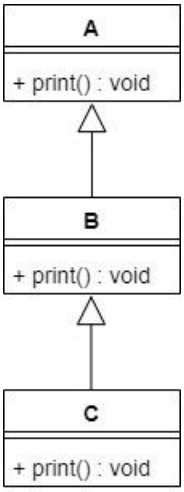

Abbildung 53: Vererbungshierarchie der Klasse A, B und C

Aufgabe 57

Suchen Sie nun eine bessere Lösung für das Problem der Aufgabe 54. Wir suchen eine Möglichkeit, um alle Personen und Studenten der Schule in einer Datenstruktur zu verwalten.

Lösung:
Wir erstellen ein Array mit Personen:
`Person[] persons = new Person[number]`
`Person[]` ist ein neuer Datentyp, nämlich ein Array der Klasse **`Person`** und **`number`** ist die Anzahl Personen der Schule plus eine gewisse Reserve.
Danach können wir den Objekten **`persons[i]`** jeweils ein Objekt der Klasse **`Person`** oder der Klasse **`Student`** zuweisen.

Beispiel 43

Um Polymorphismus besser zu verdeutlichen, werden wir nun ein weiteres Beispiel zeigen.
Wir haben wieder die Schule mit Personen und Studenten. Die Objekte all dieser Klassen können wir mit einem Array **`Person[]`** verwalten. Dies haben wir in der letzten Aufgabe gefunden. Es wird für die Schulverwaltung eine Klasse **`School`** erstellt, welches dieses Array enthält.

```
public class School {
    private Person[] employees;
    private int insertCnt;

    School(int anz){
        employees = new Person[anz];
    }
```

```
    public void insertPerson(Person person){
        employees[insertCnt] = person;
        insertCnt++;
    }
}
```

Falls wir die Personen nach Namen im Array **Person[]** sortieren möchten, dann genügt nun eine Methode in der Klasse **School** zu erstellen.
Im Folgenden werden die Methoden **public void sortEmployees()** und **private void swap(int index1, int index2)** der Klasse **School**, welche für das Sortieren (Bubblesort) des Arrays benötigt werden, aufgelistet.

```
public void sortEmployees(){
    int topLimit = insertCnt;
    while (topLimit > 1){
        for (int i = 1; i < topLimit; i++){
            String a = employees [i].getSurName();
            String b = employees [i - 1].getSurName();
            if(a.compareTo(b) < 0){
                swap (employees, i, i - 1);
            }
        }
        topLimit --;
    }
}

private void swap (int index1, int index2){
    Person tmp = employees [index1];
    employees [index1] = employees [index2];
    employees [index2] = tmp;
}
```

Man sieht weiter in diesem Beispiel eine sinnvolle Anwendung vom **private** Modifikator bei einer Methode. Die Methode **swap(…)** macht nur Sinn innerhalb der Klasse **School**, deshalb ist sie **private**.

Nun möchten wir die Informationen aller Mitarbeiter ausgeben, dafür genügt es die Methode **getInfos()** aller **gültigen** Objekte (nur solche, welche auf einen Speicherbereich zeigen) im Array **Person[]** aufzurufen. Die Methode **getInfos()** wurde in der Basisklasse implementiert und in den abgeleiteten Klassen überschrieben. Da Java die Auswahl der entsprechenden Methode erst zur Laufzeit vornimmt, wird dank dem Polymorphismus, die Methode **getInfos()** entsprechende dem Objekt im Array aufgerufen. Nun erweitern wir die Klasse **School** um die Methode **printEmpoyees()**, welche für alle Objekte im Array **employees** die Methode **getInfos()** aufruft.

```
public void printEmployees(){
    for (int i = 0; i < employees.length; i++){
        if (employees[i] != null){
            System.out.println(employees[i].getInfos());
        }
    }
}
```

Das zugehörende Testprogramm zeigt die Anwendung der Klasse **School**.

```
public class TestSchool {
    public static void main(String[] args){
        School ETH = new School(100);
        Person p = new Person("Müller", "Hans");
        ETH.insertPerson(p);
        p = new Student("Weber", "Yves", 121111);
        ETH.insertPerson(p);
        p = new Student("Brunner", "Daniel", 122222);
        ETH.insertPerson(p);
        p = new Student("Werder", "Celine", 123333);
        ETH.insertPerson(p);
        p = new Student("Dörig", "Michaela", 124444);
        ETH.insertPerson(p);
        p = new Student("Senn", "Lena", 125555);
        ETH.insertPerson(p);

        ETH.printEmployees();
        ETH.sortEmployees();
        ETH.printEmployees();
    }
}
```

6.3 Schulverwaltung mit Listen

Wir haben im Kapitel 4 «Einfach verkettete Listen» eine weitere Möglichkeit als die Arrays gezeigt, um Daten zu verwalten. Die Daten in den Beispielen waren Zahlen, wir können auch Objekte als Daten verwenden und erreichen dadurch noch mehr Flexibilität. Der Vorteile einer Liste gegenüber einem Array ist, dass wir die Grösse nicht zu Beginn festlegen müssen, wir können einfach einen weiteren Knoten hinzufügen.
Falls wir ein Element aus der Liste löschen möchten, gibt es keine «Löcher». In einem Array wäre der Platz an einem bestimmten Index leer, ausser man verschiebt danach alle folgenden Daten um eine Position zurück, was aber einen zusätzlichen Aufwand erfordert.
Weiter falls wir zu Beginn eine zu kleine Grösse für das Array gewählt haben, so müssen wir ein grösseres Array erzeugen und alle Daten kopieren, was wieder mit einem Aufwand verbunden ist. Deshalb ist die Liste als Datenstruktur für die Datenverwaltung geeigneter.

Beispiel 44

Wir werden nun die Aufgabe aus dem Beispiel 43 mit Listen lösen. Die Klasse **`SchoolList`** enthält eine Liste, die wir im Kapitel 4 «Einfach verkettete Listen» hergeleitet haben, anstatt eines Arrays.
Die Liste aus dem vorherigen Kapitel 4 «Einfach verkettete Listen» hat ein Objekt **`head`** der Klasse **`Node`**, damit wir nicht alles neu kopieren müssen, werden wir einen neue Klasse **`NodePerson`** erstellen, welche von der Klasse **`Node`** ableitet. Die neue Klasse **`NodePerson`** wird um ein Objekt **`person`** der Klasse **`Person`** erweitert. Alles andere bleibt genau gleich. Das Attribut **`data`** der Klasse **`Node`** benötigen wir für diese Aufgabe nicht. Mit diesem Ansatz wird es aber mitvererbt, jedoch lassen wir diese kleine Unschönheit bestehen.

Bemerkung:
Eine saubere Lösung wäre eine Basisklasse **`BaseNode`** zu erstellen ohne Daten und diese Klasse als Interface oder abstrakter Klasse zu definieren. Beide Themen werden wir nicht behandeln, da es sonst den Rahmen sprengen würde. Deshalb bleiben wir bei dieser kleinen Unschönheit.

Nun können wir die Klasse **`NodePerson`** erstellen:

```
public class NodePerson extends Node{
    private Person person;

    NodePerson(Person person){
        super();
        this.person = person;
    }

    public void setPerson(Person person){
        this.person = person;
    }

    public Person getPerson(){
        return this.person;
    }
}
```

Ein Objekt der Klasse **NodePerson**, welche von der Klasse **Node** ableitet, kann dem Attribut **head** der Klasse **List** zugewiesen werden (Polymorphismus). Die Methoden der Klasse **List** können für Objekte der Klasse **NodePerson** genau gleich benutzt werden. In der Klasse **SchoolList** können wir nun anstatt des Arrays unsere Liste verwenden, ohne eine Zeile Code in der Klasse **List** zu verändern.

```
public class SchoolList {
    private List employees;

    SchoolList(){
        employees = new List();
    }

    public void insertPerson(NodePerson person){
        employees.insertEnd(person);
    }

    public void printEmployees(){
        Node startPos = employees.getHead();
        while(startPos != null){
            String str =
                    ((NodePerson)startPos).getPerson().getInfos();
            System.out.println(str);
            startPos = startPos.getNext();
        }
    }
}
```

Im obigen Code werden nicht alle Methoden für das Einfügen und Löschen einer Person geschrieben, es soll nur anhand der Methoden **public void insertPerson(NodePerson person)** gezeigt werden, wie eine Person in die Liste aufgenommen werden kann.
Am Aufruf **employees.insertEnd(person)** sieht man, dass der Methode **insertEnd(…)** auch ein Objekt der Klasse **NodePerson** übergeben werden kann (auch wieder Polymorphismus). Trotzdem haben wir ein kleines Problem, falls wir auf die Daten, nämlich auf das Objekt **person** der Klasse **NodePerson** zugreifen möchten. Die Knoten in der Liste sind Objekte der Klasse **Node** und nicht der Klasse **NodePerson**, deshalb können wir nicht direkt auf den Zustand des Objektes **person** zugreifen. Wir wissen aber, dass alle Objekten unserer Liste auf einen Speicherbereich von Instanzen der Klasse **NodePerson** verweisen, deshalb können wir einen sog. **Typecast** (Typ Umwandlung) anwenden, d.h. wir müssen Java mitteilen, dass es sich im Speicher um ein Objekt der Klasse **NodePerson** handelt.
Diesen Typecast funktioniert nur, da wir sicher sind, dass die Knoten in der Liste alles Objekte der Klasse **NodePerson** sind.

