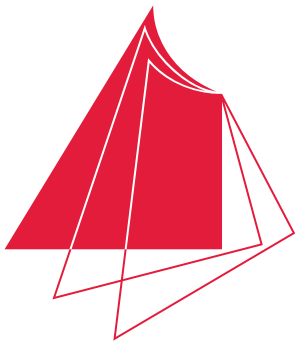
Informatik I Skript



Fakultät für Informatik und Wirtschaftsinformatik
Hochschule Karlsruhe für Technik und Wirtschaft

Das inoffizielle Skript zur Informatik I Vorlesung

geschrieben von Tobias Kerst

2014

Hinweis

Dieses Skript wurde erstellt, da eine vernünftige Zusammenfassung zur Informatik I Vorlesung an der Hochschule Karlsruhe zum Zeitpunkt dieser Ausarbeitung nicht vorlag. Das Ziel dieses Skripts ist es, eine grobe Zusammenfassung der am Anfang besprochenen Inhalte zu geben und Studenten das Lernen auf die Prüfung zu vereinfachen.
Das Skript besteht aus drei Teilen. Der erste Teil beschäftigt sich mit den grundlegenden Datentypen, mit der internen Verwaltung von gewissen Datentypen bei Java und der Umrechnung verschiedener Zahlensysteme. Dieser Teil wurde anhand der in der Vorlesung besprochenen Verfahren zusammengeschrieben.
Der zweite Teil ist eine Zusammenfassung der Programmierkonventionen in der Programmiersprache Java. Der Inhalt ist von der Webseite http://www.home.hs-karlsruhe.de/~pach0003/informatik_1/java_richtlinien/einleitung.html#uebersicht genommen, welche von Prof. Dr. Pape angefertigt wurde.
Der dritte Teil behandelt die Speicherverwaltung mittels Heap und Stack und die Modellierungsmöglichkeiten anhand der von UML zur Verfügung gestellten Aktivitäts- und Klassendiagramme. Hierzu wurden primär andere Quellen als Basis genommen, da die in der Vorlesung vorgestellten Konzepte sehr kurz gefasst wurden.
Dieses Skript ist ausschließlich zur privaten Nutzung gedacht, die Korrektheit der hier aufgeführten Informationen kann nicht gewährleistet werden. Solltest Du Fehler finden, werde ich diese gerne korrigieren. Ich hoffe, dass Dir dieses Skript beim Lernen für die Klausur hilft.

Tobias Kerst
2014

Inhaltsverzeichnis

1 Datentypen und Zahlensysteme

1.1 Einführung

Am Anfang dieses Skriptes stehen die verschiedenen Datentypen, die es in Java (und vielen anderen Programmiersprachen) gibt. Auch werden wir uns angucken, was Dual-, Dezimal und Hexadezimalzahlen sind und wie man diese ineinander umrechnen kann. Außerdem wird noch beleuchtet, wie Java negative Zahlen im Speicher repräsentiert und was es mit der *IEEE 754-Norm* auf sich hat.

1.2 Datentypen

Wenn man in Java programmiert, werden Konstanten und andere Werte in Variablen gespeichert, sodass man diese Werte später besser referenzieren kann. Diese Konstanten bezeichnet man heutzutage als `Literal`. Der `Datentyp` hingegen ist der Typ der Variable. Auf die Unterschiede dieser Datentypen gehen wir nun genauer ein. Am Ende der Ausführungen findet sich eine tabellarische Übersicht über die verschiedenen Datentypen, deren Literale und wie viel Platz diese im Speicher belegen.

1.2.1 Bool'sche-Literale

Es ist ein elementarer Bestandteil einer Programmiersprache, zu gucken, ob eine Aussage wahr oder falsch ist. Hierzu gibt es in Java den bool'schen Datentyp `Boolean`, der lediglich die Werte *true* und *false* annehmen kann. Anders als man das eventuell aus C++ kennt, existiert keine numerische Entsprechung, man kann also nicht false mit 0 gleichsetzen, oder irgendeinem anderen Wert.

1.2.2 Integer-Literale

Wenn man ganze Zahlen darstellen möchte, nutzt man hierzu für gewöhnlich das `Integer` Literal. Dieses kann Werte von etwa -2.000.000.000 (Milliarde) bis 2.000.000.000 darstellen. Genaueres kann man der Tabelle weiter unten entnehmen, welche die Wertebereiche und Größe etwas genauer darstellt. In Java kann man auch Integer-Literale dazu benutzen um Oktal-, oder Hexadezimalzahlen darzustellen. Im späteren wird darauf eingegangen, was das für Zahlen sind und wie man diese umrechnet. Für gewöhnlich reichen Integer Literale aus, um ganze Zahlen darzustellen. Für den Fall, dass der Platz nicht ausreicht, kann man auch auf das Long Literal zurückgreifen.

1.2.3 Long-Literal

Das **Long** Literal kann genau so wie das Integer Literal für ganze Zahlen benutzt werden, jedoch kann man im Long Literal größere Zahlen speichern, da dieses anstatt der 32 Bit, 64 Bit im Speicher einnimmt. Es wird aber dazu geraten, wenn nicht unbedingt nötig, ganze Zahlen als Integer zu speichern. Eine Besonderheit der Long Literale ist, dass diese ein *L*, oder *l* als Suffix haben. Möchte ich also den Wert 12 als Long Variable speichern, so schreibe ich `long variablenName = 12L`.

1.2.4 Gleitkomma-Literale

Es gibt zwei Möglichkeiten, Gleitkommazahlen darzustellen. Die normale Schreibweise und die wissenschaftliche. Die normale Schreibweie besteht aus einer ganzen Zahl, welche durch einen . von dem Bruchteil getrennt wird, bsp. 42.24, 3.1415. Wichtig: Man benutzt nach amerikanischem Vorbild einen Punkt und kein Komma.
Die wisenschaftliche Schreibweise, welche man im englischen auch als *Exponential Notation* bezeichnet, bildet man, indem man eine Gleitkommazahl schreibt und an diese ein *e*, oder *E* anhängt, welches die Zehnerpotenz angibt. Man kennt e ja auch als Basis des natürlichen Logarithmus, das ist hier *nicht* der Fall, in Java beschreibt *E* einen Exponenten zur Basis 10. Mit dieser Schreibweise kann man sehr große, oder sehr kleine Zahlen sehr schön darstellen. Möchte ich nun 876.000.000 darstellen, so kann ich auch $8.76E8$ schreiben. Aber auch 0.0000000987 lässt sich schöner als $9.87E-8$ darstellen.
Es gibt in Java zwei Datentypen, die man verwenden kann. Zum einen das `Float` Literal, welches 32 Bit groß ist, als auch das `Double` Literal (64 Bit). Um das Float Literal zu nutzen, schreiben wir die Gleitkommazahl, wie gerade beschrieben, und hängen als Suffix

ein F, oder f an. Somit kann man dann $42.24F$ (normale Schreibweise), oder aber $4.224E1f$ (wissenschaftliche Schreibweise) schreiben.

Wenn man das Suffix nicht an die Gleitkommazahl anhängt, konvertiert Java diese automatisch in ein Literal vom Typ `Double`, da dieses ohne Suffix auskommt. Somit sind die Zahlen 42.24 und $4.224E1$ vom Typ Double. Wir benutzen in der Regel Literale vom Typ Double.

1.2.5 Zeichen-Literale

Möchte man ein einzelnen Zeichen speichern, so setzt man dieses in einfache Anführungszeichen, welches man als `Char` Literal bezeichet. Solche Zeichen können beispielsweise *'§'*, *'z'*, oder *'T'* sein. Leider kann man über diese Methode nicht alle Zeichen angeben, da manche Zeichen eine besondere Funktion haben. *'\n'* ist zum Beispiel ein Symbol, welches eine Neue Zeile ausgibt, *'\t'* einen Tabulator. Und wenn man nun den Backslash in einem Zeichen Literal speichern möchte, hat man das Problem, dass dieses ja für die eben genannten Sonderzeichen benutzt. Deswegen ist für gewisse Zeichen eine sog. Escape-Sequenz nötig. '\\' speichert also den Wert '\'.

Es ist zudem möglich, mit dem Escapezeichen \ beliebige Zeichen des Unicode Zeichensatzes darzustellen, indem man eine besondere Schreibweise benutzt. *'\u0092'* steht für den Buchstaben *e*.

1.2.6 String-Literale

String Literale werden anders als die Zeichen-Literale mit doppelten Anführungszeichen angegeben und können mehrere Zeichen beinhalten. Dies ist sinnvoll, wenn man eine Nachricht ausgeben möchte, den Namen eine Person speichern möchte, oder alles, was sonst mit Text zu tun hat. Auch hier kann man Zeilenumbrüche durch ein *\n* hervorrufen, als Beispiel soll die Floskel *"Hallo \n Welt"* dienen. Es ist wichtig, vor allem wenn man das Thema Heap und Stack betrachtet, dass Strings keine primitiven Datentypen sind, sondern intern auch wie Objekte behandelt werden. Somit kann man auch auf den Strings Methoden aufrufen. Jedoch folgen Strings nicht der üblichen Syntax, die benutzt wird um Objekte zu initalisieren. Einen String initialisiert man wie folgt.

```
String helloWorld = "Hello World";
```

1.2.7 Zusammenfassung

Wie man sieht, gibt es viele unterschiedliche Datentypen. Es handelt sich bei den hier behandelten Datentypen um sog. **primitive Datentypen** (`Strings` sind hier, wie gerade besprochen ausgenommen). Das bedeutet, dass diese nur einen gewissen Wertebereich aufnehmen können. Sie besitzen eine festgelegte Anzahl von Werten, was auch als *Diskretheit* bezeichnet wird, aber auch eine feste Ober- und Untergrenze, wie man in der Tabelle 1.1 erkennen kann.
Außerdem ist es wichtig, dass man sich den begrenzten Speicherplatz bewusst macht. Das ist der Grund dafür, dass man reele Zahlen lediglich als gerundete Gleitkommazahl darstellen kann, worunter die Genauigkeit dann entsprechend leidet.