Zur Erinnerung: Weiter oben haben wir folgendes bereits gesehen.
Die Klasse **B** leitet von der Klasse **A** ab, dann funktioniert der Typecast:

```
A a = new B();
B b = (B)a;
```

Der Typecast und den Zugriff auf das Objekt **person** kann mit folgenden Zeilen erreicht werden:

1. Umwandlung von **startPos** in ein Objekt der Klasse **NodePerson**

```
NodePerson tmpNode = (NodePerson)startPos;
```

   (ergibt keinen Fehler, da wir sicher sind, dass das Objekt der Klasse **NodePerson** ist)

2. Mit **tmpNode** kann nun auf das Attribut **person** zugegriffen werden.

```
Person pers = tmpNode.getPerson();
```

3. **getInfos()** können wir nun aufrufen.

```
String str = pers.getInfos();
```

Für Erfahrene die Kurzversion:

```
String str = ((NodePerson)startPos).getPerson().getInfos();
```

Dieses Beispiel sollte zeigen, welche Vorteile OOP hat, man kann vorhandenen Code wieder verwenden, indem man von der entsprechenden Klasse ableitet.
Zuletzt wird auch das Testprogramm für diese Lösung aufgelistet.

```
public class TestSchoolList {
    public static void main(String[] args){
        SchoolList ETH = new SchoolList();
        Person p = new Person("Müller", "Hans");
        NodePerson np = new NodePerson(p);
        ETH.insertPerson(np);
        p = new Student("Weber", "Yves", 121111);
        np = new NodePerson(p);
        ETH.insertPerson(np);
        p = new Student("Brunner", "Daniel", 122222);
        np = new NodePerson(p);
        ETH.insertPerson(np);
        p = new Student("Werder", "Celine", 123333);
        np = new NodePerson(p);
        ETH.insertPerson(np);
        p = new Student("Dörig", "Michaela", 124444);
        np = new NodePerson(p);
        ETH.insertPerson(np);
        p = new Student("Senn", "Lena", 125555);
        np = new NodePerson(p);
        ETH.insertPerson(np);

        ETH.printEmployees();
    }
}
```

6.4 instanceof

Wir haben im letzten Kapitel einen Typecast durchgeführt, damit wir über ein Objekt der Klasse **`Node`** auf die Daten des Objektes der Klasse **`NodePerson`** zugreifen können. Dies ist erlaubt, sofern im Speicher das Abbild des gewünschten Objektes wirklich vorliegt. Wie können wir nun sicher sein, dass es sich um den «richtigen» Speicherbereich handelt? Diese Thematik möchte wir in diesem Unterkapitel besprechen.
Dazu erweitern wir das UML-Klassendiagramm der Aufgabe 55 um eine Klasse D, welche direkt von der Klasse A ableitet, dies ist in der Abbildung 54 dargestellt.

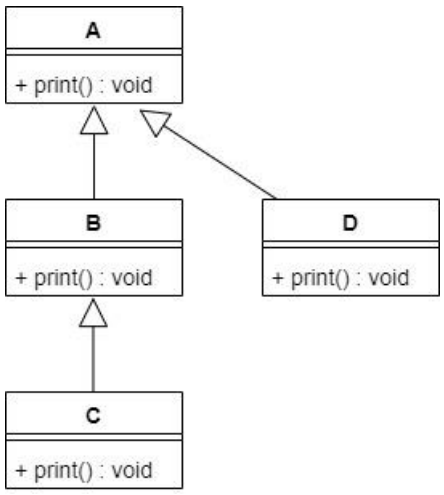

Abbildung 54: UML-Klassendiagramm der Klassen A, B, C und D

Nun erstellen wir ein Array für vier Elementen der Klasse A.

```
A[] myArray = new A[4];
```

Füllen danach das Array mit verschiedenen Objekten der Klassen A, B, C und D auf.

```
A a = new A();
B b = new B();
C c = new C();
D d = new D();

myArray[0] = a;
myArray[1] = b;
myArray[2] = c;
myArray[3] = d;
```

Weiter rufen wir die **`print`**-Methode über die Referenzen im Array auf. Wir wissen, dass aufgrund des Polymorphismus dies kein Problem darstellt und es wird jeweils auch die richtige **`print`** -Methode verwendet.

```
for (int i = 0; i < myArray.length; i++){
    myArray[i].print();
}
```

Bis hier ist es eine Repetition der oben erwähnten Theorie.
Nun ergänzen wir die Klasse D mit einer zusätzlichen Methode **printD**(), welche wir in der **for**-Schleife aufrufen möchten. Falls wir die **for**-Schleife folgendermassen abändern,

```
for (int i = 0; i < myArray.length; i++){
    myArray[i].print();
    D d2 = (D)myArray[i];
    d2.printD();
}
```

erhalten wir einen Fehler.
Exception in thread "main" java.lang.ClassCastException: class A cannot be cast to class D
Es ist so weit auch nachvollziehbar, dass dies nicht geht, da die Methode **printD()** nicht in allen Klassen definiert ist und somit nicht in allen Objekten vorhanden ist.
Wir möchten aber trotzdem **printD()** für ein Objekt der Klasse **D** aufrufen, diese Unterscheidung, ob ein Objekt der Klasse D ist oder nicht, kann in Java mit dem Operator **instanceof** gelöst werden.

Mit dem **instanceof**-Operator wird getestet, ob ein Objekt auf einen Speicherbereich einer bestimmten Klasse zeigt. Dies ist beispielsweise dann wichtig, wenn ein Objekt vom Typ einer Basisklasse auf einen Speicherbereich eines Objektes der abgeleiteten Klasse zeigt und wir wissen, dass spezielle Attribute und Methoden nur in dieser bestimmten abgeleiteten Klasse enthalten sind, also wie **printD()** in der obigen Beschreibung.
Mit **instanceof** können wir nun den Code so abändern, dass wir unsere gewünschte Methode bei einem Objekt der Klasse **D** aufrufen können.

```
for (int i = 0; i < myArray.length; i++){
    myArray[i].print();
    if (myArray[i] instanceof D){
        D d2 = (D)myArray[i];
        d2.printD();
        //oder:
        //((D)myArray[i]).printD();
    }
}
```

Entscheidend ist die Zeile **if (myArray[i] instanceof D)**, in der getestet wird, ob es sich wirklich um ein Objekt der Klasse **D** handelt.

Bemerkung:
instanceof liefert **true**, falls ein Objekt vom Typ der angegebenen Klasse ist. Es wird auch **true** geliefert, falls die angegeben Klasse eine Basisklasse ist, deshalb muss man sich gut überlegen auf welche Klasse getestet werden soll, am besten frägt man direkt die Klasse ab, in die man umwandeln möchten.
Dieser Code liefert dreimal «True», da die Klasse **C** von der Klasse **B** abgeleitet wird und die Klasse **B** von der Klasse **A** ableitet (Abbildung 54).

```
C c = new C();
if (c instanceof A){
    System.out.println("True");
}
if (c instanceof B){
    System.out.println("True");
}
if (c instanceof C){
    System.out.println("True");
}
```

Aufgabe 58

Dies ist eine grössere Aufgabe.
Sie haben den Auftrag für eine Bibliothek eine Bücherverwaltung zu entwickeln.
Die Bibliothek hat Zeitschriften, Romane, Sachbücher, und Comic-Hefte. Alle Bücher besitzen eine «International Standard Book Number» ISBN (zur Vereinfachung genügt hier eine Zahl, welche fiktiv für eine ISBN steht), einen Titel, einen Autor und eine Bibliotheksbezeichnung als Text, in der klar ersichtlich ist, in welchem Gang das Buch eingeordnet werden soll.
Die Zeitschriften sind im Gang A, die Comics im Gang B, die Roman im Gang C und die Sachbücher im Gang D.
Von jedem Buch können die entsprechenden Informationen ausgegeben werden.

Die Zeitschriften (Magazine) haben noch zusätzlich den Typ, wie z.B. Geo, Chip, Spick usw. und das Erscheinungsdatum.
Die Comics besitzen den Namen der Hauptdarstellen, z.B. Asterix, Clever & Smart, usw.
Bei einem Roman (Novel) wir die Gattung angegeben, z.B. Liebesroman, Kriegsroman, usw. sowie das Band, also Band 1, Band 2.
Bei den Sachbüchern (Spezialized) ist der wissenschaftliche Inhalt, z.B. Erdwissenschaften, Informatik, Mathematik, usw. gespeichert.

a) Erstellen Sie ein UML-Klassendiagramm der Vererbungshierarchie der Bücher. Wählen Sie dabei die Attribute als **`protected`** (#). Die Methoden können Sie vorläufig weglassen.
b) Implementieren Sie alle notwendigen Klassen.
c) Überlegen Sie sich, was Sie tun müssen, um unsere einfach verkettete Liste für die Bücher anzuwenden. Führen Sie danach diese Überlegung aus.
d) Wie könnte man ein Buch als «ausgeliehen» und «verfügbar» markieren? Fügen Sie diese Änderung ein.
e) Erweitern Sie die Klassen um eine Methode **`printInfos()`**, welche alle Infos der Klasse ausgibt.
f) Schreiben Sie in der Klasse **`Book`** eine Methode **`borrow()`**, mit der ein Buch ausgeliehen werden kann. Falls dies der Fall ist, wird das Buch als «ausgeliehen» markiert und es wird **`true`** zurückgegeben, sonst **`false`**. Der Zustand (state) des Buches («ausgeliehen» oder «verfügbar») ist ein **`String`**, welcher mit **`state.compareTo(«verfügbar») == 0`** auf Gleichheit geprüft werden kann.
g) Es fehlt noch eine weitere Methode für das Zurückbringen (**`bringBack()`**) des Buches. Mit **`bringBack()`** wird der Zustand des Buches auf «verfügbar» gesetzt.
h) Nun können Sie die Klasse **`Library`** erstellen, um die Bücher zu verwalten. In der Klasse **`Library`** befindet sich eine Liste mit allen Büchern.

i) Schreiben Sie ein Testprogramm, um die Funktionalität Ihrer Bibliothek zu prüfen.