Tabelle 1.1 Die verschiedenen Datentypen in Java

Name	kleinster Wert	größter Wert	Größe in Bit	Literale
byte	-128	127	8	12
short	-32.768	32767	16	12
int	-2.147.483.648	2.147.483.647	32	12
long	$-9.2E^{18}$	$9.2E^{18}$	64	12L
char	\u0000	\uFFFF	16 (Unicode)	'a'
boolean	true, false		1	true, false
float	$-1.4E^{45}$	$3.4E^{38}$	32 (IEEE 754)	1.2F
double	$-4.9E^{324}$	$1.7E^{308}$	64 (IEEE 754)	1.2

1.3 Zahlensysteme

Die uns bekannten Zahlen sind die Dezimalzahlen und wie man an dem Teilwort *Dezi* erkennt, haben diese die Basis 10. Man kann die Zahl 53 also wie folgt darstellen

$$53_{10} = 5 \cdot 10^1 + 3 \cdot 10^0$$

Man kann aber nicht nur die natürlichen Zahlen darstellen, sondern auch Nachkommestellen und negative Zahlen sind möglich. Für die Nachkommastellen nimmt man einen negativen Exponenten, für negative Zahlen wird für gewöhnlich ein Vorzeichenbit gesetzt, das werden wir im späteren Verlauf kennen lernen. Gucken wir uns zunächst an, wie man Nachkommastellen bestimmen kann.

$$17,53_{10} = 1 \cdot 10^1 + 7 \cdot 10^0 + 5 \cdot 10^{-1} + 3 \cdot 10^{-2}$$

Jetzt möchten wir aber einen Schritt weiter gehen und eine Binärzahl, also eine Zahl mit der Basis 2, in eine Dezimalzahl (Basis 10) umwandeln.

$$\begin{aligned} 1101_2 &= 1\cdot 2^3 + 1\cdot 2^2 + 0\cdot 2^1 + 1\cdot 2^0 && (1.1)\\ &= 1\cdot 8 + 1\cdot 4 + 0\cdot 2 + 1\cdot 1 && (1.2)\\ &= 13_{10} && (1.3) \end{aligned}$$

Nach diesem Vorbild kann man Zahlen auch im Oktalsystem (Basis 8) und Hexadezimalsystem (Basis 16) darstellen.

$$\begin{aligned} 275_8 &= 2\cdot 8^2 + 7\cdot 8^1 + 5\cdot 8^0 && (1.4)\\ &= 128 + 56 + 5 && (1.5)\\ &= 189_{10} && (1.6) \end{aligned}$$

$$\begin{aligned} 10F_{16} &= 1\cdot 16^2 + 0\cdot 16^1 + 15\cdot 16^0 && (1.7)\\ &= 256 + 0 + 15 && (1.8)\\ &= 271_{10} && (1.9) \end{aligned}$$

1.3.1 Dezimalzahlen

- Folge der Ziffern 0 bis 9
- keine führende 0
- Vorzeichen möglich

Beispiel: 234, -23, 0 usw.

1.3.2 Hexadezimalzahlen

- Folge von 0 bis F ($F_{16} = 15_{10}$)

Beispiel: 011F, FFFA

1.3.3 Oktalzahlen

- Beginnen immer mit einer 0
- Folge von 0 bis 7

Beispiel: 0117, 0771

1.3.4 Umrechnung vom Dezimalsystem in Basis b

Man kann vom Dezimalsystem sehr einfach in ein anderes Zahlensystem umwandeln, indem man eine Dezimalzahl ganzzahlig durch b teilt. Der Rest gibt dann den Wert im Zielsystem an. Diesen Schritt wiederholt man so oft, bis der ganzzahlige Anteil 0 übrig bleibt. Folgendes Beispiel gibt Auffschluss über diese Rechnung:
Basis $b = 8$

$$97_{10} = 97 : 8 = 12 \text{ Rest } 1 \tag{1.10}$$
$$12 : 8 = 1 \text{ Rest } 4 \tag{1.11}$$
$$1 : 8 = 0 \text{ Rest } 1 \tag{1.12}$$

Ergebnis: $97_{10} = 141_8$
Das Ergebnis besteht aus den Restteilen, von unten nach oben gelesen.

1.3.5 Binärdarstellung

Die maximale Anzahl der Binärstellen bei einem int sind 32 Stellen (32 Bit), bei long sind es 64 Stellen (64 Bit). Die Zahl wird von hinten aufgefüllt und falls vorne noch Zeichen frei stehen, werden diese mit einer 0 aufgefüllt. Nehmen wir beispielsweise die Zahl 15_{10}, deren Binärdarstellung ja 1111_2 ist, so wird diese Zahl im Speicher als $00\ldots0001111$. Eine weitere Besonderheit, die vorher bereits schon einmal angesprochen wurde, ist die Darstellung der negativen Zahlen. Möchten wir eine negative Zahl schreiben, beispielsweise -15_{10}, so wird diese Zahl als $100\ldots0001111$ gespeichert. Es gibt jedoch ein besseres Verfahren um negative Zahlen zu speichern und zwar das 2er Komplement.

1.4 Das 2er Komplement

1.4.1 Von der Dezimalzahl zum 2er Komplement

Java benutzt die 2er Komplement Codierung, um negative Zahlen zu speichern. Dies ist sinnvoll, da kein Platz für $+$ und $-$ verschwendet werden muss. Zuerst wird das Vorzeichen abgetrennt und die Zahl im Dualsystem dargestellt, so wie wir das bisher gemacht haben. Im Anschluss wird das Komplement gebildet. Wir beenden das Ganze, indem wir noch 1 hinzuaddieren. Anschaulicher wird dies, wenn wir uns folgendes Beispiel angucken.

$$-89_{10} \Rightarrow 0101\ 1001 \tag{1.13}$$
$$\Rightarrow 1010\ 0110 \quad \text{(Komplement bilden)} \tag{1.14}$$
$$\Rightarrow 1010\ 0111 \quad \text{(1 hinzu addieren)} \tag{1.15}$$

Zu beachten ist, dass das Vorzeichen am Anfang erst einmal ignoriert wird. Dann wird einfach jedes Zeichen umgedreht (also das Komplement gebildet) und wie bereits beschrieben wird 1 hinzuaddiert. Außerdem gilt es zu beachten, dass der Wertebereicht für ein 8-Bit 2er Komplement von -128 bis 127 geht. Angenommen man addiert $127+1$, dann springt der Wert zu -128. Dies ist eine Form des ungewollten Überlaufs.

1.4.2 Vom 2er Komplement zur Dezimalzahl

Die 2er-Komplementdarstellung eignet sich erstklassig, um schnell sagen zu können, ob eine Zahl positiv, oder negativ ist. Ein Blick auf das erste Bit gibt uns darüber eine Auskunft, wie wir die Zahl lesen müssen. Ist es eine 0, so können wir die Zahl nach unserem bereits bekannten Schema einfach in eine Dezimalzahl umwandeln. Ist es eine 1, die Zahl also negativ, so gilt es diese wie folgt umzuwandeln.

$$1010\ 0111_2 \Rightarrow 0101\ 1000 \quad \text{(Komplement bilden)} \tag{1.16}$$
$$\Rightarrow 0101\ 1001 \quad \text{(1 hinzu addieren)} \tag{1.17}$$
$$\Rightarrow -89_{10} \quad \text{(Dezimalzahl bilden und Vorzeichen anhängen)} \tag{1.18}$$

1.5 IEEE 754

1.5.1 Gleitkommazahlen

Gleitkommazahlen können sehr unterschiedlich große Zahlen darstellen. So kann man riesig große Zahlen ($1,23 \cdot 10^{88}$) darstellen, aber auch winzig kleine ($1,23 \cdot 10^{-88}$) und Selbiges gilt auch für den negativen Bereich. Wie man bereits an der Notation sieht, die genutzt wird um diese riesigen Zahlenbereiche abzudecken, lohnt es sich, hier die wissenschaftliche Schreibweise zu nutzen, um Zahlen darzustellen. Um im Folgenden sprachliche Barrieren zu vermeiden möchte ich an dieser Stelle auf die verschiedenen Bezeichnungen der einzelnen Teile in der wissenschaftlichen Schreibweise eingehen.

$$x = v \cdot m \cdot b^e$$

- **v** ist das Vorzeichen
- **m** ist die Mantisse
- **b** ist die Basis (bei IEEE 754 $b = 2$)
- **e** ist der Exponent

Für das Vorzeichen v gilt im Folgenden, dass dieses aus einem Bit besteht und 0 für ein positives und 1 für ein negatives Vorzeichen steht. Man kann auch sagen, dass $v = (-1)^v$ repräsentiert. Es ist aber einfacher sich einfach zu merken, dass das gesetzte Bit ($= 1$) eine negative Zahl repräsentiert
Diese etwas ungewohnte Art der Darstellung soll an den folgenden beiden Beispielen deutlich werden.

$$-543,21 = (-1)^1 \cdot 5,4321 \cdot 10^2 \tag{1.19}$$

$$0,00543 = 1 \cdot 5,4321 \cdot 10^{-3} \tag{1.20}$$

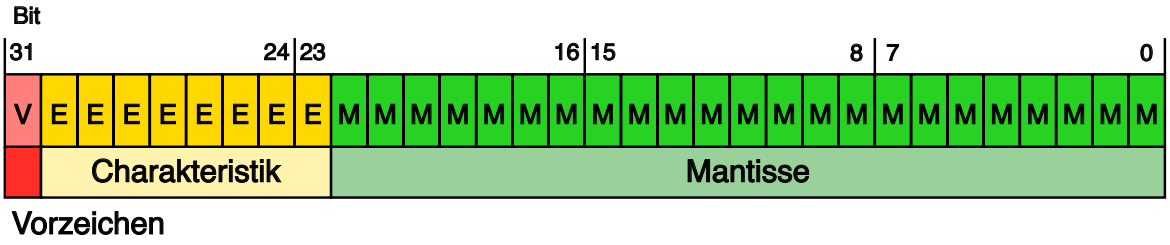
Wir müssen beachten, dass Gleitkommazahlen oft nur einen angenäherten Wert annehmen können, da reele Zahlen beispielsweise nicht in den Speicher passen (mit ihren unendlich vielen Nachkommastellen), aber auch Rundungsfehler das Arbeiten mit Gleitkommazahlen erschweren. Wie die Gleitkommazahlen in Computern dargestellt werden und was das alles mit diesem *IEEE 754* zu tun hat, wird in den folgenden Abschnitten erklärt.

1.5.2 Was ist das und warum wird es genutzt

Die IEEE 754 Norm wird genutzt um Gleitkommazahlen in Computern darzustellen und diese umzurechnen, aber auch, wie man mathematische Operationen mit diesen Zahlen durchführt und wie diese Zahlen zu runden sind. Es wird generell zwischen Datenformaten mit 32 Bit (*single precision*) und 64 Bit (*double precision*) Speicherbedarf unterschieden. Java nutzt `Float` für die Single-Precision Darstellung, und `Double` für die Double Precision. Die Single Precision Zahl benötigt den folgenden Speicherplatz für die einzelnen Teile, die zur Darstellung einer Gleitkommazahl notwendig sind

- Vorzeichen: 1 Bit
- Exponent: 8 Bit
- Mantisse: 23 Bit

Die Anordnung der verschiedenen Bits soll durch folgende Abbildung besser erklärt werden.



Die Double Precision Zahl benötigt hingegen folgenden Speicherplatz

- Vorzeichen: 1 Bit
- Exponent: 11 Bit
- Mantisse: 52 Bit

Die Mantisse wird bei der IEEE Kodierung im Bereich $1 \leq m \leq 2$ dargestellt. Da dann links immer eine 1 steht, vor dem Komma, müssen wir dieses nicht extra speichern und erhalten somit 1 Bit mehr Platz für die Mantisse.
Außerdem können noch einige Sonderfälle auftreten, die durch spezielle Bitmuster kodiert werden. Hierzu werden zwei besondere Exponentwerte benutzt, der Maximalwert ($e = 111\ldots11 = 2^r - 1$) und die Null ($e = 000\ldots00$). Mit dem maximalen Exponentwerten

werden die Sonderfälle *NaN* (Englisch für *Not a Number*) und ∞ kodiert. Mit der Null im Exponenten wird die Gleitkommazahl 0 dargestellt.
Jetzt, wo wir grob wissen, was es mit der IEEE 754 Norm auf sich hat, möchten wir uns einmal angucken, wie wir eine Dezimalzahl in eine solche IEEE 754 normierte single precision Zahl umwandeln.

1.5.3 Dezimalzahl in IEEE 754 umrechnen

Wir möchten das Umrechnen von einer Gleitkommazahl in eine IEEE 754 normierte Binärzahl am Beispiel von $-10,125$ erklären.

Vorzeichen

Da unsere Beispielzahl $-10,125$ negativ ist, setzen wir das Vorzeichenbit auf 1 und entfernen das Vorzeichen von der Zahl, sodass wir von nun an mit $10,125$ weiterrechnen werden.

In Binärzahl umwandeln

Nun wandeln wir $10,125$ in eine Binärzahl um.

$$\begin{aligned} 10,125_{10} &= 1\cdot 8+0\cdot 4+1\cdot 2+0\cdot 1+0\cdot 0.5+0\cdot 0.25+1\cdot 0.125 && (1.21)\\ &= 2^3+2^1+2^{-3} && (1.22)\\ &= 1010,001_2 && (1.23) \end{aligned}$$

Mantisse bilden, Exponent bestimmen

Nun, da wir die Binärzahl $1010,001_2$ haben, bilden wir daraus die Mantisse und können danach den Exponenten bestimmen. Zuerst verschieben wir das Komma so, dass nur noch eine 1 vor dem Komma steht.

$$1010,001 = 1,010001\cdot 2^3$$

Die Mantisse erhalten wir, wenn wir nun die vor dem Komma stehende 1 abtrennen und die hinteren Bits mit Nullen auffüllen (die Mantisse besteht ja bei der single precision aus 23 Bits). Somit erhalten wir folgende Mantisse:

$$0100010000\ 00000000000\ 000$$

Jetzt fehlt uns nur noch der Exponent. Den erhalten wir, wenn wir den Term von oben betrachten, in dem wir das Komma verschoben haben.

$$1,010001 \cdot 2^3$$

Hier erkennen wir, dass der Exponent 3 ist. Nun müssen wir 127 mit diesem Exponenten addieren und die Summe nur noch in eine Binärzahl umwandeln:

$$127 + 3 = 130_{10} = 1000\ 0010_2$$

Nun haben wir die drei notwendigen Bestandteile

$$\textit{Vorzeichen} \quad 1 \tag{1.24}$$

$$\textit{Exponent} \quad 1000\ 0010 \tag{1.25}$$

$$\textit{Mantisse} \quad 0100010000\ 00000000000\ 000 \tag{1.26}$$

aus denen wir nun die der IEEE 754 Norm entsprechende 32 Bit Gleitkommazahl zusammen fassen können

$$\underbrace{1}_{VZ}\ \underbrace{10000010}_{Exponent}\ \underbrace{01000100000000000000000}_{Mantisse}$$

1.5.4 IEEE 754 Zahl in Dezimalzahl umwandeln

Auch hier ist es einfacher, die Umformung anhand eines Beispiels zu zeigen. Nehmen wir die als IEEE 754 normierte Binärzahl 1 011111110 01101100, bei der wir die letzten Füllbits (also Nullen) weglassen.
An dem Vorzeichenbit 1 erkennen wir bereits, dass es sich um eine negative Zahl handelt. Nun gucken wir, wie groß der Exponent ist und in welche Richtung sich das Komma demzufolge verschieben wird. Dazu berechnen wir zuerst die Dezimaldarstellung.

$$01111\ 1110_2 = 126_{10}$$

Und ziehen hiervon 127 ab.

$$126 - 127 = -1$$

Das Komma verschiebt sich also nach links, die Zahl ist somit kleiner als 1. Wir nehmen nun wieder die Mantisse, fügen vorne eine 1 und folgend das Komma an und verschieben über den gerade berechneten Exponenten das Komma. Danach berechnen wir aus dieser Dualzahl die Dezimalzahl

$$
\begin{aligned}
1,\underbrace{01101100}_{Mantisse}\cdot 2^{\overbrace{-1}^{Exponent}} &= 0,101101100 && (1.27)\\
&= 0\cdot 2^0+1\cdot 2^{-1}+0\cdot 2^{-2}+1\cdot 2^{-3} && (1.28)\\
&\quad +1\cdot 2^{-4}+0\cdot 2^{-5}+1\cdot 2^{-6}+1\cdot 2^{-7} && (1.29)\\
&= 2^{-1}+2^{-3}+2^{-4}+2^{-6}+2^{-7} && (1.30)\\
&= 0.7109375_{10} && (1.31)
\end{aligned}
$$

Wenn wir nun noch das Vorzeichen beachten, welches ja eine negative Zahl forderte, erhalten wir die Dezimalzahl

$$-0.7109375$$

1.5.5 Nachkommastellen von Dualzahlen berechnen

Es gibt eine einfache Möglichkeit die Nachkommastellen von Dualzahlen zu berechnen. Auch an dieser Stelle ist ein Beispiel die beste Möglichkeit, das Verfahren zu verstehen. Wir wollen 0.625 als Dualzahl darstellen. Dies geht wie folgt:

$$
\begin{aligned}
0,625\cdot 2 = 1,25 \quad &| \quad \to 1 && (1.32)\\
0,25\cdot 2 = 0,5 \quad &| \quad \to 0 && (1.33)\\
0,5\cdot 2 = 1 \quad &| \quad \to 1 && (1.34)\\
&\Rightarrow \quad 0,101_2 && (1.35)
\end{aligned}
$$

Dies ist ein relativ einfaches Beispiel. Gucken wir uns nun an, wie wir die Zahl $0,4$ (näherungsweise berechnen können)

$$0,4 \cdot 2 = 0,8 \quad | \quad \rightarrow 0 \tag{1.36}$$
$$0,8 \cdot 2 = 1,6 \quad | \quad \rightarrow 1 \tag{1.37}$$
$$0,6 \cdot 2 = 1,2 \quad | \quad \rightarrow 1 \tag{1.38}$$
$$0,2 \cdot 2 = 0,4 \quad | \quad \rightarrow 0 \tag{1.39}$$

Wie man erkennt, wiederholt sich diese Folge, da man wieder bei $0,4$ auskommt. Es entsteht also die Binärzahl

$$0,0110\ 0110\ldots$$

1.6 Overflow

Wenn man in Java programmiert, muss man bei einigen Datentypen aufpassen, dass man keinen Overflow, oder aber Underflow erreicht. Wir haben ja bereits die Wertebereiche der verschiedenen Datentypen angeguckt. Dies ist vor allem gefährlich, da Java keine Exception schmeißt, sollte man den Wertebereich übertreten. Wählt man den größten positiven Wert ($011111\ldots11_2$) einer Zahl und addiert eine 1 hinzu, wird dieser zu $100000\ldots$ und somit zur kleinsten negativen Zahl. Nehmen wir der Einfachheit halber den Byte Datentyp, der von -128 bis 127 reicht.

```
byte b = 128; // Fehler, weil 127 der größte Wert ist
byte b = 127; //richtig

// Wenn man 1 addiert, erreicht man einen overflow
byte b = b + 1; // b = -128;

byte b = -128;
byte b = b - 3; // b = 125
```

Wie man sieht, bekommt man Werte, die man eventuell nicht erwartet hat. Man sollte deswegen in seine Programme Kontrollmechanismen einbauen, die die Operationen im vorhinein prüfen, sollte man ein solches Verhalten nicht wünschen.