Lösung:

a) UML-Klassendiagramm

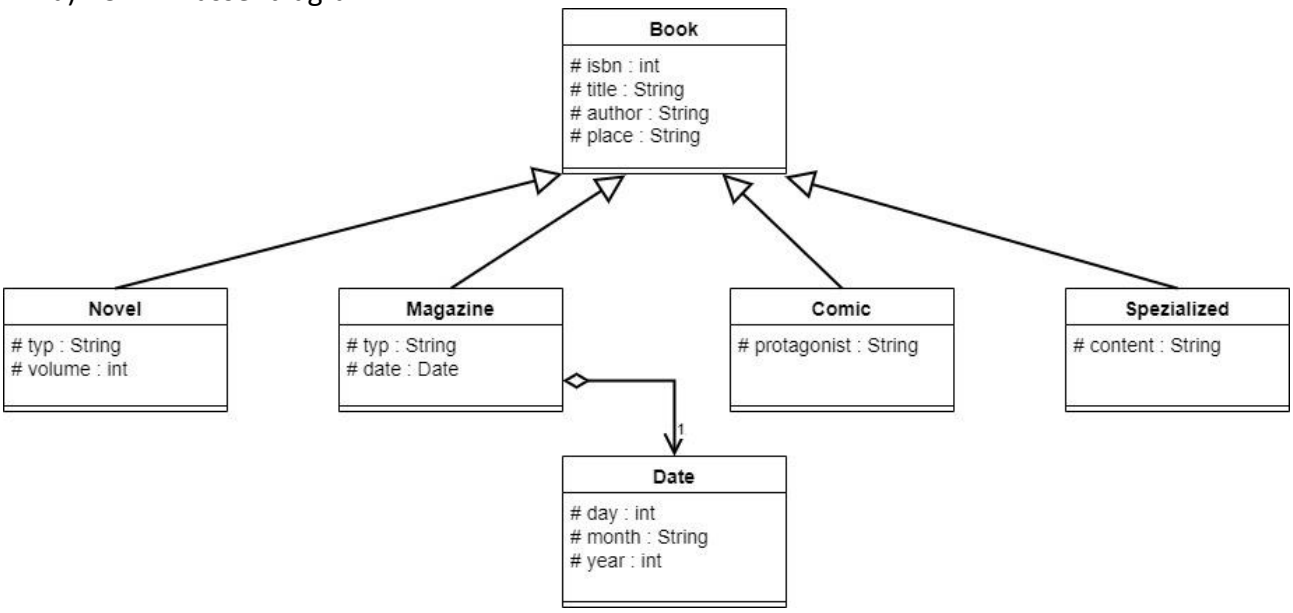

Abbildung 55: UML-Klassendiagramm der Bücher

b) Klassen

```
public class Book {
    protected int isbn;
    protected String title;
    protected String author;
    protected String place;

    public Book(int isbn, String title, String author,
                String place){
        this.isbn = isbn;
        this.title = title;
        this.author = author;
        this.place = place;
    }
}
```

```
public class Novel extends Book{
    protected String typ;
    protected int volume;

    public Novel(int isbn, String title, String author,
                 String typ, int volume){
        super(isbn, title, author, "Gang C");
        this.typ = typ;
        this.volume = volume;
    }
}
```

```
public class Magazine extends Book{
    protected Date date;
    protected String typ;
    public Magazine(int isbn, String title, String author,
                    Date date, String typ){
        super(isbn, title, author, "Gang A");
        this.date = date;
        this.typ = typ;
    }
}
```

```
public class Comic extends Book{
    protected String protagonist;
    public Comic(int isbn, String title, String author,
            String protagonist){
        super(isbn, title, author, "Gang B");
        this.protagonist = protagonist;
    }
}
```

```
public class Spezialized extends Book{
    protected String content;
    public Spezialized(int isbn, String title, String author,
                String content){
        super(isbn, title, author, "Gang D");
        this.content = content;
    }
}
```

c) Node für einfach verkettete Liste
Um die Liste zu verwenden, benötigen wir einen Knoten der Klasse **Node**, da aber dieser Knoten zu wenig Informationen hat, müssen wir eine neue **Subklasse NodeBook** erstellen, welche ein Objekt **book** als Attribut enthält.

```
public class NodeBook extends Node{
    private Book book;

    NodeBook(Book book){
        super();
        this.book = book;
    }

    public void setBook(Book book){
        this.book = book;
    }

    public Book getBook(){
        return this.book;
    }
}
```

d) Die Klasse **Book** muss um ein einziges Attribut **state** erweitert werden.

```
public class Book {
    protected int isbn;
    protected String title;
    protected String author;
    protected String place;
    private String state;

    public Book(int isbn, String title, String author,
                String place){
        this.isbn = isbn;
        this.title = title;
        this.author = author;
        this.place = place;
        this.state = "verfügbar";
    }

    public void set(String state){
        this.state = state;
    }

    public String getState(){
        return state;
    }
}
```

e) Methode **printInfos()**

Book

```
public void printInfos(){
    System.out.println("Titel: " + title);
    System.out.println("Autor: " + author);
    System.out.println("ISBN: " + isbn);
    System.out.println("Aufbewahrungsort: " + place);
    System.out.println("Das Buch ist " + state + ".");
}
```

Novel

```
public void printInfos(){
    System.out.println(typ);
    System.out.println("Band " + volume);
    super.printInfos();
}
```

Magazine

```
public void printInfos(){
    System.out.println(typ);
    super.printInfos();
    System.out.print("Erscheinungs-");
    date.print(true);
}
```

Comic

```
public void printInfos(){
    System.out.println(protagonist);
    super.printInfos();
}
```

Spezialized

```
public void printInfos(){
    System.out.println(content);
    super.printInfos();
}
```

f) ausleihen

```
public boolean borrow(){
    if (state.compareTo("verfügbar") == 0){
        state = "ausgeliehen";
        return true;
    }else{
        return false;
    }
}
```

g) zurückbringen

```
public void bringBack(){
    state = "verfügbar";
}
```

h) Bibliothek

```
public class Library {
    private List books;
    public Library(){
        books = new List();
    }
    public void insertBook(NodeBook nodeBook){
        books.insertEnd(nodeBook);
    }
    public void printBooks(){
        Node startPos = books.getHead();
        while(startPos != null){
            ((NodeBook)startPos).getBook().printInfos();
            startPos = startPos.getNext();
        }
    }
}
```

i) Testprogramm (in der Klasse **`Library`** erstellt)

```
public static void main(String[] args){
    Library lib = new Library();
    Novel nov = new Novel(123456, "Kind of Wrath",
                          "Ana Huang",
                  "Liebesroman", 0);
    NodeBook nb = new NodeBook(nov);
    lib.insertBook(nb);

    nov = new Novel(87654, "Der Adler ist entkommen",
                  "Jack Higgins",
              "Kriegsroman", 0);
    nb = new NodeBook(nov);
    lib.insertBook(nb);

    Date today = new Date(10, "Januar", 2024);
    Magazine mag = new Magazine(232323, "Mamma Mia!",
                        "verschiedene Autoren", today, "GEO");
    nb = new NodeBook(mag);
    lib.insertBook(nb);

    Comic com = new Comic(111111, "Asterix bei den Römer",
              "R. Goscinny", "Asterix & Obelix");
    nb = new NodeBook(com);
    lib.insertBook(nb);

    Spezialized spez = new Spezialized(444444,
              "Informatik für Dummies", "Haffner",
              "Informatik");
    nb = new NodeBook(spez);
    lib.insertBook(nb);

    lib.printBooks();
    com.printInfos();
    com.borrow();
    com.printInfos();
    com.bringBack();
    com.printInfos();
}
```

7 Zusammenfassung

In dieser Arbeit wurde eine Einführung in OOP erarbeitet, mit dem Ziel, dass das Konzept verstanden wird. Die grundlegende Idee der OOP ist, dass man versucht die Realität im Softwaremodell abzubilden. Möchte man eine Steuerung für ein Auto programmieren, so bedeutet dies in OOP, dass die zuständigen Module des realen Autos auch im Softwareentwurf sichtbar sind. Das heisst, dass wir im Softwaremodell eine Klasse Benzinpumpe, eine Klasse Antrieb, eine Klasse Lenkrad, usw. implementieren werden, sowie auch die Klassen für alle Sensoren oder Sensorgruppen. Diese Klassen bestehen aus Attributen und Methoden mit denen jedes Objekt, welches aus der entsprechenden Klasse instanziiert wurde, selbst weiss, welche Operationen ausgeführt werden können, z.B. falls man dem Objekt der Klasse Antrieb den Befehl (die Methode) um anzufahren sendet, so weiss das Objekt was es zu tun hat. Es wird alle Objekte ansprechen, welche wiederum wissen was sie einstellen und delegieren müssen, damit das Auto nicht ruckartig losfährt. Das könnte z.B. sein, dass das Objekt der Klasse Antrieb mit dem Objekt der Klasse Getriebe, mit dem der Benzinpumpe, usw. kommuniziert.
Dies finden wir auch bei den Operationen auf einen Standarddatentyp. So weiss eine Integer-Variable was mit der Operation «+» tun soll.