2 | Java Konventionen

Java ist eine Hochsprache. Das heißt, dass die Sprache abstrakt ist und als solche nicht direkt von der Maschine gelesen werden kann. Hierzu ist ein sog. Compiler nötig, der aus der Hochsprache ausführbare Programme erstellt. Wenn man ein Javaprogramm schreiben will, kommt der Quellcode, also die Programmanweisungen in eine Quelldatei, welche mit .java endet. Diese wird dann vom javac Compiler in eine .class Datei umgewandelt. Diese Datei kann dann im Anschluss von dem Java Interpreter, der Java Virtual Machine gelesen und ausgeführt werden. Programme bestehen aus Symbolen, die alle eine eigene Bedeutung, oder Funktion haben

- Schlüsselwörter: public, class, static, double
- Bezeichner: gewicht, BodyMassIndex usw.
- Literale: 1.82, 182736292L, "Hallo Welt"
- Trennsymole: , [], =, *, +, -, / usw.

  Um den Code auch für Menschen gut lesbar zu machen, ist es nötig, einheitlichen Code zu schreiben, also den Programmcode immer nach dem selben Schema zu schreiben und zu strukturieren. Es gilt somit, semantisch und syntaktisch korrekten Code zu schreiben. Hier ist der in der Vorlesung festgelegte Standard zusammengefasst.

2.1 Syntax

2.1.1 Aussagekräftige Namen verwenden

Regel: Verwende für Bezeichner immer aussagekräftige und selbsterklärende Namen.

Regel: Verwende keine Abkürzungen in Bezeichner – schreiben diese Namen immer aus.

Begründung: Quelltexte werden dadurch für alle lesbarer und verständlicher.

Beispiel Statt `KFZ, BA, HS, DB` besser `Kraftfahrzeug, BundesagenturFuerArbeit, HochschuleKarlsruhe, DeutscheBahn` schreiben.

2.1.2 Nur eine Sprache verwenden

Regel: Vermische keine verschiedenen Sprachen (Deutsch, Englisch, . . .) in einer Klasse

Begründung: Quelltexte werden dadurch für alle lesbarer und verständlicher.

2.1.3 Nur alphanumerische Zeichen verwenden

Regel: Vermische keine verschiedenen Sprachen (Deutsch, Englisch, . . .) in einer Klasse

Begründung: Quelltexte werden dadurch für alle lesbarer und verständlicher.

Beispiel `Person, zugFahren, ueberpruefen, MAXIMALE_ANZAHL`

2.1.4 Verwende Upper Case Camel Style

Regel: Bei Bezeichner, die aus mehreren Teilwörtern bestehen, wird der erste Buchstabe jedes Teilworts (ab dem zweiten Teilwort) groß geschrieben, wenn die entsprechenden Wörter normal auseinander geschrieben würden.

Begründung: Damit lange Bezeichner lesbarer sind.

Beispiel `zugFahren, Donaudampfschifffahrt, HochschuleKarlsruhe`

2.2 Variablen

2.2.1 Anfangsbuchstabe kleinschreiben

Regel: Schreibe Variablen klein (Ausnahme sind Konstanten). Besteht eine Variable aus mehren Teilwörtern, so wird bei jedem folgenden Teilwort der erste Buchstabe groß geschrieben.

Begründung: Ein Bezeichner ist dadurch im Quelltext sofort als Variable erkennbar.

Beispiel `manfredMueller, anzahlPersonen, quersumme`

2.2.2 Konstanten mit Grossbuchstaben schreiben

Regel: Konstanten (static final) werden immer mit Grossbuchstaben bezeichnet und Teilwörter mit dem Unterstrich _abgetrennt.

Begründung: Ein Bezeichner ist dadurch im Quelltext sofort als Konstante erkennbar.

2.2.3 Zählvariablen

Regel: Verwende die Buchstaben `i, j, k, l` für rein technische ganzzahlige Schleifenvariablen.

Begründung: Aus historischen Gründen.

```
for (int i = 0; i < personen.length; i++) {
   ...
}
```

Für die Aufzählung aller Elemente eines Feldes oder einer Datenstruktur des Java Collection Frameworks, sollte aber möglichst die ab JDK 5.0 zusätzliche for-Schleife verwendet werden. Die technischen Schleifenvariablen werden dann ganz vermieden:

```
for (Person person : personen) {
  ...
}
```

2.2.4 this Objektattributen voranstellen

Regel: Um besser zwischen Objektattribute und lokalen Variablen zu unterscheiden, sollten Objektattribute mit this referenziert werden.

Begründung: Die Quelltexte werden lesbarer und Fehler werden vermieden.

Beispiel

```
this.name = name; // wobei name ein Parameter ist
```

2.2.5 Ungarische Notation vermeiden

Regel: Verwende keine Ungarische Notation.

Begründung: Damit Quelltexte lesbarer und wartbarer werden.
Bei der Ungarischen Notation werden zwei Präfixe dem eigentlichen Bezeichner vorangestellt: Der erste für den Verwendungszweck des Bezeichners, der zweite für den Typ. Zum Beispiel p für Pointer, i für Integer: piAnzahl ist ein Zeiger auf einen Integerwert.

2.3 Methoden

2.3.1 Verb verwenden

Regel: Verwende mindestens ein Verb in Präsensform für eine Methode. Das Verb soll möglichst genau beschreiben, was die Methode macht.

Begründung: Damit Quelltexte verständlicher werden.

Beispiel `personSuchen()`, `loeschen()`, `produktVerkaufen()`

2.3.2 Getter/Setter

Regel: Verwende als Verb das englische Präfix `get` mit einem nachfolgenden Substantiv, wenn eine Methode (ohne Parameter) einen Wert, der durch das Substantiv beschrieben ist, zurückgibt. Bei getter-Methoden mit booleschen Rückgabewert, wird statt get meist is verwendet.

Regel: Verwende als Verb das englische Präfix `set` mit einem nachfolgenden Substantiv, wenn eine Methode ohne Rückgabewert und mit einem Parameter einen Zustandswert (insbesondere ein Attribut) eines Objekts ändert.

Begründung: Zur Wahrung des Geheimnisprinzips bei der Programmierung.

Beispiel `String getVorname(), void setVorname(String vorname), boolean isSchaltjahr(), void setSchaltjahr(boolean schaltjahr)`

2.4 Quelltextformatierung

2.4.1 Programmierstile

Regel: Mische in einer Klasse nie verschiedene Programmierstile.

Regel: Bei bestehenden Quelltexten gilt insbesondere, dass der bestehende Programmierstil übernommen oder überall in der Klasse geändert wird.

Begründung: Quelltexte werden dadurch für alle lesbarer.

2.4.2 Allgemeine Regel

Regel: Schreibe nie Zeilen mit mehr als 80 Zeichen.

Regel: Stelle den Editor so ein, dass er diese Grenze als vertikale Linie klar darstellt.

Begründung: Quelltexte werden dadurch lesbarer.
Die 80 Zeichengrenze hat historischen Gründe: frühe Terminals hatten max. 80 Zeichen pro Zeile. Heutzutage ist die Grenze je nach Bildschirmauflösung und Font etwas größer. Grundsätzlich sollte man aber nicht von einer zu hohen Anzahl Zeichen pro Zeile ausgehen. Überlange Zeilen sind für Menschen schlecht lesbar, da nur eine begrenzte Anzahl von „Wörtern“ (ca. 5-7) mühelos erfasst werden können ohne dass man das Gefühl bekommt angestrengt zu lesen. Des Weiteren ist das horizontale Scrollen bei überlangen Zeilen im Fenster im Gegensatz zum vertikalen mühsam, da bei Sprung auf die nächste Zeile, insgesamt der Quelltext sich sprunghaft nach links verschiebt (zu starker Kontextwechsel).

2.4.3 Einrückungen

Regel: Verwende bei Kontrollstrukturen wie if, else, while, ... immer geschweifte Klammer für die Anweisungen (auch bei Einzelanweisungen)

Regel: Rücke jede Anweisung um 2 bis 4 Zeichen nach rechts ein (immer konsistent die gleiche Anzahl von Zeichen und nicht mal 2, mal 3 und dann mal wieder 4 Zeichen)

Regel: Stelle den Editor so ein, dass bei Drücken der Tabulatortaste, immer Leerzeichen eingefügt werden.

Begründung: Bessere Lesbarkeit von Programmen und vermeiden von Programmierfehlern.

Beispiel

```
if (student.hatZulassung () && student.hatZugesagt()) {
  student.immatrikulieren();
}
```

Falls die Klammern bei einzelnen Anweisungen weggelassen werden, kann bei nachträglichem Hinzufügen von zusätzlichen Anweisungen oder anderen Änderungen vergessen werden, die Klammern zu setzten, da durch die Einrückung suggeriert wird, dass die Anweisungen zusammengehören.
Werden Tabulatore statt Leerzeichen im Quelltext eingefügt, dann geht aufgrund anderer Tabulatoreinstellungen bei einem anderen Editor die Struktur meist die Struktur verloren.
Wenn bei if und else Klammern weggelassen werden, dann gehört das else zum "nächsten"vorangehenden if. Dies kann zu ungewollten Fehlern führen. Im folgenden Beispiel scheint das else zum ersten if zu gehören, weil es nicht eingerückt ist. Tatsächlich gehört es zum zweiten if!!!

Beispiel

```
if ( )
   if (  )
     ....
else
  ....
```

Wenn immer Klammern verwendet werden, ist es für jeden Programmierer eindeutig, zu welchem if ein else gehört:

```
if ( ) {
   if (  ) {
     ....
   }
} else {
  ....
}
```

2.4.4 else-if

Ausnahme obiger Regel: Folgt auf `else` direkt wieder ein `if`, so braucht das if nicht in ein Paar geschweifter Klammern eingeschlossen zu werden.

Begründung: Damit Quelltexte lesbarer werden.

Beispiel

```
if (a < 7) {
   a = a + 1;
} else if (a > 10) {
   a = a - 1;
} else if (a == 1) {
   ...
}
```

Dies vermeidet, dass bei einem kaskadierenden if-else if-else if - ... - else durch fortgesetztes Einrücken die Zeilen immer weiter nach rechts wandern und so der Quelltext unnötig tief verschachtelt ist. Man kann sich diese Regel auch so verstellen, dass damit im Gegensatz zu anderen Programmiersprachen ein in Java nicht vorhandenes Schlüsselwort „elsif“ simuliert wird.

2.4.5 Klammerungsstil

Regel: Schreibe entweder
a) die geschweifte öffnende Klammer auf die nächste Zeile oder

b) hinter der Kontrollanweisung ohne weitere Anweisungen hinter der öffnenden Klammer. Schreibe die zugehörige schließende Klammer immer auf eine neue Zeile ohne weitere Anweisungen oder Bezeichner davor oder dahinter (Ausnahme else).

Begründung: Damit Quelltexte kürzer und lesbarer werden.

Beispiel zu a)

```
if (a > 7)
{
   a = a + 1;
}
```

Beispiel zu b)

```
if (a > 7) {
   a = a + 1;
} else {
   a = a - 1;
}
```

2.4.6 Eine Anweisung pro Zeile

Regel: Schreibe jede einzelne Anweisung in eine separate Zeile.

Begründung: Damit Quelltexte lesbarer werden.

2.4.7 Continue und Break vermeiden

Regel: Vermeide die Schlüsselwörter continue und break, um Schleifen fortzuführen oder eine Kontrollstruktur abzubrechen. Ausnahme: break bei case-Anweisungen.

Begründung: Durch break und continue in Kontrollanweisungen wird der sequentielle Kontrollfluss unterbrochen. Dadurch ist dieser für Menschen nicht so leicht zu verstehen.

2.4.8 Verschachtelungstiefe von Kontrollanweisungen

Regel: Vermeide eine zu tiefe Verschachtlung von Kontrollanweisungen (ca. drei Kontrollanweisungen).

Begründung: Damit Quelltexte lesbarer werden.

2.4.9 Leerzeichen bei Operatoren

Regel: Setzte vor und nach jedem binären Operator ein Leerzeichen.

Begründung: Damit Ausdrücke bei langen Bezeichnern lesbarer werden.

Beispiel

```
a = 1 + 7 * (5 / a);
flaecheninhalt = kreisradius * kreisradius * 3.14159265;
```

2.4.10 Umbrechen von überlangen Ausdrücken

Regel: Breche überlange Zeilen wie folgt um:

- bei arithmetischen Operatoren vor einem Operator mit der schwächsten Bindung (Operatoren so auswählen, dass nicht zu viele Zeilenumbrüche entstehen)
- bei Methodenaufruf oder Parameterdeklaration nach einem Komma rücke den umgebrochenen Teil soweit nach rechts ein, dass er sich unterhalb des zugehörigen linken Teilausdrucks des Operators befindet mit zwei bis vier zusätzlichen Leerzeichen

Begründung: Damit Quelltexte lesbarer werden.

Beispiel

```
                          | <-- soll das Zeilenende markieren
a * a * a + 3 * a * a * b + 3 * a * b * b + b * b *b
```

Nicht bei *, sondern bei + umbrechen (+ bindet schwächer als *)

```
a * a * a + 3 * a * a * b
    + 3 * a * b * b + b * b *b
```

Beispiel

```
                                  |
x1 * 3 + (a + b + 6) * ((x2 + 8) + x3 / 11)
```

Bei + umbrechen im zweiten Faktor, wäre schlecht, da durch Einrücken, dann x3 / 11 über die Grenze hinausragt. Deswegen besser bei * umbrechen.

```
x1 * 3 + (a + b + 6)
          * ((x2 + 8) + x3 / 11)
```

Beispiel

```
public Address(String strasse,
               String hausnummer,
               int postleitzahl,
               String ortsname) {
  ...
}
```

2.4.11 Vergleich mit boolschen Werten

Regel: Vergleiche Boolesche Werte in einem Booleschen Ausdruck nicht mit == auf true oder false. Verwende stattdessen bei true, den Booleschen Ausdruck selbst und bei false dessen Negation (bei != entsprechend umgekehrt)

Begründung: Quelltexte werden leserlicher

Beispiel Anstatt folgenden boolschen Ausdruck

```
schaltjahr == true && volljaehrig == false
```

verwende besser

```
schaltjahr && ! volljaehrig
```

2.4.12 Operatorenreihenfolge

Regel: Bei mathematischen Vergleichen wie 0 < i < j < n behalte die Reihenfolge der Operatoren und Variablen in der Implementierung bei.

Begründung: Der resultierende Ausdruck ist dann sehr nah an der mathematischen Schreibweise orientiert und leichter verständlich, da er die ursprüngliche zu implementierende Form beibehält.

Beispiel

```
if (0 < i && i < j && j < n) {
   ...
}
```

Oder für $i = j = 7 < n$

```
if (i == j && j == 7 && 7 < n) {
   ...
}
```

2.4.13 Vermeide Seiteneffekte

Regel: Ein Ausdruck sollte keine Seiteneffekte haben, das heisst, ein Ausdruck sollte keine Zustandsänderungen durchführen.

Regel: Insbesondere sollten Funktionen keine Seiteneffekte haben.

Begründung: Vermeidung von Programmierfehlern und bessere Verständlichkeit von Quelltexten.

Beispiel Welchen Wert hat a, nachdem die letzte Anweisung ausgeführt wurde?

```
int a = 1;

a = (a = 2) + (a += a) * (a = 1 + a);
```

2.4.14 Vermische Kurzschlußoperatoren nie mit "normalen"Booleschen Operatoren

Regel: Vermische nie die Kurzschlußoperatoren (&&,||) mit den "normalen"Booleschen Operatoren (&, ^, |) in einem Ausdruck

Begründung: Fehlervermeidung und Wartbarkeit

Beispiel Welchen Wert hat folgender Ausdruck?

```
false && true | true
```

&& bindet **schwächer** als |. Deswegen ist der Wert des Ausdrucks false und nicht true, wie es *logischerweise* sein müsste.

2.4.15 Kurzer Methodenrumpf

Regel: Verwende nur so viele Zeilen für die Implementierung einer Methode, so dass die Methode komplett auf den Bildschirm passt (ca. 20-30 Zeilen), wenn möglich auch noch die zugehörige Javadokumentation.

Regel: Verlagere bei zu langen Zeilen zusammengehörende Teile in private Methoden.

Begründung: Um die Struktur und Funktionsweise einer Methode zu verstehen, sollte der Programmierer nicht gezwungen sein, vertikal zu scrollen: zugehörige Programmteile if-else sind dann oft nicht mehr auf einen Blick zu erfassen. Mit den Refactor-Funktionen der Entwicklungsumgebung geht eine derartige nachträgliche Formatierung des Quelltextes sehr schnell.

2.4.16 Deklaration lokaler Variablen

Regel: Deklariere alle lokalen Variablen (außer Schleifenvariablen) am Anfang der Methode gefolgt von einer Leerzeile.

Regel: Bei einer Funktion sollte insgesamt nur ein return in der Methode existieren (notwendigerweise am Ende der Methode). Setzte vor diesem return eine Leerzeile.

Begründung: Damit Methodenimplementierung verständlicher wird.

2.4.17 Primitive Datentypen

Regel: Verwende möglichst int statt byte, short oder long

Regel: Verwende möglichst double statt float

Regel: Mische möglichst keine verschiedenen Zahlentypen in einem Ausdruck.

Regel: Verwende wissenschaftliche Notation nur bei sehr kleinen oder sehr großen Zahlen.

Byte und short haben im Gegensatz zu int und long einen relativ kleinen Wertebereich. Da ein Überlauf in Java anderes als in C# nicht durch eine Ausnahme angezeigt wird, sollte möglichst mit Datentypen höherer Genauigkeit gerechnet werden. int ist meist ausreichend vom Wertebereich und schneller als long. Es ist dabei normalerweise auch nicht langsamer als byte oder short, da die interne Prozessorregister heutzutage mindestens 32bit breit sind. Es entfällt auch das bei long lästige Schreiben von l oder L hinter jeder Zahl. Wenn ein Java-Programm innerhalb einer Java Virtual Machine abläuft benötigen alle Werte entweder 32 Bit oder 64 Bit (long, double). Bei byte und short wird intern immer mit int-Werten gerechnet.