Eine Klasse ist eine Vorlage, die beschreibt, wie ein Objekt erstellt werden soll. Darin wird definiert, welche Zustände und welche Operationen das Objekt enthalten wird. Ein Objekt hingegen ist eine Instanz einer Klasse, d.h. es verweist auf einen Speicherbereich, in welchem der Zustand des Objektes gespeichert ist. Ein Synonym dafür ist auch Referenz, also eine Referenz ist dasselbe wie ein Objekt und zeigt auch auf einen Speicherbereich.

Damit lassen sich neue Datenstrukturen herleiten, wir haben die einfach verkette Liste mit diesem Konzept implementiert, jeder Knoten, was einem Objekt entspricht, kennt seinen Nachfolger und die Liste kennt den ersten Knoten. Wie man einen neuen Knoten einfügt, weiss das Objekt der Klasse Liste selbst und wie man auf die Daten des Knotens zugreift, ist im Objekt der Klasse Node festgelegt. Mit Listen lassen sich viele Verwaltungen abbilden, wir haben als Beispiel eine Schule und eine Bibliothek modelliert, die Schule enthält eine Liste und die Knoten enthalten die Daten von Mitarbeiter der Schule als Objekt.
Damit wir eine reale Abbildung der Schule haben, wurden Studenten eingefügt. Da aber alle Mitarbeiter Personen sind und eine gewisse Basisfunktionalität haben, konnten wir mit dem Konzept der Vererbung die Grundfunktionalität der Klasse Person auf die Klasse Student vererben. In den spezialisierten Klassen mussten wir nur noch die zusätzliche Funktionalität integrieren, dies ist ein weiterer Vorteil der OOP. Die Verwaltung der Mitarbeiter wurde als eine Liste von Personen erstellt, da, wie bereits erwähnt, die restlichen Mitarbeiter auch Personen sind, konnten wir Objekten der Klasse Student in die Liste einfügen. Dies nennt man Polymorphismus oder Vielseitigkeit der Objekte. Wieder ein schönes Konzept, um verschiedene Daten zu verwalten.
Mit Überschreiben war es uns möglich bestimmte Methoden der Basisklasse zu erweitern, dies bewirkt wegen dem Polymorphismus, dass obwohl Objekte der Klasse Person sich in der Liste befinden, trotzdem die Methode des Objektes der Klasse Student aufgerufen werden konnte. Da das Objekt auf den Speicherbereich einer Instanz der Klasse Student zeigt, wurden auch die Daten und Methoden dieses Objektes verwendet.

Ein weiteres Konzept, das Überladen von Methoden, hat uns ermöglicht den Namen einer Methode zu behalten, dies ist wichtig, da der Name meistens etwas über die Funktionalität aussagt. Damit kann man z.B. eine Methode `summe(…)` für die Berechnung der Summe von ganzen Zahlen erstellen und mit demselben Namen eine Methode `summe(…)` für Kommazahlen schreiben. Dieses Konzept ist vor allem bei den Konstruktoren wichtig, welche vom System bei der Instanziierung eines Objektes automatisch aufgerufen werden, um die Daten im Objekt zu initialisieren.

Wir haben dafür Java verwendet, da Java die Konzepte der OOP, von mir aus gesehen, am besten umgesetzt hat, ohne dass man sich dabei um Speichermanagement zu kümmern hat. Das Modell für die Softwareentwicklung kann mit UML dargestellt werden, was den Entwurf unabhängig von einer Programmiersprache macht. UML bedeutet «**U**nified **M**odeling **L**anguage», also eine Sprache, um die Modellierung des Softwareentwurfs zu beschreiben. UML ist eine mächtige Sprache, aus der wir nur die Klassendiagramme gezeigt haben, sonst hätte es den Umfang dieser Arbeit gesprengt.

8 Kontrollfragen

Kontrollfrage 1

Was ist die Idee von OOP?

Lösung:
OOP versucht reale Objekte in der Software abzubilden, dies unterstützt auch den Modularen Aufbau eines Programmes.

Kontrollfrage 2

Was ist der Unterschied zwischen Klassen und Objekten?

Lösung:
Ein Klasse ist eine Vorlage für die Instanziierung eines Objektes. Die Klasse beschreibt die Funktionalität des Objektes und welche Zustände im Speicher gehalten werden sollen. Ein Objekt zeigt auf einen Speicherbereich, in dem der Zustand des Objektes gespeichert wird.

Kontrollfrage 3

Was ist der Unterschied zwischen einem Objekt und einer Referenz?

Lösung:
Referenz ist ein Alias für Objekt. Es ist dasselbe, beide Zeigen auf den Speicherbereich, in dem der Zustand des Objektes abgespeichert ist.

Kontrollfrage 4

Erklären Sie die Begriffe Überschreiben und Überladen.

Lösung:
Überschreiben wird im Zusammenhang mit Vererbung erwähnt. Es definiert eine neue Funktionalität einer bestehenden Methode der Basisklasse in der abgeleiteten Klasse. Überladen ist eine neue Methode, welche denselben Namen hat wie eine bestehende. Die Parameter müssen sich entweder hinsichtlich der Anzahl oder des Datentyps unterscheiden.

Kontrollfrage 5

Was ist Polymorphismus?

Lösung:
Polymorphismus ist die Vielseitigkeit eines Objektes. Ein Objekt der Basisklasse kann auch auf eine Instanz einer abgeleiteten Klasse zeigen. Dies ermöglicht z.B. eine Liste mit Objekten der Basisklasse zu erstellen und diese Objekte können auf einen Speicherbereich der abgeleitete Klasse zeigen.

Kontrollfrage 6

Stellen Sie graphisch die Darstellung eines Objektes p der Klasse **`Point`** und dem Speicherbereich dar.

Lösung:

Aus der Klasse **`Point`** wird ein Objekt **`p`** instanziiert, das Objekt zeigt nun auf den Speicherbereich mit dem Zustand des Objektes.

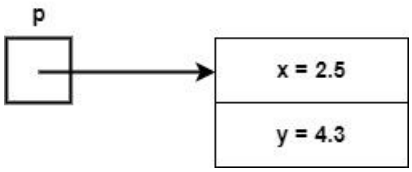

Abbildung 56: Objekt zeigt auf einen Speicherbereich

Kontrollfrage 7

Überlegen Sie sich, was im folgenden Beispiel fehlschlägt, beschreiben Sie das Problem und korrigieren Sie es.

funktionierender Code:

```
int[] myArray = new int[5];
System.out.println(myArray[0]);
```

fehlerhafter Code:

```
Point[] pointArray = new Point[5];
pointArray[0].print();
```

Lösung

Beim fehlerhaften Code wird eine Methode **`print()`** aufgerufen, ohne dass das Objekt im Array auf einen Speicherbereich zeigt. D.h. es fehlt die Instanziierung der Klasse Point für das Objekt im Array.

Korrektur:

```
Point[] pointArray = new Point[5];
pointArray[0] = new Point();
pointArray[0].print();
```

Kontrollfrage 8

Die Klasse **KA** ist eine Basisklasse der Klasse **KB**.

```
public class KA {
    private double a;
    private double b;

    public KA(double a, double b){
        this.a = a;
        this.b = b;
    }

    public double pythagoras(){
        return Math.sqrt(a*a + b*b);
    }
}
```

```
public class KB extends KA{
    private double c;

    public KB(double a, double b, double c){
        super(a, b);
        this.c = c;
    }
    public double pythagoras(){
        return Math.sqrt(a*a + b*b + c*c);
    }

    public static void main(String[] args){
        KB b = new KB(2.1, 3.2, 4.5);
        System.out.println(b.pythagoras());
    }
}
```

Bei der Ausführung des Hauptprogrammes, erscheinen folgende Fehler:

The field KA.a is not visible.
The field KA.a is not visible.
The field KA.b is not visible.
The field KA.b is not visible.

Beschreiben Sie das Problem und korrigieren Sie den Fehler.