Float hat gegenüber double eigentlich nur den Vorteil halb so viel Speicher zu verbrauchen – Speicherverbrauch ist heutzutage aber kein Problem mehr. Die Berechnungen mit float sind normalerweise nicht schneller als mit double, da Mikroprozessoren intern mindestens mit der Genauigkeit von double rechnen. Teilweise kann float sogar langsamer sein, da die Konvertierung vom internen Gleitkommaformat zu float mehr Zeit kosten kann als zu double: insbesondere, falls die CPU intern immer mit double rechnet. Analog wie bei long entfällt auch das lästige Schreiben von f oder F hinter jeder Floatzahl.

Verschiedene Zahlentypen in einem Ausdruck zu mischen sollte vermieden werden, da dadurch unbeabsichtigt Fehler entstehen. Zum Beispiel ist der folgende Ausdruck immer 0, auch wenn x ein double Wert ist: $5/9 * x$. Ebenso kann es bei der impliziten Konvertierung von z.B. int (31 bit für positive Zahlen) nach float (23 Bit) zu Fehlern kommen - diese werden nicht vom Compiler angezeigt.
Die wissenschaftliche Notation sollte nur bei sehr großen oder sehr kleinen Zahlen (ab ca. 7-8 Ziffern) angewendet werden, da diese Notation für den Menschen sonst schwerer zu verstehen ist.

2.5 Javadoc

2.5.1 Allemein

Die durch Javadoc erstellte HTML Dokumentation dient Entwicklern dazu Klassen verwenden zu können ohne die Quelltexte lesen zu müssen oder wenn nur der Bytecode aber nicht die Quelltexte verfügbar sind. Die Javadoc-Kommentare müssen deswegen kurz und spezifisch für alle nicht private Member einer Klasse beschreiben

- was eine Klasse für eine Menge von Objekten beschreibt
- was ein Attribute bedeutet und welche Werte es annehmen kann (bzw. nicht darf)
- was eine Methode macht, was für ein Wert zurückgegeben wird und was die Parameter bedeuten und welche Werte sie annehmen dürfen (bzw. nicht dürfen) In einem Javadoc-Kommentar sollte nie beschrieben werden wie etwas implementiert ist, es sei denn es ist für die Verwendung der Klasse und deren Objekte notwendig.

Ein Javadoc-Kommentar ist ein mehrzelliger Kommentare der mit $/**$ eingeleitet und mit $*/$ beendet wird. Ob man im Javadoc-Kommentare in jeder Zeile auch noch einen zusätzlichen $*$ hinzufügt oder nicht, ist Geschmackssache. Der erste * in jeder Zeile innerhalb eines Javadoc-Kommentars wird ignoriert. Folgende Varianten sind in der resultierenden HTML-Dokumentation identisch. Man sollte sich generell für eine entscheiden. Am besten die Voreinstellung der Entwicklungsumgebung verwenden.

Wichtig Ein Kommentar muss so kurz wie möglich und spezifisch wie nötig sein.

2.5.2 Entwurfsentscheidung dokumentieren

Regel: Drücke im Klassenkommentar in einem Satz beginnende mit dem Klassennamen die wesentlichen Entwurfseinscheidung (Attribute und Beziehungen) und/oder die Verantwortung (abstrakte Beschreibung des Verhaltens der Objekte) aus.

Regel: Um das Geheimnisprinzip zu waren, vermeide möglichst jeden rein Implementierungstechnischen Bezug (z.B. Nennung von Datentypen).

Begründung: Zum besseren Verständis der Klasse

Beispiel Nehmen wir an es existieren die Klassen Hochschule, Student, Dozent und Studiengang.

```
/**
       Eine Hochschule mit Studenten, Dozenten und Studiengängen.
       */
public class Hochschule {

       }
```

Der obige Kommentar drückt den Entwurf der Beziehungen von Hochschule zu anderen Klassen aus: die Klasse Hochschule hat jeweils eine 1-n Beziehung zu Student, Dozent und Studiengang.

```
/**
       Ein Student mit Namen, Matrikelnummer und seine
       eingeschriebenen Studiengänge.
       */
public class Student {
       private String name;
       private String matrikelnummer;
       private Studiengang studiengaenge[];
}
```

Um eine Klasse besser verstehen und verwenden zu können, muss deren Zweck mindestens grob beschrieben sein. Ansonsten muss jeder Entwickler sich aus den Detailinformationen (Methoden, Attribute) den Zweck der Klasse selbst zusammenreimen. Um die Klasse zu

ändern, muss deren Zweck durch Nennung der wesentlichen Entwurfsentscheidungen klar abgegrenzt sein. Ansonsten ist die Gefahr groß, dass die Klasse so erweitert wird, dass sie die ursprünglichen Entwurfszielen nicht mehr erfüllt sind: normalerweise wird dadurch das Programm komplexer, schwerer zu verstehen und fehleranfälliger.

2.5.3 Beschreibe *was* eine Methode macht

Regel: Beschreibe immer was die Methode macht, nicht wie sie implementiert ist.

Regel: Fange den Kommentar mit dem Verb des Methodennamens in Präsensform an.

Regel: Führe alle Parameter mit dem @param-Tag auf und beschreiben sie.

Regel: Beschreibe die Bedeutung des Rückgabewerts bei @return mit einer kurzen Phrase

3 | Heap, Stack und die UML

In diesem Abschnitt werden noch einige weitere Themen behandelt, die prüfungsrelevant sind. Diese Themen werden lediglich angeschnitten und sollen einen Überblick geben. Die hier aufgeschriebenen Themen bestehen hauptsächlich aus den in den Tutorien übermittelteln Inhalten, da diese einen besseren Überblick vermittelten, als der in der Vorlesung vermittelte Stoff. Das kann auch der Grund für kleinere Abweichungen sein. Ich hoffe dennoch, dass die hier dargestellten Verfahren eine gute Übersicht geben können.

3.1 Heap und Stack

In Java gibt es zwei Speicherbereiche, den `Stack` (dieser wurde in der Vorlesung auch als `Laufzeitkeller` bezeichnet) und den `Heap`.
Der Stack speichert Methoden, Variablen und Parameter und funktioniert nach dem `LIFO` Prinzip (Last in First out). Das kann man sich wie eine Pringels Packung vorstellen. Man kann immer nur auf das oberste Element zugreifen, wenn dieses abgearbeitet wurde, kann man dann auf das darunter Liegende zugreifen usw.
Den Heap kann man sich als Wolke vorstellen, in dem die Objekte gespeichert werden. Dabei wird von der JVM (Java Virtual Machine) dafür gesorgt, dass genug Platz für die Objekte reserviert wird. Dieser Platz wird beispielsweise für Instanzvariablen benötigt. An dieser Stelle ist es wichtig, dass man versteht, dass diese Instanzvariablen nicht auf dem Stack leben, sondern im Objekt, welches auf dem Heap liegt. Für int-Literale werden 32 Bit Speicherplatz beansprucht, für long und double Werte 64 Bit, diese sind jedoch als Einheit zu betrachten, werden also nicht aufgeteilt in $2 \cdot 32$ Bit.
Eine Besonderheit ist, wenn eine dieser Instanzvariablen ein weiteres Objekt ist. Stellen wir uns eine Klasse `Person` vor, die als Instanzvariable einen Ehepartner vom Typ `Person` hat. Dann macht Java nicht in dem eigentlichen Objekt Platz für den Ehepartner, sondern initialisiert ein komplett neues Objekt vom Typ Person, und speichert in dem ursprünglichen Objekt eine Referenz (auch als `Zeiger` oder `Pointer` bezeichnet) zu dem neu erstellten

Objekt.

Das alles hört sich jetzt erst mal sehr komplex an, deswegen macht es Sinn, wenn man sich das Ganze mal anhand eines Beispiels anguckt. Dies wollen wir anhand des folgenden Codes machen:

```
Person meier;
Person mueller;

meier = new Person();
meier.alter = 74;
meier.nachname = "Meier";

mueller = new Person();
mueller.alter = 22;
mueller.nachname = "Müller";
```

Wenn wir die letzte Zeile des obigen Code-Beispiels durchlaufen, sehen Stack (links) und Heap (rechts) wie folgt aus. Die kryptische Bezeichnung 0x123 soll lediglich heißen, dass hier eine Adresse gespeichert wird, die auf das Objekt zeigt. Es handelt sich hierbei um eine Beispieladresse, da Objekte dynamisch allokiert wird (also der Speicher reserviert wird).

Nun wollen wir das oben angesprochene Konzept erweitern, indem wir die beiden Personen heiraten lassen. Hierzu fügen wir die Funktion `heiraten(Person person)` auf.

```
Person mueller;

meier = new Person();
meier.alter = 74;
meier.nachname = "Meier";

mueller = new Person();
mueller.alter = 22;
mueller.nachname = "Müller";

meier.heiraten(mueller);
```

Durch die Methode heiraten, wird die Adresse des ehepartners in der Variable Ehepartner gespeichert. Diese Adresse zeigt auf das Objekt des ehepartners. Da beide Personen miteinander verheiratet sind, zeigen sie jeweils aufeinander.