Lösung:
In der Klasse **KA** sind die Attribute **a** und **b** als **private** deklariert, d.h. sie sind nur in dieser Klasse sichtbar. Will man in der Klasse **KB** auf diese Attribute zugreifen, ist dies nicht erlaubt. Dieses Problem lässt sich mit dem Schlüsselwort **protected** lösen, dann sind die Attribute **a** und **b** immer noch gegenüber Klassen, welche nichts mit **KA** zu tun haben, geschützt, aber eine abgeleitete Klasse **KB** darf darauf zugreifen.

```
public class KA {
    protected double a;
    protected double b;

    public KA(double a, double b){
        this.a = a;
        this.b = b;
    }

    public double pythagoras(){
        return Math.sqrt(a*a + b*b);
    }
}
```

Kontrollfrage 9

Was wird im folgenden Code ausgegeben?

```
class K9A {
    public void print(){
        System.out.println("Klasse A.");
    }
}
```

```
class K9B extends K9A{
    public void print(){
        System.out.println("Klasse B.");
    }
}
```

```
public class Kontrollfrage9 {
    public static void main(String[] args){
        K9A a = new K9B();
        a.print();
    }
}
```

Lösung:
Klasse B.

Kontrollfrage 10

Im nächsten Code wird die Klasse **`Point`** aus Beispiel 24 verwendet.
Überlegen Sie sich, was bei der Ausführung des untenstehenden Code in der Konsole geschrieben wird, und begründen Sie die Ausgabe.

```
public class Kontrollfrage10 {
    public static void main(String[] args){
        Point[] myArray = new Point[2];
        Point p = new Point(2.5, 1.5);
        myArray[0] = p;
        myArray[1] = p;
        myArray[1].x = 0.5;
        myArray[1].y = 0;

        p.print();
    }
}
```

Lösung:
Koordinaten: x = 0.5 und y = 0.0.
Es wird ein Objekt **`p`** der Klasse **`Point`** erzeugt, dieses Objekt zeigt nun auf einen Speicherbereich mit dem Zustand x = 2.5 und y = 1.5.
Danach wird dem Array **`myArray`** das Objekt **`p`** an den Index 0 und 1 zugewiesen.
Über das Objekt am Index 1 wird nun der Zustand verändert. Da **`p`**, **`myArray[0]`** und **`myArray[1]`** auf denselben Speicherbereich zeigen, erhalten wir bei der Ausgabe mit **`p.print()`** die neuen Werten, da wie bereits erwähnt alle Objekte auf denselben Speicherbereich zeigen.

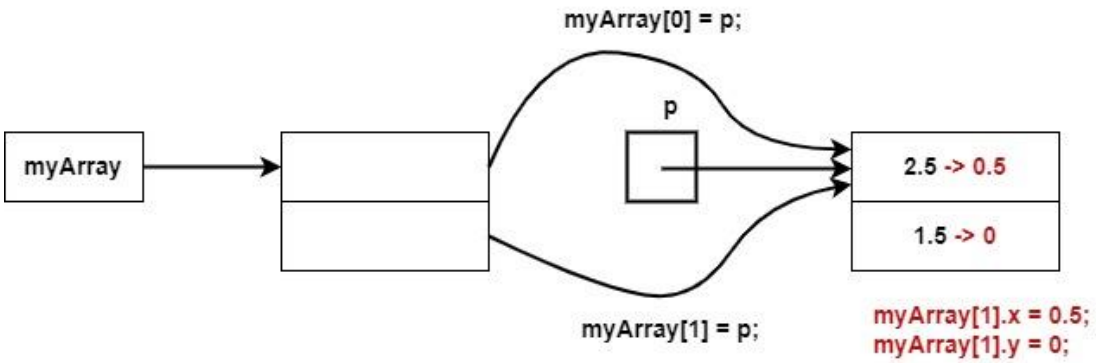

Abbildung 57: Visualisierung Kontrollfrage 10

Kontrollfrage 11

Was sind die Stärken von OOP? Nenne ein Beispiel.

Lösung:
Die Stärken von OOP sind:

- Vererbung, damit der geschriebene Code (Klassen) wieder verwendet werden kann.
- Polymorphismus, damit können Listen mit den Objekten der Basisklasse erstellt werden, welche auf einen Speicher einer abgeleiteten Klasse zeigen. Dies ist möglich, da erst zur Laufzeit den Typ bestimmt wird. (dynamische Bindung)
- Überschreiben von Methoden, damit kann in einer Subklasse die Funktionalität der Superklasse spezialisiert werden.

-

9 Kontrollaufgaben

Kontrollaufgabe 1

Erstellen Sie eine Klasse für die Komplexen Zahlen, damit Sie die Grundoperationen, Addition, Subtraktion, Multiplikation und Division ausführen können. Beachten Sie, dass eine Addition und Subtraktion mit der kartesischen Darstellung einfacher berechnet werden können, jedoch bei einer Multiplikation und Division die Darstellung in der Polarform geeigneter ist.
Schreiben Sie zudem ein Testprogramm für Ihre Klasse.

Lösung:

```
public class Complex {
  private double real;
  private double imag;
  Complex(double real, double imag){
    this.real = real;
    this.imag = imag;
  }
  Complex(Complex c){
    real = c.real;
    imag = c.imag;
  }
  public double real(){
    return real;
  }
  public double imag(){
    return imag;
  }
  public double phase(){
    double phase = 0;
    phase = Math.atan(imag / real);
    return phase;
  }
  public double value(){
    double value = 0;
    value = real*real + imag*imag;
    value = Math.sqrt(value);
    return value;
  }
  public Complex add(Complex c){
    double realtmp = real + c.real;
    double imagtmp = imag + c.imag;
    Complex res = new Complex(realtmp, imagtmp);
    return res;
  }
```

```
  public Complex sub(Complex c){
    double realtmp = real - c.real;
    double imagtmp = imag - c.imag;
    Complex res = new Complex(realtmp, imagtmp);
    return res;

  }
  public Complex mult(Complex c){
    double rtmp = value() * c.value();
    double phtmp = phase() + c.phase();
    Complex res = new Complex(rtmp*Math.cos(phtmp),
                              rtmp*Math.sin(phtmp));
    return res;
  }
  public Complex div(Complex c){
    double rtmp = value() / c.value();
    double phtmp = phase() - c.phase();
    Complex res = new Complex(rtmp*Math.cos(phtmp),
                              rtmp*Math.sin(phtmp));
    return res;

  }
  public void print(){
    System.out.println(real+" + i"+imag);
  }

  public static void main(String[] args) {
    Complex c1 = new Complex(3,4);
    Complex c2 = new Complex(2,3);
    Complex res = null;
    res = c1.add(c2);
    res.print();
    res = c1.sub(c2);
    res.print();
    res = c1.mult(c2);
    res.print();
    res = c1.div(c2);
    res.print();
  }
}
```

Kontrollaufgabe 2

Schreiben Sie eine Klasse **PointCnt**, in der die Anzahl erzeugter Objekte mitgezählt wird. Diesen Zähler kann mit einer **get**-Methode gelesen werden. Die Klasse **PointCnt** kann zudem noch die Koordinaten eines Punktes speichern, somit haben wir eine Erweiterung der Klasse **Point** mit einem Objektzähler.
Testen Sie ihre Klasse.

Lösung:

```
public class PointCnt extends Point{
    private static int objectCnt = 0;

    public PointCnt(){
        super();
        objectCnt++;
    }

    public PointCnt(double a, double b){
        super(a, b);
        objectCnt++;
    }

    public static int getObjectCnt(){
        return objectCnt;
    }

    public static void main(String[] args){
        PointCnt p1 = new PointCnt();
        PointCnt p2 = new PointCnt(3, 2);
        PointCnt p3 = new PointCnt(3.2, 4.5);

        System.out.println("Anzahl PointCnt Objekte: " +
                           PointCnt.getObjectCnt());
    }
}
```

Kontrollaufgabe 3

In Java existiert eine Klasse **Random**, mit der man eine Zufallszahl erzeugen kann.
Der Code für eine Zufallszahl zwischen **0** und **bound** (Obergrenze) lautet:

```
import java.util.Random;

//In der entsprechenden Methode
Random r = new Random()
r.nextInt(bound) ;
```

Ihre Aufgabe ist es nun eine eigene Klasse **RandomNumber** zu entwerfen, welche eine Methode **int getRandom()** enthält, die Ihnen eine Zufallszahl mit Hilfe der Klasse **Random** zwischen zwei beliebigen Werten erzeugt, welche in der Klasse **RandomNumber** als Attribute gespeichert sind. Die zwei Attribute stehen somit für die Untergrenze (**lowerBound**) und für die Obergrenze (**upperBound**).
Die Grenzen können nur im Konstruktor gesetzt werden. Beachten Sie, dass die Grenzen auch einen negativen Wert haben können, aber die Methode **nextInt(int bound)** der Klasse **Random** liefert nur positive Wert.
Schreiben Sie die Klasse **RandomNumber** und testen Sie sie.