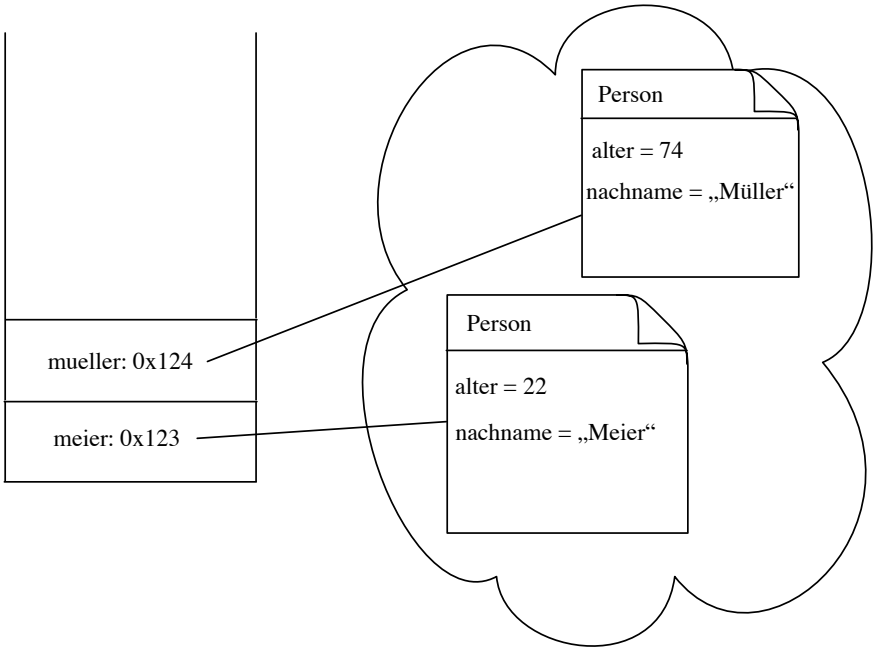


Abbildung 3.1 Die Adresse der zwei Objekte liegt auf dem Stack, die zwei Objekte liegen jedoch auf dem Heap.

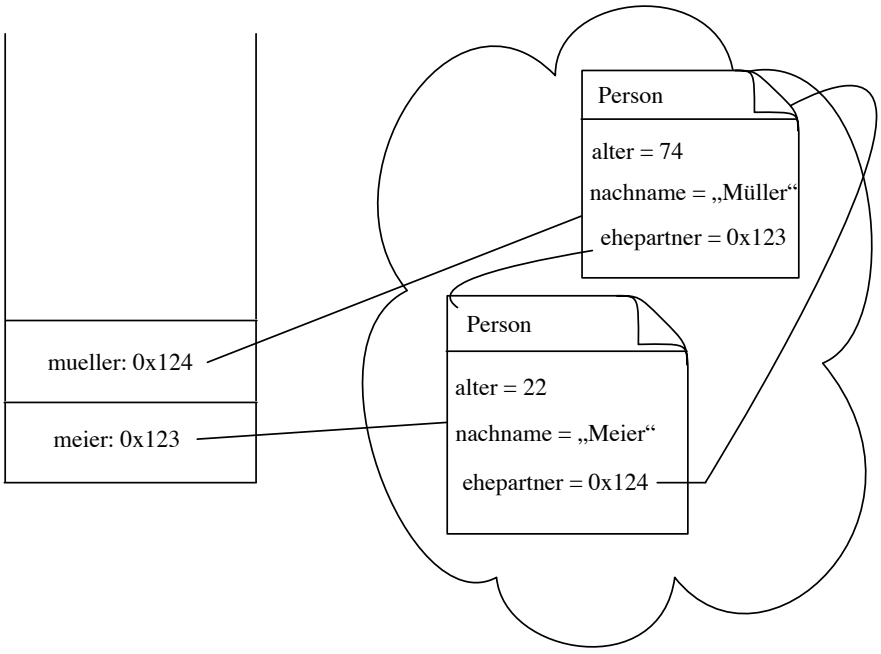


Abbildung 3.2 Der Stack und Heap nach dem Aufruf der Methode heiraten().

Es gilt zu beachten, dass hier die String Literale, die in `nachname` gespeichert werden, hier vereinfacht wie Instanzvariablen behandelt wurden. In der Realität sind Strings jedoch auch Objekte und somit würde nachname eigentlich auf ein anderes Objekt vom Typ String zeigen, welches dann den Nachnamen darstellt. Ich habe der Anschaulichkeit halber die Strings jedoch wie Instanzvariablen behandelt.

Garbage Collector

Ein interessantes Konzept ist das des Garbage Collectors, welches hier nur angeschnitten werden soll. Dieser ist für die Speicherverwaltung auf dem Heap zuständig und sorgt dafür, dass nicht mehr benutzte Objekte aus dem Heap genommen werden, um Speicherplatz zu sparen. Es gibt verschiedene Möglichkeiten, zu bestimmen, welche Objekte wann nicht mehr gebraucht werden, jedoch ist dies ein eigenes Feld für sich und wird eventuell in einer späteren Vorlesung genauer besprochen.

3.2 UML

Die `Unified Modeling Language` ist eine standardisierte Modellierungssprache, die 1997 aus vielen verschiedenen Modellierungssprachen in der Version 1.0 vereint wurde. Diese Modellierungssprache wird benutzt, um Objekte abstrahiert darzustellen, sodass man ohne die genaue Logik zu verstehen, eine ungefähre Ahnung hat, was ein Programm macht. Das Vereinfachen von komplexeren Abläufen ist also die Grundidee von UML. Ein Modell ist damit kein exakter Nachbau eines Objektes, sondern eine Vereinfachung, welche das Original beschreibt. Dies bezweckt ein besseres Verständnis des Originals.

Trotz des Teilworts *Language* handelt es sich bei UML um Zeichnungen; diese werden als Diagramme bezeichnet. Es gibt unterschiedliche Arten von Diagrammen, die unterschiedliche Aspekte eines Programms berücksichtigen. Insgesamt gibt es 13 verschiedene Diagrammtypen, die entweder `Strukturdiagramme` sind, oder aber `Verhaltensdiagramme`. Strukturdiagramme sind (wer hätte es gedacht) Modelle einer Struktur und somit auch statisch. Hier werden Zusammenhänge beschrieben, was wir im Folgenden anhand der `Klassendiagramme` zeigen werden. Verhaltensdiagramme erklären die Abläufe eines Programms und sind aus diesem Grund dynamisch. `Aktivitätsdiagramme` zählen zu den Verhaltensdiagrammen.

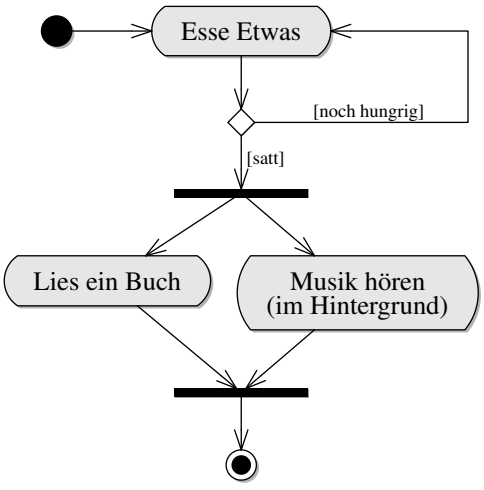
3.3 Aktivitätsdiagramme

Aktivitätsdiagramme beschreiben den Ablauf, wie ein Programm funktioniert, besonders die Eigenschaft, dass die Reihenfolge dargestellt wird ist teilweise sehr praktisch, um nachzuvollziehen, was ein Programm macht. Prinzipiell kennt ein Aktivitätsdiagramm nur zwei Arten von Bausteinen. Knoten und Kanten. Knoten sind die Stellen, an denen etwas passiert, und werden deshalb auch als `Aktivitäten` bezeichnet; Kanten verbinden diese Aktivitäten miteinander. Es gibt zudem noch Startknoten und Endknoten. Wichtig: Es ist zwar unüblich, dass mehrere Start- und/oder Endknoten vorkommen, aber dies ist nicht verboten.

Wie der Name es schon impliziert, beginnt eine Prozedur bei einem Startknoten. Man kann sich vorstellen, dass ein `Token` von dort aus über die Kanten zu den einzelnen Knoten fährt, dort eine Aktion ausgeführt wird und das Token im Anschluss weiterfährt, bis es zu einem Endknoten gelangt.

Wir möchten uns nun angucken, wie ein solches Aktivitätsdiagramm aussieht.

Beispiel



Das oben abgebildete Diagramm beinhaltet alle wichtigen Komponenten, die nötig sind, wenn man einen Ablauf beschreibt. Es ist schnell erkenntlich, was dort dargestellt wird. Man isst etwas. Es wird geprüft, ob man satt ist, ist dies nicht der Fall, isst man so lange weiter, bis man satt ist. Im Anschluss liest man ein Buch und parallel dazu hört man Musik. Außerdem sieht man hier, wie die Knoten und Kanten aussehen. Wir wollen jetzt etwas genauer auf die verschiedenen Objekte eingehen.

3.3.1 Start- und Endknoten

Es muss für einen Menschen, der sich mit dem Programm, bzw. der Prozedur noch gar nicht auskennt, schnell ersichtlich sein, wo ein Programm beginnt und wann dieses beendet ist. Dazu sind spezielle Knoten wichtig, die unten abgebildet sind. Der ausgefüllte Knoten ist ein Startknoten, der ausgefüllte Knoten mit dem Kreis drum, ist ein Endknoten.

3.3.2 Aktionen

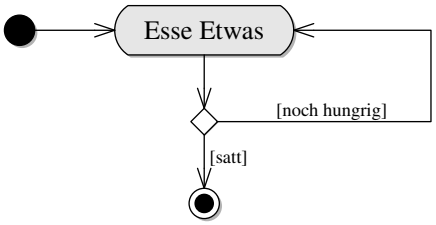
Aktionen werden durch Knoten mit einem rechteckigen Kasten und abgerundeten Ecken dargestellt. Darin steht kurz und knapp, welche Aktion bei diesem Knoten ausgeführt wird. Es ist eventuell sinnvoll, sich vorzustellen, dass jede Aktion der Aufruf einer Funktion im Programm ist.

3.3.3 Verzweigungen

Verzweigungen sind sinnvoll, um ein komplexeres Programm zu schreiben. Im Beispiel oben soll beispielsweise so lange gegessen werden, bis man satt ist. Somit gibt es nach der Aktion essen zwei Möglichkeiten: Weiter essen, oder zur nächsten Aktion springen. Bereits die Fomulierung ähnelt sehr stark einer Schleife aus der Programmierung.

```
do {
        nimmNahrungAuf();
} while (hungrig());
```

Dies entspricht diesem Aktivitätsdiagramm:

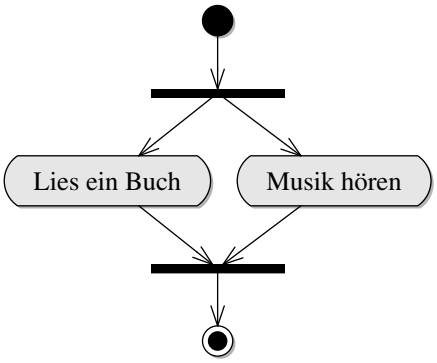


Nehmen wir nochmal das Token, das über die Kanten zu den verschiedenen Knoten fährt. Dann nimmt dieses Nahrung auf, fährt zu der Verzweigung und es wird geguckt, ob genug

Nahrung aufgenommen wird. An den ausgehenden Kanten sieht man, welche der jeweiligen Wege das Token einschlagen wird, da dort (in eckigen Klammern) die Bedingung zum weiterfahren steht. Somit hat es etwas von einem if statement, welches true, oder false zurückgibt. Somit sind sowohl sehr unterschiedliche Wege möglich und im speziellen, wie im Beispiel, Schleifen. Verzweigungen führen jedoch nicht immer nur verschiedene Kanten in verschiedene Richtungen, sie werden auch genutzt um unterschiedliche Pfade zusammen zu führen.

3.3.4 Gabelungen

Bei Verzweigungen wird ja das Token nur auf eine der ausgehenden Kantenlinien weitergeleitet. Möchte man jedoch bestimmte Schritte nebeneinander ausführen (Stichwort: Multitasking), so sind `Gabelungen` nötig. Hier wird das Token kopiert und nimmt jede der möglichen Ausgangskanten. Dies sieht dann so aus:



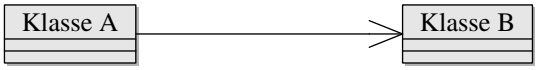
Bei diesem Aktivitätsdiagramm wird zur selben Zeit gelesen und Musik gehört. Am Ende, wenn Beides beendet ist, terminiert das Programm. Es ist sehr wichtig zu beachten, dass beim zusammenführen auf die anderen noch laufenden Prozesse gewartet wird. Erst wenn alle Aktionen abgeschlossen sind und bei der Gabelung (die die Kanten wieder zusammen fasst) eintreffen, läuft das Programm weiter.

Nehmen wir als Beispiel den Boxenstopp eines Rennautos: Das Auto fährt in die Boxengasse und ab dem Zeitpunkt wo es hält, werden unterschiedliche Aktionen (parallel) ausgeführt: Die Reifen werden gewechselt, das Rennauto wird betankt, das Visier des Rennfahrers wird eventuell geputzt usw. Der Rennfahrer darf aber erst weiterfahren, wenn alle diese Aktionen abgeschlossen sind, sonst steckt eventuell noch der Tankrüssel im Wagen.

3.4 Klassendiagramme

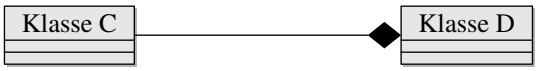
`Klassendiagramme` werden dazu benutzt um den Aufbau von Klassen zu beschreiben und die Zusammenhänge zwischen einzelnen Klassen darzustellen. Es ist sehr praktisch objektorientierte Projekte anhand dieser Klassendiagramme zu planen, da man die einzelnen Eigenschaften der Objekte sehr schön darstellen kann und somit einen engen Bezug zur Wirklichkeit hat. Aber bevor wir auf diese Eigenschaften näher eingehen wollen, möchten wir uns erst einmal die Beziehungen zwischen den Klassen ansehen.

3.4.1 Assoziation



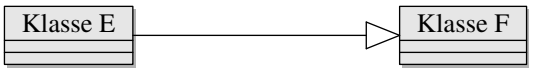
Die Verbindungslinie zwischen der `Klasse A` zu der `Klasse B` bedeutet, dass `Klasse A` auf `Klasse B` zugreifen kann. Die Pfeilrichtung gibt an, welche Klasse auf wen Zugriff hat, es ist auch möglich, dass beiden Klassen aufeinander zugreifen, dann ist an beiden Enden der Verbindungslinie ein Pfeil. Es ist auch möglich die Pfeilspitzen wegzulassen, das ist bei unwichtigen Beziehungen teilweise der Fall.

3.4.2 Komposition



Hier ist eine `Komposition` dargestellt. Eine Komposition bedeutet, dass eine Klasse ohne eine andere Klasse nicht auskommt. Die Raute befindet sich an dem Ende der Verbindungslinie, an dem die abhängige Klasse steht. Nehmen wir als Beispiel die Klassen Mensch und Erde. Ein Mensch muss auf der Erde leben, die Erde kann aber auch ohne Menschen auskommen.

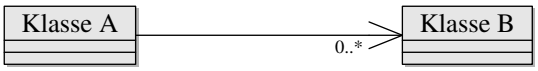
3.4.3 Vererbung



Der Vollständigkeit halber soll hier auch einmal die Beziehung im Sinne der Vererbung gezeigt werden. Da Vererbung aber noch nicht in der Vorlesung besprochen wurde, wird hierauf im Weiteren nicht näher eingegangen.

3.4.4 Assoziationen verfeinern

Oben haben wir gesehen, dass man beschreiben kann, ob eine Klasse von einer Anderen abhängig ist. Möchte man jedoch genauer sagen, in welchem Maße diese Abhängigkeit steht, kann man sog. `Multiplizitäten` angeben. Es gibt `1:1`, `1:n` und `n:n` Beziehungen. Steht keine genauere Angabe an den Verbindunglinien, wird von einer 1:1 Beziehung ausgegangen. Das bedeutet, dass jedes Objekt einer Klasse mit genau einem Objekt der anderen Klasse in Kontakt steht. Eine 1:n Beziehung wird angegeben, wenn ein Objekt einer Klasse mit mehreren Objekten einer anderen Klasse in Kontakt steht. Somit ist auch klar, dass eine n:n Beziehung aussagt, dass Objekte beider Klassen Zugriff auf mehrere Objekte der anderen Klasse haben. Wie bereits beschrieben, muss man 1:1 Beziehungen nicht explizit angeben. Um eine 1:n Beziehung darzustellen, schreibt man ein 0..* an das navigierbare Ende. 0..* liest man wie $n \in N_0$.

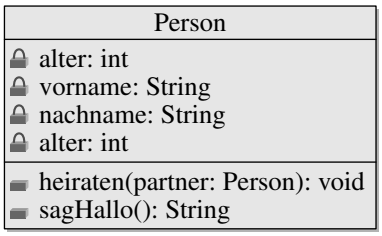


Man kann somit aber auch Ober- und Untergrenzen festlegen, also beispielsweise 2..5 (mindestens 2, aber maximal 5).

3.4.5 Klassen beschreiben

Eine Klasse ist ein Rechteck und hat drei Unterbereiche, welche oben bisher leer gelassen wurden. So gibt es den `Klassennamen`, darunter befinden sich die `Eigenschaften` und im letzten Unterbereich befinden sich die `Methoden`.

Eigenschaften beschreibt man, indem man den Namen angibt, danach kommt ein Doppelpunkt und zuletzt der Datentyp. Bei den Methoden sieht es fast genau so aus, hier kommt zuerst der Methodenkopf und nach dem Doppelpunkt dann der Rückgabewert. Wie bei Java auch, benutzt man bei Klassendiagrammen `void`, falls man keinen Wert zurück gibt. Hier ist eine Beispielklasse mit verschiedenen Eigenschaften und einer Methode.



Hier haben wir eine Klasse `Person` mit den privaten Eigenschaften `vorname` und `nachname` vom Typ `String`, der Eigenschaft `alter` vom Typ int und eine Liste von `Personen`, die als `freunde` bezeichnet werden. Alle diese Eigenschaften sind private, bei dem hier benutzten Programm wird dafür ein Schloss verwendet, jedoch ist es üblicherweise einfach ein - (Minuszeichen).
Zudem besitzt diese Klasse zwei Methoden, die beide `public` sind. Die Methode `heiraten()`, die den Rückhabewert void hat (also nichts zurück gibt), nimmt einen Parameter `partner`, der vom Typ `Person`, entgegen. Und die Methode `sagHallo()` gibt einen String zurück.
Man sieht, dass die Klassendiagramme einen guten ersten Eindruck über Klassen geben und man die Logik die hinter den einzelnen Methoden steht, nicht unbedingt verstehen muss.