Lösung:

```
import java.util.Random;

public class RandomNumber {
    private int upperBound;
    private int lowerBound;
    private Random random;

    public RandomNumber(){
        upperBound = 1;
        lowerBound = 0;
        random = new Random();
    }

    public RandomNumber(int lowerBound, int upperbound){
        this.lowerBound = lowerBound;
        this.upperBound = upperbound;
        random = new Random();
    }

    public int getRandom(){
        //z ist im Bereich 0 bis upperBouond-lowerBound
        int z = random.nextInt(upperBound-lowerBound);
        //for Test
        //System.out.println("0 bis upperBound: " + z);
        //z ist im Bereich lowerBound bis upperBouond
        z = z + lowerBound;
        return z;
    }
```

```
    public static void main(String[] args){
        RandomNumber randomNumber= new RandomNumber(-4, 10);
        for (int i = 0; i < 5; i++){
            System.out.println("Zufallszahl " + (i+1) + " :"+
                                randomNumber.getRandom());
        }
    }
}
```

Kontrollaufgabe 4

a) Schreiben Sie eine Klasse **Bike** mit den Attributen für die Marke (**brand:String**), für die Farbe (**color:String**), für den Besitzer (**owner:String**) und für die Geschwindigkeit in km/h (**velocity:double**).
Die Attribute **brand**, **color** und **owner** sollen direkt beim Erzeugen des Objektes, also im Konstruktor, gesetzt werden.
Die Attribute **owner** und **velocity** können je über eine Methode verändert und je über eine weitere Methode gelesen werden.
Ferner können alle Daten eines Objektes mit der Methode **void print(int data)** angezeigt werden. Der Parameter **data** steuert dabei die Ausgabe, ist **data == 0** so wird alles ausgegeben, bei **data ==1** werden nur die Marke und die Farbe ausgegeben, bei **data == 2** wird die Geschwindigkeit ausgegeben und mit **data == 3** wird der Besitzer angezeigt.

b) Mit der Methode **void start(double vel)** wird die Fahrt mit der Geschwindigkeit **vel** simuliert, indem das Attribut **velocity** gesetzt wird. Mit der Method **void stop()** wird die Fahrt beendet, d.h. es wird die Geschwindigkeit **velocity = 0** gesetzt.

c) Schreiben Sie eine zweite Klasse **EBike**, welche von der Klasse **Bike** abgeleitet wird. Die neue Klasse hat ein zusätzliches Attribut **charge:int**, welches den Ladezustand der Akku in **%** darstellt. Die Akkuladung kann mit **int getCharge()** gelesen und mit **void setCharge(int charge)** geschrieben werden.

d) Überschreiben Sie nun in der Klasse **EBike** die Methode **void start(int vel)** , welche nur die Fahrt startet, wenn die Akkuladung grösser als 30% ist, sonst wird auf der Konsole eine Meldung angezeigt. Überschreiben Sie auch die Methode **void print(int data)**, welche bei **data == 0** wiederum alles ausgibt, auch der Ladezustand und bei **data == 4** den Ladezustand anzeigt.

e) Schreiben Sie ein Testprogramm für die obige Klasse

Lösung:

```
public class Bike {
    private String brand;
    private String owner;
    private String color;
    private double velocity;
```

```
    public Bike(String brand, String color, String owner){
        this.brand = brand;
        this.color = color;
        this.owner = owner;
        this.velocity = 0;
    }

    public void setOwner(String owner){
        this.owner = owner;
    }
    public String getOwner(){
        return owner;
    }
    public void setVelocity(double velocity){
        this.velocity = velocity;
    }
    public double getVelocity(){
        return velocity;
    }
    public void print(int data){
        switch(data){
        case 0:
            System.out.println("Marke: " + brand);
            System.out.println("Farbe: " + color);
            System.out.println("Besitzer: " + owner);
            System.out.println("Geschwindigkeit: " + velocity);
            break;
        case 1:
            System.out.println("Marke: " + brand);
            System.out.println("Farbe: " + color);
            break;
        case 2:
            System.out.println("Geschwindigkeit: " + velocity);
            break;
        case 3:
            System.out.println("Besitzer: " + owner);
            break;
        }
    }
    public void start(double vel){
        velocity = vel;
    }
    public void stop(){
        velocity = 0;
    }
}
```

```
public class EBike extends Bike {
    private int charge;


    public EBike(String brand, String color, String owner){
        super(brand, color, owner);
        charge = 100;
    }

    public void setCharge(int charge){
        this.charge = charge;
    }
    public int getCharge(){
        return charge;
    }

    public void start(double vel) {
        if (charge > 30){
            super.start(vel);
        }else{
            System.out.println("Die Akkuladung ist zu schwach.");
        }
    }

    public void print(int data) {
        super.print(data);
        if (data == 0){
            System.out.println("Akkuladung: " + charge + "%");
        }else if (data == 4){
            System.out.println("Akkuladung: " + charge + "%");
        }
    }
}
```

Kontrollaufgabe 5

Eine doppelt verkette Liste hat im Knoten nicht nur ein Objekt **`next`**, sondern auch ein Objekt **`previous`**.

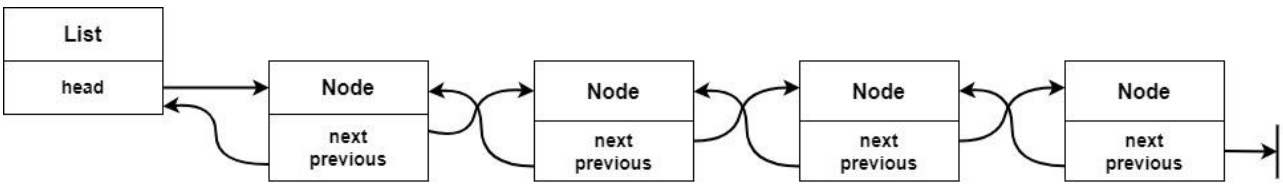

Abbildung 58: doppelt verkettete Liste

Überlegen Sie sich und zeichnen Sie eine Graphik mit den Schritten, die erforderlich sind, um einen Knoten in die doppelt verkettete Liste einzufügen. Sie müssen das Einfügen am Anfang und am Ende **nicht** berücksichtigen.

Lösung:

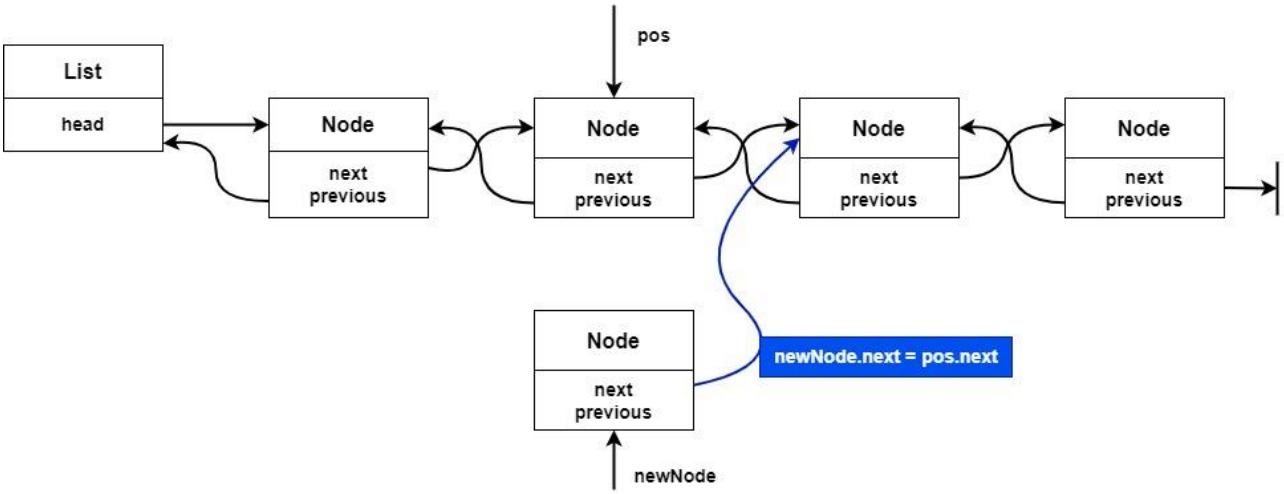

Abbildung 59: Schritt 1: Einfügen neuer Knoten in eine doppelt verkettete Liste.

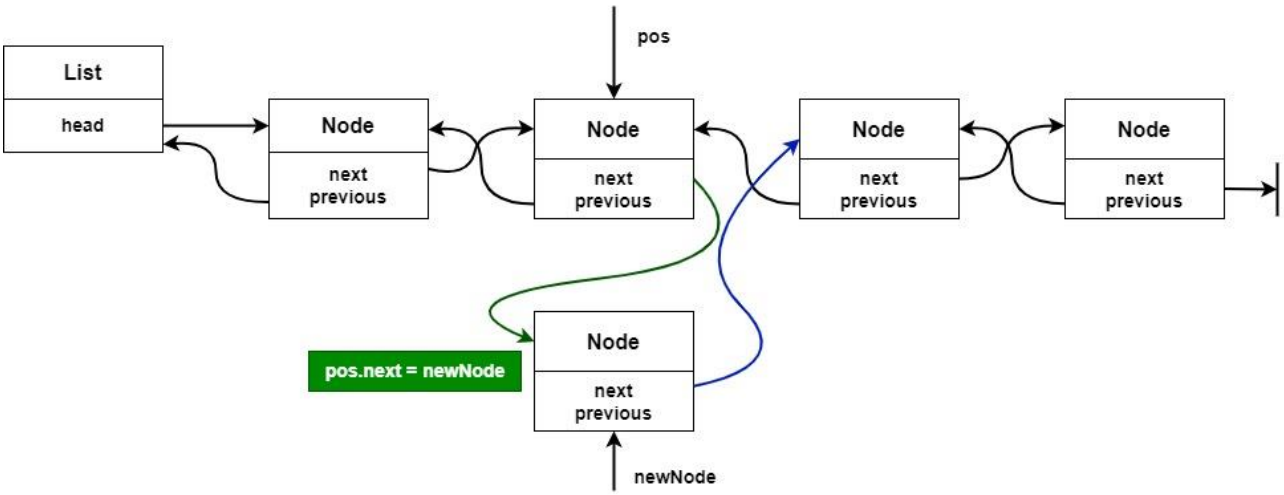

Abbildung 60: Schritt 2: Einfügen neuer Knoten in eine doppelt verkettete Liste.

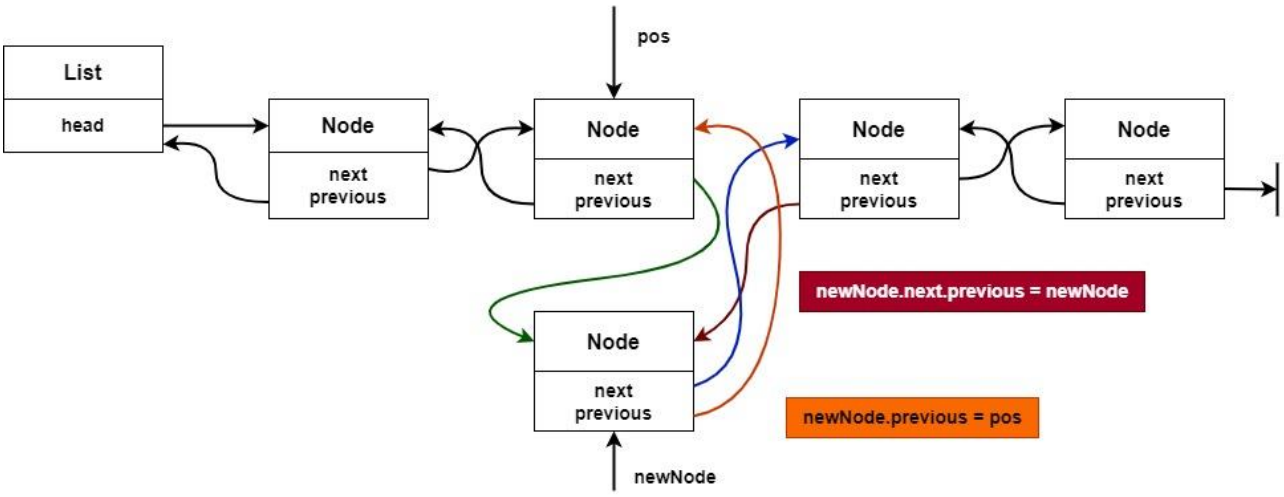

Abbildung 61: Schritt 3: Einfügen neuer Knoten in eine doppelt verkettete Liste.

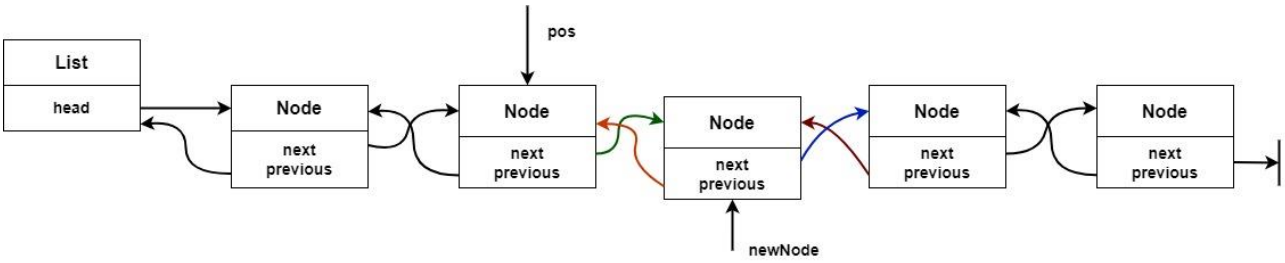

Abbildung 62: Neuer Knoten wurde in eine doppelt verkettete Liste eingefügt.

Kontrollaufgabe 6

Von einem Projekt kennen Sie das Testprogramm. Erstellen Sie alle notwendigen Klassen damit das Testprogramm fehlerfrei ausgeführt werden kann.

```
public class TestFood {
 void main(String[] args){
        Food[] food = new Food[3];
        Food f = new Milk();
        food[0] = f;
        f = new Fish();
        food[1] = f;
        f = new Eggs();
        food[2] = f;

        for (Food food2 : food){
            food2.printPrice();
            food2.printfromCity();
        }
    }
}
```

Dies Ausgabe des Testprogrammes ist die folgende:

1.90 CHF pro Liter
Ursprung: St. Gallen, Hof Inauen
4.50 CHF pro 100 g.
Ursprung: Norwegen, Fischzucht.
0.60 CHF pro Stück
Ursprung: Wil, Hof Brunner.

Lösung:

```
public class Food {
    public void printPrice(){
        System.out.println("Dies ist kein gültiger Aufruf.");
    }
    public void printfromCity(){
        System.out.println("Dies ist kein gültiger Aufruf.");
    }
}
```

```
public class Milk extends Food{
    public void printPrice(){
        System.out.println("1.90 CHF pro Liter");
    }
    public void printfromCity(){
        System.out.println("Ursprung: St. Gallen, Hof Inauen");
    }
}
```

```
public class Eggs extends Food{
    public void printPrice(){
        System.out.println("0.60 CHF pro Stück");
    }
    public void printfromCity(){
        System.out.println("Ursprung: Wil, Hof Brunner.");
    }
}
```

```
public class Fish extends Food{
    public void printPrice(){
        System.out.println("4.50 CHF pro 100 g.");
    }
    public void printfromCity(){
        System.out.println("Ursprung: Norwegen, Fischzucht.");
    }
}
```

Kontrollaufgabe 7

In dieser Aufgabe werden wir geometrische Objekte verwalten, damit die Aufgabe nicht zu gross wird und wir trotzdem den Nutzen der OOP zeigen können, beschränken wir uns auf Rechtecke und Kreise.
Ihre Aufgabe ist nun diese Verwaltung zu implementieren.

a) Ein geometrisches Objekt, kann seine Fläche und seinen Umfang berechnen, alle Masseinheiten sollen als cm interpretiert werden. Erstellen Sie die Klassen **`GeoObjekt`**, **`Rechteck`** und **`Kreis`**.
b) Eine weitere Funktonalität ist das Zeichnen der Objekte. Da wir aber keine Einführung in AWT oder SWING gemacht haben, was für die graphische Darstellung von Objekten notwendig ist, wird hier nur die Ausgabe in der Konsole programmiert. Das bedeutet, dass nur einen Text erscheinen soll, z.B. Rechteck mit Breite 5 und Länge 8.
c) Erstellen Sie ein Klassendiagramm in UML für die Modellierung dieser Aufgabe.
   Erstellen Sie die notwendigen Klassen mit den Methoden aus Aufgabe a) in Java, um die Kreise und Rechtecke verwalten zu können.
d) Für die Verwaltung erstellen Sie eine Klasse **`CAD`**, welche ein Array mit den geometrischen Objekten enthält (Bemerkung: Wir verwenden hier ein Array als Übung, sinnvoller wäre hier eine Liste).
   Schreiben Sie eine Methode, um ein Objekt der Klasse **`GeoObject`** einzufügen.
   Die Klasse **`CAD`** kann die gesamte Fläche sowie den gesamten Umfang aller Figuren ermitteln. Über einen Parameter kann zudem der Methode angegeben werden, ob man die Flächen aller Kreise, die Fläche aller Rechtecke oder die Fläche aller Figuren berechnen will. Schreiben Sie eine Methode für diese Funktionalität.
e) Programmieren Sie eine weitere Methode in der Klasse **`CAD`**, welche die Anzahl Kreise, die Anzahl Rechtecke und die Anzahl aller geometrischen Objekte zurückgeben kann. Welcher Wert zurückgegeben wird, ist analog der Teilaufgabe d).
f) Vervollständigen Sie das UML-Klassendiagramm mit der Klasse **`CAD`**.
g) Schreiben Sie ein Testprogramm.

Lösung:

a) und b) Java Klassen

```
public class GeoObject {
    public double calcArea(){
        System.out.println("Not implemented!");
        return 0;
    }

    public double calcCircumference(){
        System.out.println("Not implemented!");
        return 0;
    }

    public void draw(){
        System.out.println("Not implemented!");
    }
}
```

```
public class Rectangle extends GeoObject{
    private double wide;
    private double height;

    public Rectangle(double wide, double height){
        this.wide = wide;
        this.height = height;
    }
    public double calcArea(){
        double area;
        area = wide * height;
        return area;
    }

    public double calcCircumference(){
        double circ;
        circ = 2*wide + 2*height;
        return circ;
    }

    public void draw(){
        System.out.println("Rechteck mit Breite: " + wide
                           + " Höhe: " + height);
    }
}
```

```
public class Circle extends GeoObject{
    private double radius;

    public Circle(double radius){
        this.radius = radius;
    }
```

```
    public double calcArea(){
        double area;
        area = Math.pow(radius, 2) * Math.PI;
        return area;
    }

    public double calcCircumference(){
        double circ;
        circ = 2 * radius * Math.PI;
        return circ;
    }

    public void draw(){
        System.out.println("Kreis mit Radius: " + radius);
    }
}
```

c) UML

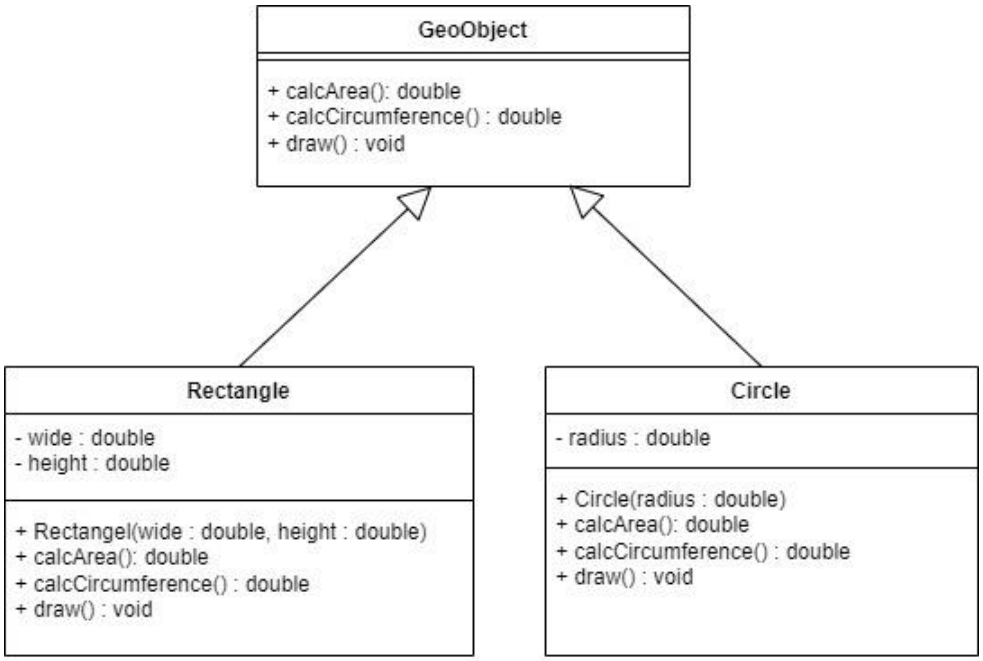

Abbildung 63: UML-Klassendiagramm der geometrischen Objekten

d) und e) Klasse CAD

```
public class CAD {
    private GeoObject[] myGeoObjects;
    public int countObjects = 0;

    CAD(int n){
        myGeoObjects = new GeoObject[n];
    }

    public void insertGeoObject(GeoObject geoObject){
        myGeoObjects[countObjects] = geoObject;
```

```
        countObjects++;
    }

    //what:
    //0: all areas,
    //1: only areas of all the circles,
    //2: only areas of all the rectangle
    public double calcAreas(int what){
        double areas = 0;

        if (what == 0){
            for (int i = 0; i < countObjects; i++){
                areas = areas + myGeoObjects[i].calcArea();
            }
        }else if (what == 1){
            for (int i = 0; i < countObjects; i++){
                if (myGeoObjects[i] instanceof Circle){
                    areas = areas + myGeoObjects[i].calcArea();
                }
            }
        }else if (what == 2){
            for (int i = 0; i < countObjects; i++){
                if (myGeoObjects[i] instanceof Rectangle){
                    areas = areas + myGeoObjects[i].calcArea();
                }
            }
        }
        return areas;
    }

    //what:
    //0: all circumferences,
    //1: only circumferences of all the circles,
    //2: only circumferences of all the rectangle
    public double calcCircumferences(int what){
        double circs = 0;

        if (what == 0){
            for (int i = 0; i < countObjects; i++){
                circs = circs +
                myGeoObjects[i].calcCircumference();
            }
        }else if (what == 1){
            for (int i = 0; i < countObjects; i++){
                if (myGeoObjects[i] instanceof Circle){
                    circs = circs +
                    myGeoObjects[i].calcCircumference();
                }
            }
        }else if (what == 2){
            for (int i = 0; i < countObjects; i++){
```

```
                if (myGeoObjects[i] instanceof Rectangle){
                    circs = circs +
                    myGeoObjects[i].calcCircumference();
                }
            }
        }
        return circs;
    }

    //what:
    //0: number of all GeoObjects,
    //1: number of all circles,
    //2: number of all rectangle
    public int getGeoObjects(int what){
        if (what == 0){
            return Circle.cntCircle + Rectangle.cntRectangle;
        }else if (what == 1){
            return Circle.cntCircle;
        }else if (what == 2){
            return Rectangle.cntRectangle;
        }
        return 0;
    }
}
```

In der Klasse **`Circle`** und **`Rectangle`** muss noch folgende Zeile eingefügt werden:

```
public static int cntCircle = 0;
```

```
public static int cntRectangle = 0;
```

f) Gesamtes UML-Klassendiagramm

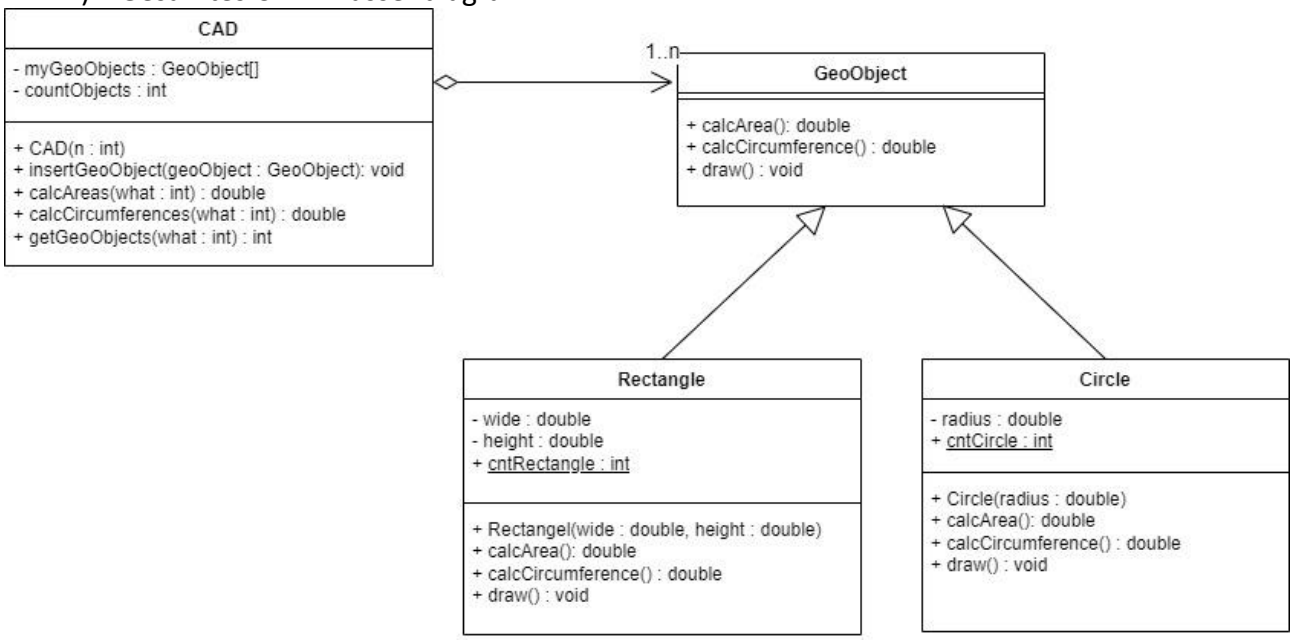

Abbildung 64: gesamtes UML-Klassendiagramm der geometrischen Objekten

g) Testprogramm

```
public class TestCAD {
    public static void main(String[] args){
        CAD cad = new CAD(10);

        Rectangle r = new Rectangle(5, 10);
        cad.insertGeoObject(r);
        Circle c = new Circle(3.2);
        cad.insertGeoObject(c);
        r = new Rectangle(2.5, 3.2);
        cad.insertGeoObject(r);
        r = new Rectangle(2.2, 2.8);
        cad.insertGeoObject(r);
        c = new Circle(0.5);
        cad.insertGeoObject(c);
        c = new Circle(1.7);
        cad.insertGeoObject(c);

        double a = cad.calcAreas(1);
        System.out.println("Die gesamte Fläche aller Kreise ist: "
                           + a);
        a = cad.calcAreas(0);
        System.out.println("Die gesamte Fläche aller Objekte ist:"
                           + a);

        a = cad.calcCircumferences(2);
        System.out.println("Der gesamte Umfang aller Rechtecke
                            ist: "+a);
        a = cad.calcCircumferences(1);
        System.out.println("Der gesamte Umfang aller Kreise ist: "
                           + a);

        int i = cad.getGeoObjects(1);
        System.out.println("Das CAD hat " + i + " Kreise.");
    }
}
```

10 Abbildungsverzeichnis

11 Literaturverzeichnis

Goll, J., & Heinisch, C. (2016). *Java als erste Programmiersprache.* Wiesbaden: Springer Vieweg.

Hromkovič, J., Gallenbacher, J., Komm, D., Lacher, R., & Pierhöfer, H. (2023). *Informatik Algorithmen und Künstliche Intelligenz.* Zürich: Klett und Balmer.

Krüger, G. (2000). *Go To Java 2.* Deutschland: Addison-Wesley.

Mössenböck, H. (2005). *Sprechen Sie Java.* Heidelberg: dpunkt.